SC09-1525-00

IBM

AS/400

ILE RPG/400 Programmer's Guide

Version 3

Take Note!

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

First Edition (September 1994)

This edition to Version 3 Release 1 Modification 0, of IBM Application System/400 ILE RPG/400 (program 5763-RG1) and to all subsequent releases and modifications until otherwise indicated in new editions. Make sure you are using the proper edition for the level of the product.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address given below.

A form for readers' comments is provided at the back of this publication. If the form has been removed, address your comments to:

IBM Canada Ltd. Laboratory, Information Development
2G/345/1150/TOR
1150 Eglinton Avenue East
North York, Ontario, Canada. M3C 1H7

You can also send your comments by facsimile (attention: RCF Coordinator), or you can send your comments electronically to IBM. See "Communicating Your Comments to IBM" for a description of the methods. This page immediately precedes the Readers' Comment Form at the back of this publication.

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

Notices

Any reference to an IBM licensed program in this publication is not intended to state or imply that only IBM's program or other product may be used. Any functionally equivalent product, program or service that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, is the user's responsibility.

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, 208 Harbor Drive, Stamford, Connecticut, USA 06904-2501.

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.

Programming Interface Information

This publication is intended to help you create programs using RPG IV source. This publication documents *general-use programming interfaces and associated guidance information* provided by the ILE RPG/400 compiler.

General-use programming interfaces allow you to write programs that request or receive services of the ILE RPG/400 compiler.

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:

Application System/400	AS/400
C/400	COBOL/400
DB2	GDDM
IBM	ILE
Integrated Language Environment	Operating System/2
Operating System/400	OS/2
OS/400	RPG IV
RPG/400	400

About This Guide

This guide provides information that shows how to use ILE RPG/400* programming language in the Integrated Language Environment*. (ILE RPG/400 language is an implementation of the RPG IV* language on the AS/400* system with the Operating System/400* (OS/400*) operating system.) Use this guide to create and run ILE* applications using RPG IV source.

This guide shows how to:

- Enter RPG IV source statements
- Create modules
- Bind modules
- Run an ILE program
- Call other objects
- Debug an ILE program
- Handle exceptions
- Define and process files
- Access devices
- Convert programs from an RPG III format to RPG IV format
- Read compiler listings

This guide may refer to products that are announced but are not yet available.

You may need to refer to other IBM* guides for more specific information about a particular topic. The *Publications Ordering*, SC41-3000, provides information on all of the guides in the AS/400 library. For a list of related publications, see the bibliography on page 371.

Who Should Use This Guide

This guide is for programmers who are familiar with the RPG/400 programming language, but who want to learn how to use it in the ILE framework. This guide is also for programmers who want to convert programs from the RPG III to the RPG IV format. It is designed to guide you in the use of the ILE RPG/400 compiler on the AS/400 system.

Though this guide shows how to use the RPG IV in an ILE framework, it does not provide detailed information on RPG IV specifications and operations. For a detailed description of the language, see the *ILE RPG/400 Reference* SC09-1526.

Before using this guide, you should

- Know how to use applicable AS/400 menus and displays or Control Language (CL) commands.
- Have the appropriate authority to the CL commands and objects described here.
- Have a firm understanding of ILE as described in detail in the *ILE Concepts*, SC41-3606.

ILE RPG/400 Introduction

Before using ILE RPG/400 to create a program, you must know certain aspects of the environment in which you will be using it. This part provides information on the following topics that you should know:

- Overview of RPG IV language
- Role of ILE components in RPG programming
- ILE program creation strategies

Chapter 1. Overview of the RPG IV Programming Language

This chapter presents a high-level review of the features of the RPG IV programming language that distinguish RPG from other programming languages. You should be familiar and comfortable with all of these features before you program in the RPG IV language. The features discussed here encompass the following subjects:

- Coding specifications
- The program cycle
- Indicators
- Operation codes

For more information on RPG IV, see *ILE RPG/400 Reference*.

RPG IV Specifications

RPG code is written on a variety of specification forms, each with a specific set of functions. Many of the entries which make up a specification type are position-dependent. Each entry must start in a specific position depending on the type of entry and the type of specification.

There are six types of RPG IV specifications. Each specification type is optional. Specifications must be entered into your source program in the order shown below.

1. **Control specifications** provide the compiler with information about generating and running programs, such as the program name, date format, and use of alternate collating sequence or file translation.
2. **File description specifications** describe all the files that your program uses.
3. **Definition specifications** describe the data used by the program.
4. **Input specifications** describe the input records and fields used by the program.
5. **Calculation specifications** describe the calculations done on the data and the order of the calculations. Calculation specifications also control certain input and output operations.
6. **Output specifications** describe the output records and fields used by the program.

Cycle Programming

When a system processes data, it must do the processing in a particular order. This logical order is provided by:

- The RPG/400 compiler
- Your program code.

The logic the compiler supplies is called the **program cycle**. When you let the compiler provide the logic for your programs, it is called **cycle programming**.

The program cycle is a series of steps that your program repeats until an end-of-file condition is reached. Depending on the specifications you code, the program may or may not use each step in the cycle.

If you want to have files controlled by the cycle, the information that you code on RPG specifications in your source program need not specify when records for these files are read. The compiler supplies the logical order for these operations, and some output operations, when your source program is compiled.

If you do not want to have files controlled by the cycle, you must create an end-of-file condition, usually by setting on the last record (LR) indicator.

Figure 1 shows the specific steps in the general flow of the RPG program cycle.

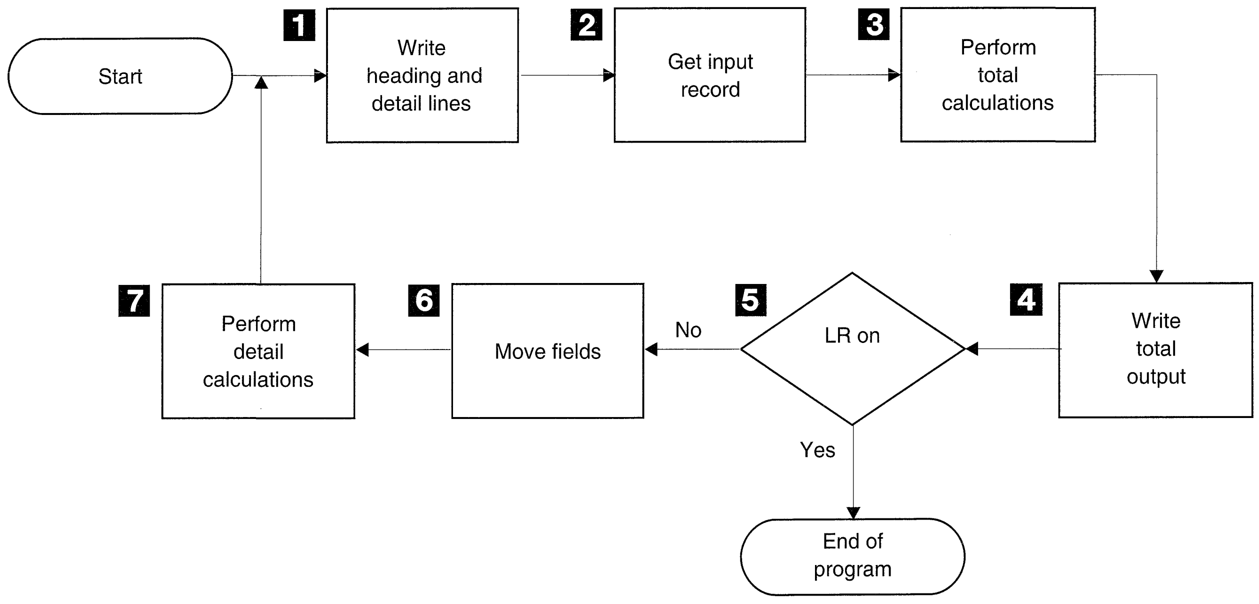

Start
1
Write heading and detail lines
2
Get input record
3
Perform total calculations
4
Write total output
5
LR on
No
Yes
End of program
6
Move fields
7
Perform detail calculations

Figure 1. RPG Program Logic Cycle

1 RPG processes all heading and detail lines (H or D in position 17 of the output specifications).

2 RPG reads the next record and sets on the record identifying and control level indicators.

3 RPG processes total calculations (conditioned by control level indicators L1 through L9, an LR indicator, or an L0 entry).

4 RPG processes all total output lines (identified by a T in position 17 of the output specifications).

5 RPG determines if the LR indicator is on. If it is on, the program ends.

6 The fields of the selected input records move from the record to a processing area. RPG sets on field indicators.

7 RPG processes all detail calculations (not conditioned by control level indicators in positions 7 and 8 of the calculation specifications). It uses the data from the record at the beginning of the cycle.

The first cycle

The first and last time through the program cycle differ somewhat from other cycles. Before reading the first record the first time through the cycle, the program does three things:

- handles input parameters, opens files, initializes program data
- writes the records conditioned by the 1P (first page) indicator
- processes all heading and detail output operations.

For example, heading lines printed before reading the first record might consist of constant or page heading information, or special fields such as PAGE and *DATE. The program also bypasses total calculations and total output steps on the first cycle.

The last cycle

The last time a program goes through the cycle, when no more records are available, the program sets the LR (last record) indicator and the L1 through L9 (control level) indicators to ***on***. The program processes the total calculations and total output, then all files are closed, and then the program ends.

Indicators

An **indicator** is a one-byte character field that is either set on ('1') or off ('0'). Each indicator has a two-character name (for example, LR, 01, H3), and is referred to in some entries of some specifications just by the two-character name, and in others by the special name *INxx where xx is the two-character name. An indicator is either the result of an operation or is used to condition (or control) the processing of an operation. Indicators are like switches in the flow of the program logic. They determine the path the program will take during processing, depending on how they are set.

You can use several types of indicators; each type signals something different. In an RPG program, indicators are defined either by making entries on the specifications or by the RPG program cycle itself. The positions on the specification in which you define an indicator determine the use of the indicator. Once you define an indicator in your program, it can limit or control calculation and output operations.

An RPG program sets and resets certain indicators at specific times during the program cycle. In addition, the state of indicators can be changed explicitly in calculation operations.

Operation Codes

The RPG IV programming language allows you to do many different types of operations on your data. **Operation codes**, entered on the calculation specifications, indicate what operations will be done. For example, if you want to read a new record, you could use the `READ` operation code. The following is a list of the types of operations available.

```
Arithmetic operations
Array operations
Bit operations
Branching operations
Call operations
Compare operations
Data-Area operations
Date/Time/Timestamp operations
Declarative operations
File operations
Indicator-setting operations
Information operations
Initialization operations
Message operation
Move operations
String operations
Structured programming operations
Subroutine operations
Test operations
```

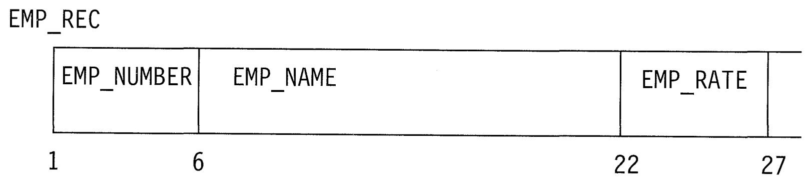
Example of an ILE RPG/400 Program

This section illustrates a simple ILE RPG/400 program that performs payroll calculations.

Problem Statement

The payroll department of a small company wants to create a print output that lists employees' pay for that week. Assume there are two disk files, EMPLOYEE and TRANSACT, on the system.

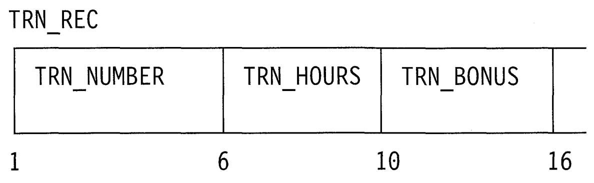
The first file, EMPLOYEE, contains employee records. The figure below shows the format of an employee record:



```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ..*
A..........T.Name++++++RLen++TDpB......Functions++++++++++++++++++++*
A          R EMP_REC
A            EMP_NUMBER     5         TEXT('EMPLOYEE NUMBER')
A            EMP_NAME      16         TEXT('EXPLOYEE NAME')
A            EMP_RATE       5  2      TEXT('EXPLOYEE RATE')
A          K EMP_NUMBER
```

Figure 2. DDS for Employee physical file

The second file, TRANSACT, tracks the number of hours each employee worked for that week and any bonus that employee may have received. The figure below shows the format of a transaction record:



```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ..*
A.........T.Name++++++RLen++TDpB......Functions++++++++++++++++++++++*
A         R TRN_REC
A           TRN_NUMBER      5         TEXT('EMPLOYEE NUMBER')
A           TRN_HOURS       4  1      TEXT('HOURS WORKED')
A           TRN_BONUS       6  2      TEXT('BONUS')
```

Figure 3. DDS for TRANSACT physical file

Each employee's pay is calculated by multiplying the "hours" (from the TRANSACT file) and the "rate" (from the EMPLOYEE file) and adding the "bonus" from the TRANSACT file.

Control Specifications

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8
HKeywords++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
H DATEDIT(*DMY/)
```

Today's date will be printed in day, month, year format with "/" as the separator.

File Description Specifications

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+...
FFilename++IPEASFRlen+LKlen+AIDevice+.Keywords+++++++++++++++++++++++++++

FTRANSACT  IP   E           K DISK
FEMPLOYEE  IF   E           K DISK
FQSYSPRT   O    F   80        PRINTER
```

There are three files defined on the File Description Specifications:

- The TRANSACT file is defined as the Input Primary file. The ILE RPG/400 program cycle controls the reading of records from this file.
- The EMPLOYEE file is defined as the Input Full-Procedure file. The reading of records from this file is controlled by operations in the Calculation Specifications.
- The QSYSPRT file is defined as the Output Printer file.

Definition Specifications

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+...
D+Name++++++++++ETDsFrom+++To/L+++IDc.Keywords+++++++++++++++++++++++++

D Pay             S              8P 2
D Heading1        C                   'NUMBER  NAME              RATE    H-
D                                     OURS  BONUS    PAY        '
D Heading2        C                   '______  ________________  ______  _-
D                                     ____  _______  __________'
```

Using the Definition Specifications, declare a variable called "Pay" to hold an employees' weekly pay and two constants "Heading1" and "Heading2" to aid in the printing of the report headings.

Calculation Specifications

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+...
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq..

C     TRN_NUMBER    CHAIN     EMP_REC                            99
C                   IF        NOT *IN99
C                   EVAL (H)  Pay = EMP_RATE * TRN_HOURS + TRN_BONUS
C                   ENDIF
```

The coding entries on the Calculation Specifications include:

- Using the CHAIN operation code, the field TRN_NUMBER from the transaction file is used to find the record with the same employee number in the employee file.
- If the CHAIN operation is successful (that is, indicator 99 is off), the pay for that employee is evaluated. The result is "rounded" and stored in the variable called Pay.

Output Specifications

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+...
OFilename++DF..N01N02N03Excnam++++B++A++Sb+Sa+..........................
O..............N01N02N03Field+++++++++YB.End++PConstant/editword/DTformat

OQSYSPRT   H    1P                    2  3
O                                          35 'PAYROLL REGISTER'
O                        *DATE        Y    60
O          H    1P                    2
O                                          60 Heading1
O          H    1P                    2
O                                          60 Heading2
O          D   N1PN99                 2
O                        TRN_NUMBER         5
O                        EMP_NAME          24
O                        EMP_RATE     L    33
O                        TRN_HOURS    L    40
O                        TRN_BONUS    L    49
O                        Pay               60 '$      0.  '
O          D   N1P 99                 2
O                        TRN_NUMBER         5
O                                          35 '** NOT ON EMPLOYEE FILE **'
O          T    LR
O                                          33 'END OF LISTING'
```

The Output Specifications describe what fields are to be written on the QSYSPRT output:

- The Heading Lines that contain the constant string 'PAYROLL REGISTER' as well as headings for the detail information will be printed if indicator 1P is on. Indicator 1P is turned on by the ILE RPG/400 program cycle during the first cycle.
- The Detail Lines are conditioned by the indicators 1P and 99. Detail Lines are not printed at 1P time. The N99 will only allow the Detail lines to be printed if indicator 99 is off, which indicates that the corresponding employee record has been found. If the indicator 99 is on, then the employee number and the constant string '** NOT ON EMPLOYEE FILE **' will be printed instead.
- The Total Line contains the constant string 'END OF LISTING'. It will be printed during the last program cycle.

The Entire Source Program

The following figure combines all the specifications used in this program. This is what you should enter into the source file for this program.

```
 *-------------------------------------------------------------------------*
 * DESCRIPTION:  This program creates a printed output of employee's pay  *
 *               for the week.                                             *
 *-------------------------------------------------------------------------*

H DATEDIT(*DMY/)

 *-------------------------------------------------------------------------*
 * File Definitions                                                        *
 *-------------------------------------------------------------------------*
FTRANSACT  IP   E           K DISK
FEMPLOYEE  IF   E           K DISK
FQSYSPRT   O    F   80        PRINTER

 *-------------------------------------------------------------------------*
 * Variable Declarations                                                   *
 *-------------------------------------------------------------------------*
D Pay             S              8P 2

 *-------------------------------------------------------------------------*
 * Constant Declarations                                                   *
 *-------------------------------------------------------------------------*
D Heading1        C                   'NUMBER  NAME               RATE    H-
D                                     OURS  BONUS    PAY       '
D Heading2        C                   '______  ________________  ______  _-
D                                     ____  _______  __________'

 *-------------------------------------------------------------------------*
 * For each record in the transaction file (TRANSACT), if the employee    *
 * is found, compute the employees pay and print the details.              *
 *-------------------------------------------------------------------------*
C     TRN_NUMBER    CHAIN     EMP_REC                            99
C                   IF        NOT *IN99
C                   EVAL (H)  Pay = EMP_RATE * TRN_HOURS + TRN_BONUS
C                   ENDIF
```

Figure 4 (Part 1 of 2). A Sample Payroll Calculation Program

```
*-----------------------------------------------------------------------*
* Report Layout                                                         *
*  -- print the heading lines if 1P is on                               *
*  -- if the record is found (indicator 99 is off) print the payroll    *
*     details otherwise print an exception record                       *
*  -- print 'END OF LISTING' when LR is on                              *
*-----------------------------------------------------------------------*
OQSYSPRT   H    1P                    2  3
O                                          35 'PAYROLL REGISTER'
O                        *DATE        Y    60
O          H    1P                    2
O                                          60 Heading1
O          H    1P                    2
O                                          60 Heading2
O          D   N1PN99                 2
O                        TRN_NUMBER         5
O                        EMP_NAME          24
O                        EMP_RATE     L    33
O                        TRN_HOURS    L    40
O                        TRN_BONUS    L    49
O                        Pay               60 '$     0.  '
O          D   N1P 99                 2
O                        TRN_NUMBER         5
O                                          35 '** NOT ON EMPLOYEE FILE **'
O          T    LR
O                                          33 'END OF LISTING'
```

Figure 4 (Part 2 of 2). A Sample Payroll Calculation Program

Using the OS/400 System

The operating system that controls all of your interactions with the AS/400 system is called the Operating System/400 (OS/400) system. From your workstation, the OS/400 system allows you to:

- Sign on and sign off
- Interact with the displays
- Use the online help information
- Enter control commands and procedures
- Respond to messages
- Manage files
- Run utilities and programs.

Refer to the *Publications Ordering*, SC41-3000 for a complete list of publications that discuss the OS/400 system.

Interacting with the System

You can manipulate the OS/400 system using Command Language (CL). You interact with the system by entering or selecting CL commands. The AS/400 system often displays a series of CL commands or command parameters appropriate to the situation on the screen. You then select the desired command or parameters.

Commonly Used Control Language Commands

The following table lists some of the most commonly used CL commands, their function, and the reasons you might want to use them.

Table 1. Commonly Used CL Commands

Action	CL command	Result
Using System Menus	GO MAIN	Display main menu
	GO INFO	Display help menu
	GO CMDRPG	List commands for RPG
	GO CMDCRT	List commands for creating
	GO CMDxxx	List commands for 'xxx'
Calling	CALL `program-name`	Runs a program
Compiling	CRTxxxMOD	Creates xxx Module
	CRTBNDxxx	Creates Bound xxx Program
Binding	CRTPGM	Creates a program from ILE modules
	CRTSRVPGM	Creates a service program
	UPDPGM	Updates a bound program object
Debugging	STRDBG	Starts ILE source debugger
	ENDDBG	Ends ILE source debugger
Creating Files	CRTPRTF	Creates Print File
	CRTPF	Creates Physical File
	CRTSRCPF	Creates Source Physical File
	CRTLF	Creates Logical File

AS/400 Tools

The AS/400 system offers a full set of tools that you may find useful for programming. The following products are available to help you develop ILE RPG/400 applications more effectively. See "Bibliography" on page 371 for the publications associated with these products.

Application Development Toolset/400 (ADTS/400)

The Application Development Toolset/400 (ADTS/400) provides an integrated suite of host-based tools designed to meet the needs of the application developer. This product provides tools for manipulating source, objects, and database files on the AS/400 system. Some of the tools provided are: PDM, SEU, and SDA. A menu driven interface is available from which you can perform all of the tasks involved in application development, such as object management, editing, compiling and debugging.

Application Development Manager/400

The Application Development Manager/400 provides application development organizations with a mechanism for efficient and effective management of application objects throughout the life of the application. This feature allows a group of developers to create, manage, and organize multiple versions of their application through the Programming Development Manager (PDM) interface or directly from the AS/400 command line.

CoOperative Development Environment/400

The CoOperative Development Environment/400 (CODE/400) enhances program development and moves the program development workload off the host. For RPG application development and maintenance, CODE/400 provides:

- language sensitive editing— includes token highlighting, format lines, a full suite of prompts, and online help.
- incremental syntax checking— provides immediate error feedback as each line of source is entered
- program verification— performs, at the workstation, the full range of syntax and semantic checking that the compiler does, without generating object code
- an OS/2* interface for submitting host compiles and binds
- source-level debugging
- a DDS design utility—allows you to easily change screens, reports, and database files
- access to Application Development Manager/400.

Chapter 2. RPG Programming in ILE

ILE RPG/400 is an implementation of the RPG IV programming language in the Integrated Language Environment. It is one of the family of ILE compilers available on the AS/400 system.

ILE is a new approach to programming on the AS/400 system. It is the result of major enhancements to the AS/400 machine architecture and the OS/400 operating system. The ILE family of compilers includes: ILE RPG/400, ILE C/400*, ILE COBOL/400*, and ILE CL. Figure 5 depicts the ILE enhancement to the operating system in support of the ILE languages. It shows that support for the original program model (OPM) and extended program model (EPM) languages is retained.

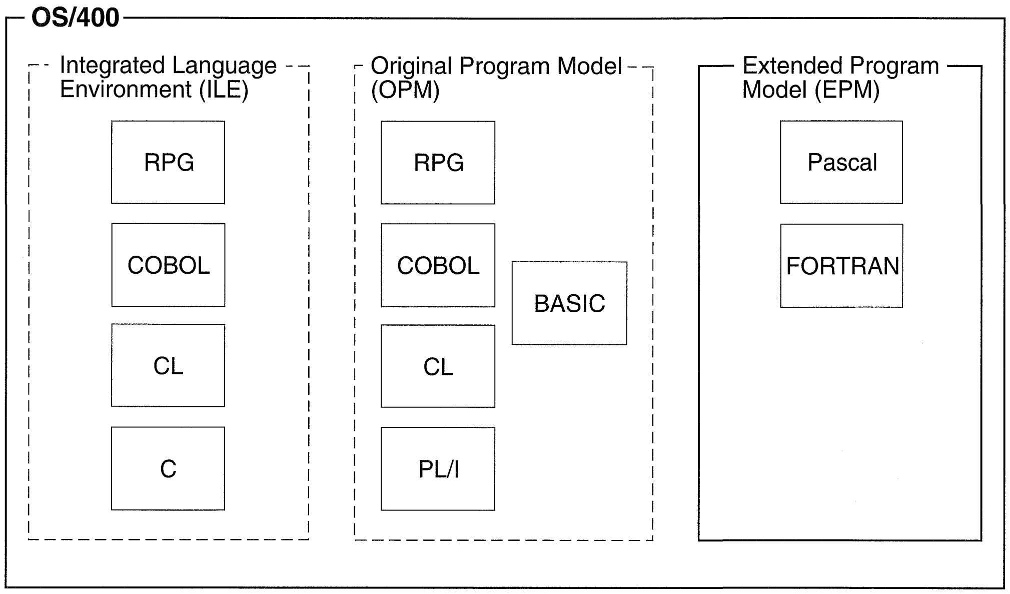

OS/400
Integrated Language Environment (ILE)
RPG
COBOL
CL
C
Original Program Model (OPM)
RPG
COBOL
BASIC
CL
PL/I
Extended Program Model (EPM)
Pascal
FORTRAN

Figure 5. Integrated Language Environment and its place in the Operating System

ILE provides RPG users with improvements or enhancements in the following areas of application development:

Program creation
Program management
Program call
Source debugging
Bindable application program interfaces (APIs)

Each of the above areas is explained briefly in the following paragraphs and discussed further in the following chapters.

Program Creation

In ILE, program creation consists of

1. Compiling source code into modules
2. Binding (combining) one or more modules into a program object

You can create a program object much like in the OPM framework with a one-step process using the Create Bound RPG Program (CRTBNDRPG) command. This command creates a temporary module which is then bound into a program object. It also allows you to bind other objects through the use of a binding directory.

Alternatively, you may create a program using separate commands for compilation and binding. This two-step process allows you to reuse a module or update one module without recompiling the other modules in a program. In addition, because you can combine modules from any ILE language, you can create and maintain mixed-language programs.

In the two-step process, you create a module object using the Create RPG Module (CRTRPGMOD) command. This command compiles the source statements into a module object. A module is a nonrunnable object; it must be bound into a program object to be run. To bind one or more modules together use the Create Program (CRTPGM) command.

Modules can also be bound into a service program. Service programs are a means of packaging callable routines (functions or procedures) into a separately bound program object. The use of service programs provides modularity and maintainability. They enable you to use off-the-shelf modules developed by third parties or, conversely, to package your modules for third-party use. A service program is created using the Create Service Program (CRTSRVPGM) command.

Figure 6 shows the two approaches to program creation.

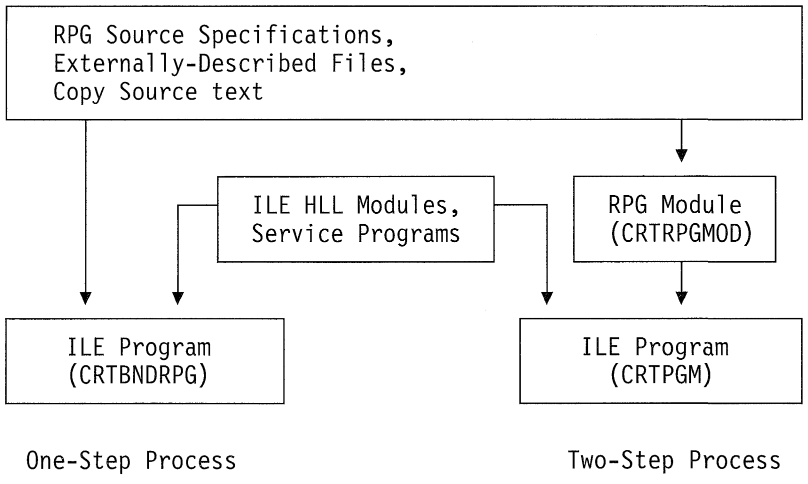


Figure 6. Program Creation in ILE

Once a program is created you can update the program using the UPDPGM or UPDSRVPGM commands. This is useful, because it means you only need to have the new or changed module objects available to update the program.

For more information on the one-step process, see Chapter 5, "Creating a Program with the CRTBNDRPG Command" on page 37. For more information on the two-step process, see Chapter 6, "Creating a Program with the CRTRPGMOD and CRTPGM Commands" on page 49. For more information on service programs, see Chapter 7, "Creating a Service Program" on page 63.

Program Management

ILE provides a common basis for managing program flow, sharing resources, and handling language semantics during the run time of a program. For RPG users this means that you can have much better control over resources than was previously available.

ILE programs must be activated into activation groups, which are specified at program creation time. Activation is the allocation of working storage within a job so that one or more programs can run in that space. If the specified activation group for a program does not exist when the program is called, then it is created within the job to hold the program's activation.

An activation group is the key element in governing an ILE application's resources and behavior. For example, you can scope commitment control operations to the activation group level. You can also scope file overrides and shared open data paths to the activation group of the running application. Finally, the behavior of a program upon termination is also affected by the activation group in which the program runs.

For more information on activation groups, see “Managing Activation Groups” on page 79.

You can dynamically allocate storage for a runtime array using the bindable APIs provided for all ILE programming languages. These APIs allow single- and mixed-language applications to access a central set of storage management functions and offers a storage model to languages, such as RPG, that do not now provide one.

Program Call

With ILE, you can write RPG applications where ILE RPG/400 programs and OPM RPG/400 programs continue to interrelate through the traditional use of dynamic program calls. Programs call other programs by using dynamic calls. They specify the name of the called program on a CALL statement. The called program's name is resolved to an address at run time, just before the calling program passes control to the called program.

However, you can also write ILE applications which can interrelate with faster static calls. Static calls involve calls between procedures. A procedure is a self-contained set of code which performs a task and then returns to the caller. An ILE RPG/400 module consists of one procedure. Because the procedure names are resolved at bind time, (that is, when you create the program) static calls are faster than dynamic calls.

Static calls also allow operational descriptors, omitted parameters, and allow a greater number of parameters to be passed. (Omitted parameters are parameters which serve as placeholders when no data is actually passed.) Operational descriptors and omitted parameters can be useful when calling bindable APIs or procedures written in other ILE languages. To enable procedure calls and the omitting of parameters, RPG IV provides the CALLB operation.

For information on running a program refer to Chapter 8, “Running a Program” on page 73. For information on program/procedure call, refer to Chapter 9, “Calling Programs and Procedures” on page 91.

Source Debugging

With ILE, you can perform source-level debugging on any single- or mixed-language ILE application. You can control the flow of a program by using debug commands while the program is running. You can set conditional and unconditional breakpoints prior to running the program. Then after you call the program, you can step through a specified number of statements, and display or change variables. When a program stops because of a breakpoint, a step command, or a runtime error, the pertinent module is shown on the display at the point where the program stopped. At that point, you can enter more debug commands.

For information on the debugger, refer to Chapter 10, "Debugging Programs" on page 113.

Bindable APIs

ILE offers a number of bindable APIs that can be used to supplement the function currently offered by ILE RPG/400. The bindable APIs provide program calling and activation capability, condition and storage management, math functions, and dynamic screen management.

Some APIs which you may wish to consider using in an ILE RPG/400 application include:

CEETREC — Signal the Termination-Imminent Condition
CEE4ABN — Abnormal End
CEEFRST — Free Storage
CEEGTST — Get Heap Storage
CEECZST — Reallocate Storage
CEEDOD — Decompose Operational Descriptor

For more information on these ILE bindable APIs, see Chapter 8, "Running a Program" on page 73 and also the *System API Reference*.

Chapter 3. Program Creation Strategies

There are many approaches you can take in creating programs using an ILE language. This section presents three common strategies for creating ILE programs using ILE RPG/400 or other ILE languages.

1. Create a program using CRTBNDRPG to maximize OPM compatibility.
2. Create an ILE program using CRTBNDRPG.
3. Create an ILE program using CRTRPGMOD and CRTPGM.

The first strategy is recommended as a temporary one. It is intended for users who have OPM applications and who, perhaps due to lack of time, cannot move their applications to ILE all at once. The second strategy can also be a temporary one. It allows you time to learn more about ILE, but also allows you to immediately use some of its features. The third strategy is more involved, but offers the most flexibility.

Both the first and second strategy make use of the one-step program creation process, namely, CRTBNDRPG. The third strategy uses the two-step program creation process, namely, CRTRPGMOD followed by CRTPGM.

Strategy 1: OPM-Compatible Application

Strategy 1 results in an ILE program which interacts well with OPM programs. It allows you to take advantage of RPG IV enhancements, but not all of the ILE enhancements. You may want such a program temporarily while you complete your migration to ILE.

Method

Use the following general approach to create such a program:

1. Convert your source to RPG IV using the CVTRPGSRC command.

   Be sure to convert all /COPY members which are used by the source you are converting.
2. Create a program object using the CRTBNDRPG command, specifying DFTACTGRP(*YES).

Specifying DFTACTGRP(*YES) means that the program object will run only in the default activation group. (The default activation group is the activation group where all OPM programs are run.) As a result, the program object will interact well with OPM programs in the areas of override scoping, open scoping, and RCLRSC.

When you use this approach you cannot make use of ILE static binding. This means that your program cannot contain a CALLB operation, nor can you use the BNDDIR or ACTGRP parameters on the CRTBNDRPG command when creating this program.

Example of OPM-Compatible Program

Figure 7 shows the run-time view of a sample application where you might want an OPM-compatible program. The OPM application consisted of a CL program and two RPG programs. In this example, one of the RPG programs has been moved to ILE; the remaining programs are unchanged.

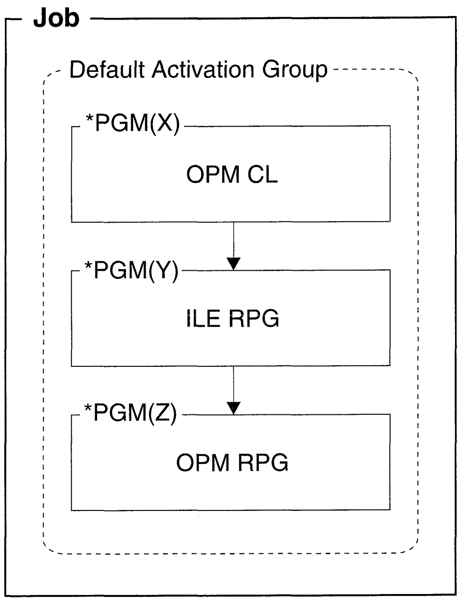


Figure 7. OPM-Compatible Application

Effect of ILE

The following deals with the effects of ILE on the way your application handles:

Program call OPM programs behave as before. The system automatically creates the OPM default activation group when you start your job, and all OPM applications run in it. One program can call another program in the default activation group by using a dynamic call.

Data Storage for static data is created when the program is activated, and it exists until the program is deactivated. When the program ends (either normally or abnormally), the program's storage is deleted. To clean up storage for a program which returns without ending, use the Reclaim Resource (RCLRSC) command.

Files File processing is the same as in previous releases. Files are closed when the program ends normally or abnormally.

Errors As in previous releases, the compiler handles errors within each program separately. The errors you see that originated within your program are the same as before. However, the errors are now communicated between programs by the ILE condition manager, so you may see different messages between programs. The messages may have new message IDs, so if your CL program monitors for a specific message ID, you may have to change that ID.

Related Information

Converting to RPG IV	"Converting Your Source" on page 310
One-step creation process	Chapter 5, "Creating a Program with the CRTBNDRPG Command" on page 37
ILE static binding	Chapter 9, "Calling Programs and Procedures" on page 91; also *ILE Concepts*
Exception handling differences	"Differences between OPM and ILE RPG/400 Exception Handling" on page 156

Strategy 2: ILE Program Using CRTBNDRPG

Strategy 2 results in an ILE program which can take advantage of ILE static binding. Your source can contain static procedure calls using CALLB because you can bind the module to other modules or service programs using a binding directory. You can also specify the activation group in which the program will run.

Method

Use the following general approach to create such a program:

1. If starting with RPG III source, convert your source to RPG IV using the CVTRPGSRC command.

   If converting, be sure to convert all /COPY members and any programs which are called by the source you are converting. Also, if you are using CL to call the program, you should also make sure that you are using ILE CL instead of OPM CL.

2. Determine the activation group the program will run in.

   You may want to name it after the application name, as in this example.

3. Identify the name of the binding directory, if any, to be used.

   It is assumed with this approach that if you are using a binding directory, it is one which is already created for you. For example, there may be a third-party program to which you want to bind your source. Consequently, all you need to know is the name of the binding directory.

4. Create an ILE program using CRTBNDRPG, specifying DFTACTGRP(*NO), the activation group on the ACTGRP parameter, and the binding directory, if any, on the BNDDIR parameter.

Note that if ACTGRP(*CALLER) is specified and this program is called by a program running in the default activation group, then this program will behave according to ILE semantics in the areas of override scoping, open scoping, and RCLRSC.

The main drawback of this strategy is that you do not have a permanent module object which you can later reuse to bind with other modules to create an ILE program. Furthermore, any procedure calls must be to modules or service programs which are identified in a binding directory. If you want to bind two or more modules without using a binding directory you need to use the third strategy.

Example of ILE Program Using CRTBNDRPG

Figure 8 shows the run-time view of an application in which an ILE CL program calls an ILE RPG/400 program which is bound to a supplied service program. The application runs in the named activation group XYZ.

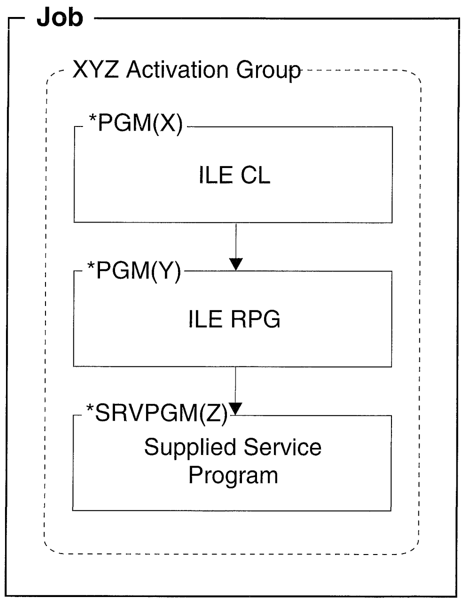


Figure 8. ILE Program Using CRTBNDRPG

Effect of ILE

The following deals with the effects of ILE on the way your program handles:

Program call The system automatically creates the activation group if it does not already exist, when the application starts.

The application can contain dynamic program calls or static procedure calls. Procedures within bound programs call each other by using static calls. Procedures call ILE and OPM programs by using dynamic calls.

Data The lifetime of a program's storage is the same as the lifetime of the activation group. Storage remains active until the activation group is deleted.

The ILE RPG/400 run time manages data so that the semantics of ending programs and reinitializing the data are the same as for OPM RPG, although the actual storage is not deleted as it was when an OPM RPG program ended. Data is reinitialized if the previous call to the procedure ended with LR on, or ended abnormally.

Program data which is identified as exported or imported (using the keywords EXPORT and IMPORT respectively) is external to the individual modules. It is known among the modules that are bound into a program.

Files By default, file processing (including opening, sharing, overriding, and commitment control) by the system is scoped to the activation group level. You cannot share files at the data management level with programs in different activation groups. If you want to share a file across activation groups, you must open it at the job level by specifying SHARE(*YES) on an override command or create the file with SHARE(*YES).

Errors

When you call an ILE RPG program or procedure in the same activation group, if it gets an exception that would previously have caused it to display an inquiry message, now your calling program will see that exception first.

If your calling program has an error indicator or *PSSR, the program or procedure that got the exception will end abnormally without the inquiry message being displayed. Your calling program will behave the same (the error indicator will be set on or the *PSSR will be invoked).

When you call an OPM program or a program or procedure in a different activation group, the exception handling will be the same as in OPM RPG, with each program handling its own exceptions. The messages you see may have new message IDs, so if you monitor for a specific message ID, you may have to change that ID.

Each language processes its own errors and can process the errors which occur in modules written in another ILE language. For example, RPG will handle any C errors if an error indicator has been coded. C can handle any RPG errors.

Related Information

Converting to RPG IV	“Converting Your Source” on page 310
One-step creation process	Chapter 5, “Creating a Program with the CRTBNDRPG Command” on page 37
Activation groups	“Managing Activation Groups” on page 79
RCLRSC	“Reclaim Resources Command” on page 82
ILE static binding	Chapter 9, “Calling Programs and Procedures” on page 91; also *ILE Concepts*
Exception handling differences	“Differences between OPM and ILE RPG/400 Exception Handling” on page 156
Override and open scope	“Overriding and Redirecting File Input and Output” on page 197 and “Sharing an Open Data Path” on page 201; also *ILE Concepts*

Strategy 3: ILE Application Using CRTRPGMOD

This strategy allows you to fully utilize the concepts offered by ILE. However, while being the most flexible approach, it is also more involved. This section presents three scenarios for creating:

A single-language application
A mixed-language application
An advanced application

The effect of ILE is the same as described in "Effect of ILE" on page 22.

You may want to read about the basic ILE concepts in *ILE Concepts* before using this approach.

Method

Because this approach is the most flexible, it includes a number of ways in which you might create an ILE application. The following list describes the main steps which you may need to perform:

1. Create a module from each source member using the appropriate command, for example, CRTRPGMOD for RPG source, CRTCLMOD for CL source, etc..
2. Determine the ILE characteristics for the application, for example:
   - Determine which module will be the starting point for the application. The module you choose as the entry module is the first one that you want to get control. In an OPM application, this would be the command processing program, or the program called because a menu item was selected.
   - Determine the activation group the application will run in. (Most likely you will want to run in a named activation group, where the name is based on the name of the application.)
   - Determine the exports and imports to be used.
3. Determine if any of the modules will be bound together to create a service program. If so, create the service programs using CRTSRVPGM.
4. Bind the appropriate modules and service programs together using CRTPGM, specifying values for the parameters based on the characteristics determined in step 2.

An application created using this approach can run fully-protected, that is, within its own activation group. Furthermore, it can be updated easily through use of the UPDPGM or UPDSRVPGM commands. With these commands you can add or replace one or more modules without having to re-create the program object.

Single-Language ILE Application Scenario

In this scenario you compile multiple source files into modules and bind them into one program which is called by an ILE RPG/400 program. Figure 9 shows the run-time view of this application.

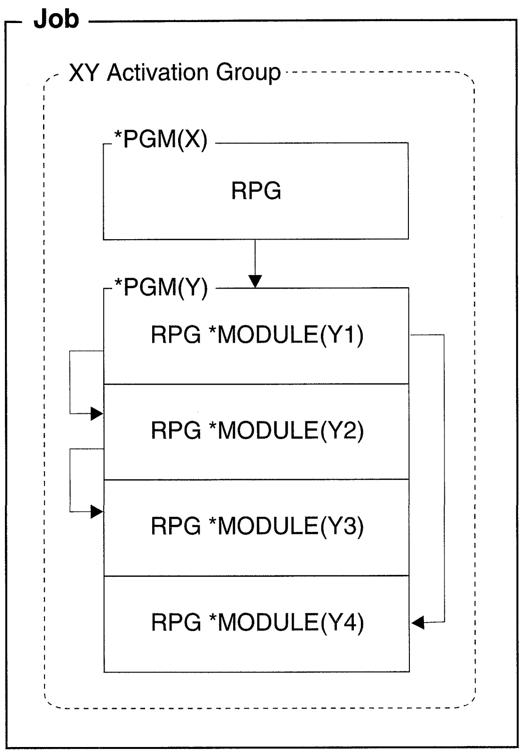


Figure 9. Single-Language Application Using CRTRPGMOD and CRTPGM

The call from program X to program Y is a dynamic call. The calls among the modules in program Y are static calls.

See “Effect of ILE” on page 22 for details on the effects of ILE on the way your application handles calls, data, files and errors.

Mixed-Language ILE Application Scenario

In this scenario, you create integrated mixed-language applications. The main module, written in one ILE language, calls procedures written in another ILE language. The main module opens files that the other modules then share. Because of the use of different languages, you may not expect consistent behavior. However, ILE ensures that this occurs.

Figure 10 shows the run-time view of an application containing a mixed-language ILE program where one module calls a non-bindable API, QUSCRTUS (Create User Space).

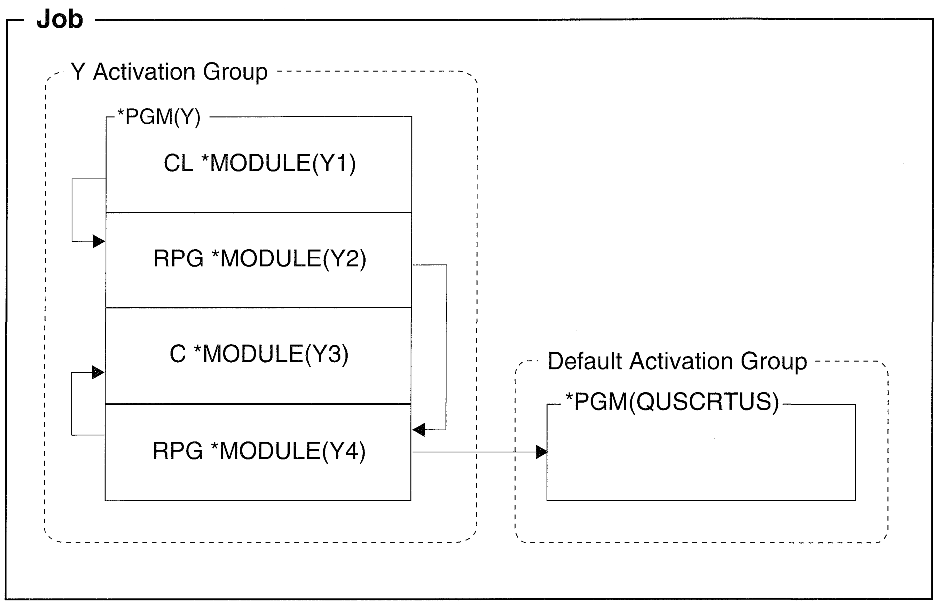


Figure 10. Mixed-Language Application

The call from program Y to the OPM API is a dynamic call. The calls among the modules in program Y are static calls.

See "Effect of ILE" on page 22 for details on the effects of ILE on the way your application handles calls, data, files and errors.

Advanced Application Scenario

In this scenario, you take full advantage of ILE function, including service programs. The use of bound calls, used for procedures within modules and service programs, provide improved performance especially if the service program runs in the same activation group as the caller.

Figure 11 shows an example in which an ILE program is bound to two service programs.

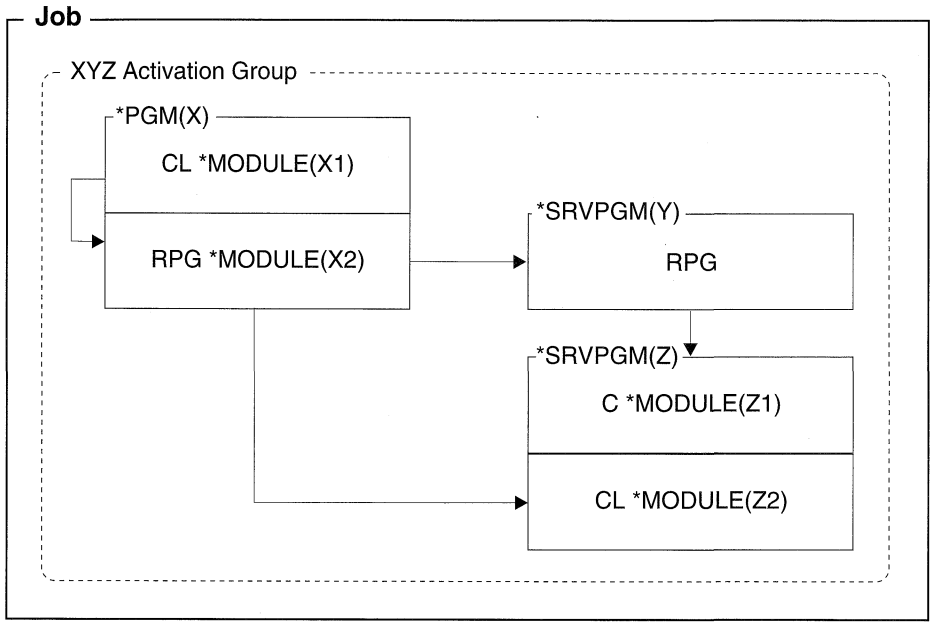


Figure 11. Advanced Application

The calls from program X to programs Y and Z are static calls.

See “Effect of ILE” on page 22 for details on the effects of ILE on the way your application handles calls, data, files and errors.

Related Information

Two-step creation process	Chapter 6, “Creating a Program with the CRTRPGMOD and CRTPGM Commands” on page 49
Activation groups	“Managing Activation Groups” on page 79
ILE static binding	Chapter 9, “Calling Programs and Procedures” on page 91; also *ILE Concepts*
Exception Handling	Chapter 11, “Handling Exceptions” on page 153; also *ILE Concepts*
Service programs	Chapter 7, “Creating a Service Program” on page 63; also *ILE Concepts*
Updating a Program	“Using the UPDPGM Command” on page 59

A Strategy to Avoid

ILE provides many alternatives for creating programs and applications. However, not all are equally good. In general, you should avoid a situation where an application consists of OPM and ILE programs which is split across the OPM default activation group and a named activation group. In other words, try to avoid the scenario shown in Figure 12.

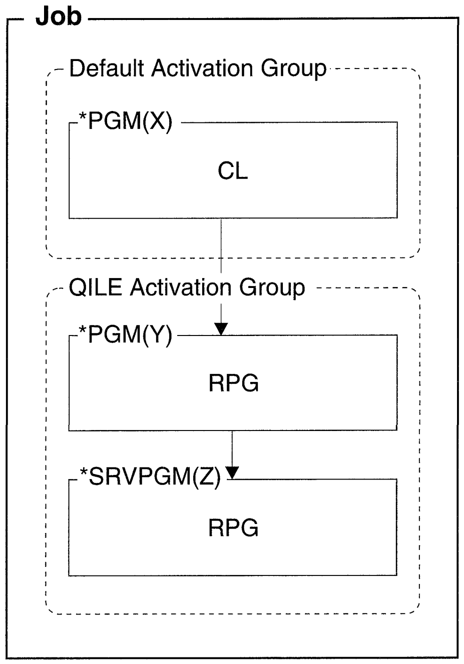


Figure 12. Scenario to Avoid. An application is split between the OPM default activation group and a named activation group.

When split across the default activation group and any named activation group, you are mixing OPM behavior with ILE behavior. For example, programs in the default activation group may be expecting the ILE programs to free their resources when the program ends. However, this will not occur until the activation group ends.

Similarly, the scope of overrides and shared ODPs will be more difficult to manage when an application is split between the default activation group and a named one. By default, the scope for the named group will be at the activation group level, but for the default activation group, it can be either call level or job level, not activation group level.

Creating and Running an ILE RPG/400 Application

This section provides you with the information needed to create and run ILE RPG/400 programs. It describes how to:

- Enter source statements
- Create modules
- Read compiler listings
- Create programs
- Create service programs
- Run programs
- Pass parameters
- Manage the run time
- Call other programs or procedures

Many ILE terms and concepts are discussed briefly in the following pages. These terms and concepts are more fully discussed in *ILE Concepts*.

Chapter 4. Entering Source Statements

This chapter provides the information you need to enter RPG source statements. It also briefly describes the tools necessary to complete this step.

To enter RPG source statements into the system, use one of the following methods:

- Interactively using SEU
- Interactively using CODE/400

Initially, you may want to enter your source statements into a file called QRPGLESRC. New members of the file QRPGLESRC automatically receive a default type of RPGLE. Furthermore, the default source file for the ILE RPG/400 commands that create modules and bind them into program objects is QRPGLESRC. IBM supplies a source file QRPGLESRC in library QGPL. It has a record length of 112 characters.

Note: You can use mixed case when entering source. However, the ILE RPG/400 compiler will convert most of the source to uppercase when it compiles it. It will not convert literals, array data or table data.

Creating a Library and Source Physical File

Source statements are entered into a member of a source physical file. Before you can enter your program, you must have a library and a source physical file.

To create a library, use the CRTLIB command. To create a source physical, use the Create Source Physical file (CRTSRCPF) command. The recommended record length of the file is 112 characters. This record length takes into account the new ILE RPG/400 structure as shown in Figure 13.

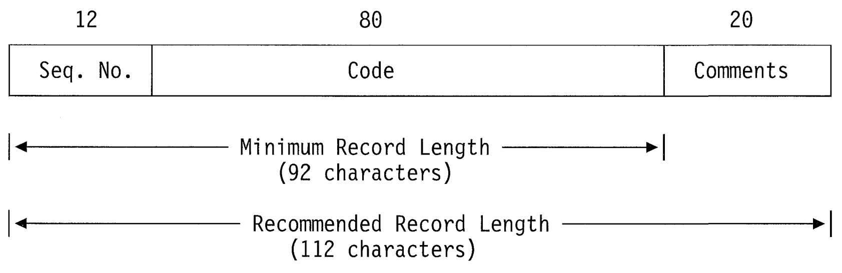


Figure 13. ILE RPG/400 Record Length Breakdown

Since the system default for a source physical file is 92 characters, you should specify a minimum record length of 112. If you specify a length less than 92 characters, the program is not likely to compile since you will be truncating source code.

Note: If you are creating a file in a new library, you must create that library prior to creating the source file.

For more information about creating libraries and source physical files, refer to *ADTS/400: Source Entry Utility* and *ADTS/400: Programming Development Manager*.

Using the Source Entry Utility (SEU)

You can use the Source Entry Utility (SEU) to enter your source statements. SEU also provides prompting for the different specification templates as well as syntax checking. To start SEU, use the STRSEU (Start Source Entry Utility) command. For other ways to start and use SEU, refer to *ADTS/400: Source Entry Utility*.

If you name your source file QRPGLESRC, SEU automatically sets the source type to RPGLE when it starts the editing session for a new member. Otherwise, you have to specify RPGLE when you create the member.

If you need prompting after you type STRSEU, press F4. The STRSEU display appears, lists the parameters, and supplies the default values. If you supply parameter values before you request prompting, the display appears with those values filled in.

In the following example you enter source statements for a program which will print employee information from a master file. This example shows you how to:

- Create a library
- Create a source physical file
- Start an SEU editing session
- Enter source statements.

1. To create a library called MYLIB, type

   ```
   CRTLIB LIB(MYLIB)
   ```

   The CRTLIB command creates a library called MYLIB.

2. To create a source physical file called QRPGLESRC type

   ```
   CRTSRCPF FILE(MYLIB/QRPGLESRC) RCDLEN(112)
   TEXT('Source physical file for all RPG programs')
   ```

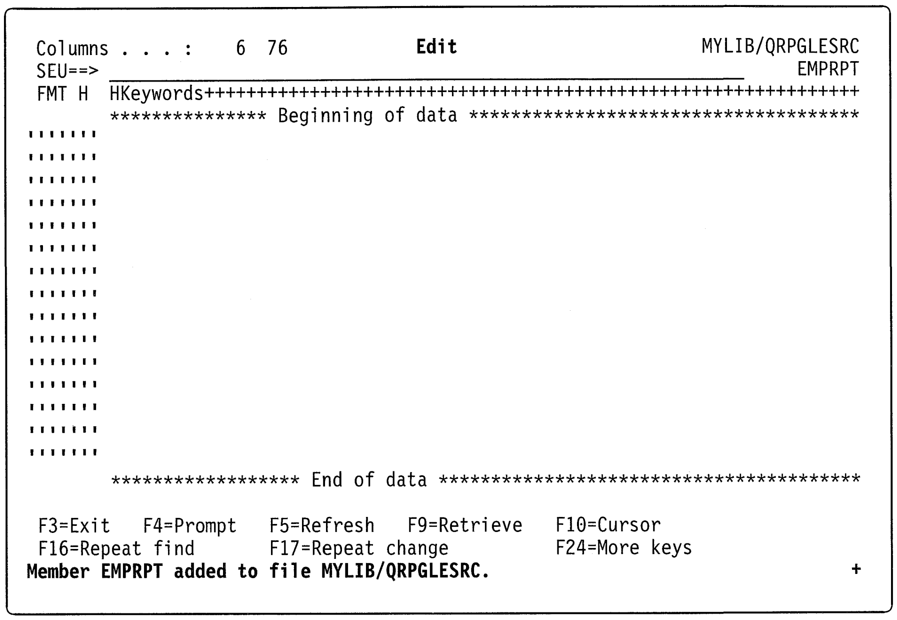
   The CRTSRCPF command creates a source physical file QRPGLESRC in library MYLIB.

3. To start an editing session and create source member EMPRPT type:

   ```
   STRSEU SRCFILE(MYLIB/QRPGLESRC)
   SRCMBR(EMPRPT)
   TYPE(RPGLE) OPTION(2)
   ```

   Entering OPTION(2) indicates that you want to start a session for a new member. The STRSEU command creates a new member EMPRPT in file QRPGLESRC in library MYLIB and starts an edit session.

   The SEU Edit display appears as shown in Figure 14 on page 33. Note that the screen is automatically shifted so that position 6 is (for specification type) is at the left edge.



```
 Columns . . . :    6  76            Edit                   MYLIB/QRPGLESRC
 SEU==> ___________________________________________________________  EMPRPT
 FMT H  HKeywords++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
        *************** Beginning of data ************************************
''''''' 
''''''' 
''''''' 
''''''' 
''''''' 
''''''' 
''''''' 
''''''' 
''''''' 
''''''' 
''''''' 
''''''' 
''''''' 
''''''' 
''''''' 
        ****************** End of data ***************************************

 F3=Exit   F4=Prompt   F5=Refresh   F9=Retrieve   F10=Cursor
 F16=Repeat find       F17=Repeat change          F24=More keys
Member EMPRPT added to file MYLIB/QRPGLESRC.                                  +
```

Figure 14. Edit Display for a New Member

4. Type the following source in your SEU Edit display, using the following SEU prefix commands to provide prompting:
   - IPF — for File Description specifications
   - IPI — for Input specifications
   - IPC — for Calculation specifications
   - IPCX — for Calculation specifications with extended Factor 2
   - IPO — for Output specifications
   - IPP — for Output specifications continuation

```
F*****************************************************************
F* MODULE NAME:    EMPRPT                                        *
F* RELATED FILES:  EMPMST   (PHYSICAL FILE)                      *
F*                 PRINT    (PRINTER FILE)                       *
F* DESCRIPTION:    THIS PROGRAM PRINTS EMPLOYEE INFORMATION      *
F*                 FROM THE FILE EMPMST.                         *
F*****************************************************************
FQSYSPRT   O    F   80         PRINTER
FEMPMST    IP   E            K DISK

D TYPE            S              8
```

Figure 15 (Part 1 of 2). Source for EMPRPT member

```
IEMPREC        01

C                   IF        ETYPE = 'M'
C                   EVAL      TYPE = 'MANAGER'
C                   ELSE
C                   EVAL      TYPE = 'REGULAR'
C                   ENDIF

OQSYSPRT   H    1P                  2  6
O                                        50 'EMPLOYEE INFORMATION'
O          H    1P
O                                        12 'NAME'
O                                        34 'SERIAL #'
O                                        45 'DEPT'
O                                        56 'TYPE'
O          D    01
O                       ENAME            20
O                       ENUM             32
O                       EDEPT            45
O                       TYPE             60
```

Figure 15 (Part 2 of 2). Source for EMPRPT member

5. Press F3 (Exit) to go to the Exit display. Type Y (Yes) to save EMPRPT.

   The member EMPRPT is saved.

Figure 16 shows the DDS which is referenced by the EMPRPT source.

```
A*****************************************************************
A* DESCRIPTION:  This is the DDS for the physical file EMPMST.   *
A*               It contains one record format called EMPREC.    *
A*               This file contains one record for each employee *
A*               of the company.                                 *
A*****************************************************************
A*
A          R EMPREC
A            ENUM          5  0       TEXT('EMPLOYEE NUMBER')
A            ENAME        20          TEXT('EMPLOYEE NAME')
A            ETYPE         1          TEXT('EMPLOYEE TYPE')
A            EDEPT         3  0       TEXT('EMPLOYEE DEPARTMENT')
A            ENHRS         3  1       TEXT('EMPLOYEE NORMAL WEEK HOURS')
A          K ENUM
```

Figure 16. DDS for EMPRPT

To create a program from this source using the CRTBNDRPG command, see “Creating an OPM-Compatible Program Object” on page 39.

Using DB2/400 SQL Statements

The DB2* for OS/400 database can be accessed from an ILE RPG/400 program by embedding SQL statements into your program source. Use the following rules to enter your SQL statements:

- Enter your SQL statements on the Calculation specification
- Start your SQL statements using the delimiter /EXEC SQL in positions 7-15 (with the / in position 7)

- You can start entering your SQL statements on the same line as the starting delimiter
- Use the continuation line delimiter (a + in position 7) to continue your statements on any subsequent lines
- Use the ending delimiter /END-EXEC in positions 7-15 (with the slash in position 7) to signal the end of your SQL statements.

Note: SQL statements cannot go past position 80 in your program.

Figure 17 shows an example of embedded SQL statements.

```
...+....1....+....2....+....3....+....4....+....5....+....6....+....7..
     C
     C            (ILE RPG/400 calculation operations)
     C
     C/EXEC SQL    (the starting delimiter)
     C+
     C+       (continuation lines containing SQL statements)
     C+
     .
     .
     .
     C/END-EXEC    (the ending delimiter)
     C
     C            (ILE RPG/400 calculation operations)
     C
```

Figure 17. SQL Statements in an ILE RPG/400 Program

You must enter a separate command to process the SQL statements. Refer to the *DB2/400 SQL Programming* and the *DB2/400 SQL Reference* for more information.

Refer to the *ADTS/400: Source Entry Utility* for information about how SEU handles SQL statement syntax checking.

Including Graphic Data in Programs

Full support is now provided for graphic data type (G) fields in the RPG IV language. See the *ILE RPG/400 Reference* for details on ILE RPG/400 graphic support.

Chapter 5. Creating a Program with the CRTBNDRPG Command

This chapter shows how to create an ILE program using RPG IV source with the Create Bound RPG Program (CRTBNDRPG) command. With this command you can create one of two types of ILE programs:

- OPM-compatible program with no static binding
- Single-module ILE program with static binding

Whether you obtain a program of the first type or the second type depends on whether the DFTACTGRP parameter of CRTBNDRPG is set to *YES or *NO respectively.

Creating a program of the first type produces a program which behaves like an OPM program in the areas of open scoping, override scoping, and 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, you cannot use the CALLB operation in your source nor can you bind to other modules during program creation.

Creating a program of the second type produces a program with ILE characteristics such as static binding. You can specify at program creation time the activation group the program is to run in, and any modules for static binding. In addition, you can use CALLB in your source.

Using the CRTBNDRPG Command

The Create Bound RPG (CRTBNDRPG) command creates a program object from RPG IV source in one step. It also allows you to bind in other modules or service programs using a binding directory.

The command starts the ILE RPG/400 compiler and creates a temporary module object in the library QTEMP. It then binds it into a program object of type *PGM. Once the program object is created, the temporary module used to create the program is deleted.

The CRTBNDRPG command is useful when you want to create a program object from standalone source code (code that does not require modules to be bound together), because it combines the steps of creating and binding. Furthermore, it allows you to create an OPM-compatible program.

Note: If you want to keep the module object in order to bind it with other modules into a program object, you must create the module using the CRTRPGMOD command. For more information see Chapter 6, “Creating a Program with the CRTRPGMOD and CRTPGM Commands” on page 49.

You can use the CRTBNDRPG command interactively, in batch, or from a Command Language (CL) program. If you are using the command interactively and require prompting, type CRTBNDRPG and press F4 (Prompt). If you need help, type CRTBNDRPG and press F1 (Help).

Table 2 summarizes the parameters of the CRTBNDRPG command and shows their default values.

Table 2. CRTBNDRPG Parameters and Their Default Values Grouped by Function

Program Identification	
PGM(*CURLIB/*CTLSPEC)	Determines created program name and library
SRCFILE(*LIBL/QRPGLESRC)	Identifies source file and library
SRCMBR(*PGM)	Identifies file member containing source specifications
TEXT(*SRCMBRTXT)	Provides brief description of program.
Program Creation	
GENLVL(10)	Conditions program creation to error severity (0-20).
OPTION(*GEN)	*GEN/*NOGEN, determines if program is created.
DBGVIEW(*STMT)	Specify type of debug view, if any, to be included in program.
OPTIMIZE(*NONE)	Determine level of optimization, if any.
REPLACE(*YES)	Determine if program should replace existing program.
BNDDIR(*NONE)	Specify the binding directory to be used for symbol resolution.
USRPRF(*USER)	Specify the user profile that will run program.
AUT(*LIBCRTAUT)	Specify type of authority for created program
TGTRLS(*CURRENT)	Specify the release level the object is to be run on.
Compiler Listing	
OUTPUT(*PRINT)	Determine if there is a compiler listing.
INDENT(*NONE)	Determine if indentation should show in listing, and identify character for marking it.
OPTION(*XREF *NOSECLVL *SHOWCPY *EXPDDS *EXT)	Specify contents of compiler listing.
Data Conversion Options	
CVTOPT(*NONE)	Specify how various data types from externally-described files are handled.
ALWNULL(*NO)	Determine if the program will accept values from null-capable fields.
FIXNBR(*NONE)	Determine if invalid zoned decimal data is to be fixed upon conversion to packed data.
Run-Time Considerations	
DFTACTGRP(*YES)	Identify whether this program always runs in the OPM default activation group.
ACTGRP(QILE)	Identify the activation group the program should run in.
SRTSEQ(*HEX)	Specify the sort sequence table to be used.
LANGID(*JOBRUN)	Used with SRTSEQ to specify the language identifier for sort sequence.
TRUNCNBR(*YES)	Specify action to take when numeric overflow occurs.

See Appendix C, “The Create Commands” on page 333 for the syntax diagram and parameter descriptions of CRTBNDRPG.

Creating an OPM-Compatible Program Object

In this example you use the CRTBNDRPG command to create an OPM-compatible program object from the source for EMPRPT, shown in Figure 15 on page 33.

1. To create the object, type:

   ```
   CRTBNDRPG PGM(MYLIB/EMPRPT)
             SRCFILE(MYLIB/QRPGLESRC)
             TEXT('ILE RPG/400 program')  DFTACTGRP(*YES)
   ```

   The CRTBNDRPG command creates the program EMPRPT in MYLIB which will run in the default activation group. By default, a compiler listing is produced.

   Note: The setting of DFTACTGRP(*YES) is what provides the OPM compatibility. This setting also prevents you from entering a value for the ACTGRP and BNDDIR parameters. Furthermore, if the source contained any CALLB operations an error would be issued and the compilation would end.

2. Type one of the following CL commands to see the listing that is created:

   - DSPJOB and then select option 4 (*Display spooled files*)
   - WRKJOB
   - WRKOUTQ *queue-name*
   - WRKSPLF

Creating a Program for Source Debugging

In this example you create the program EMPRPT so that you can debug it using the source debugger. The DBGVIEW parameter on either CRTBNDRPG or CRTRPGMOD determines what type of debug data is created during compilation. The parameter provides six options which allow you to select which view(s) you want:

- *STMT — allows you to display variables and set breakpoints at statement locations using a compiler listing. No source is displayed with this view.
- *SOURCE — creates a view identical to your input source.
- *COPY — creates a source view and a view containing the source of any /COPY members.
- *LIST — creates a view similar to the compiler listing.
- *ALL — creates all of the above views.
- *NONE — no debug data is created.

The source for EMPRPT is shown in Figure 15 on page 33.

1. To create the object type:

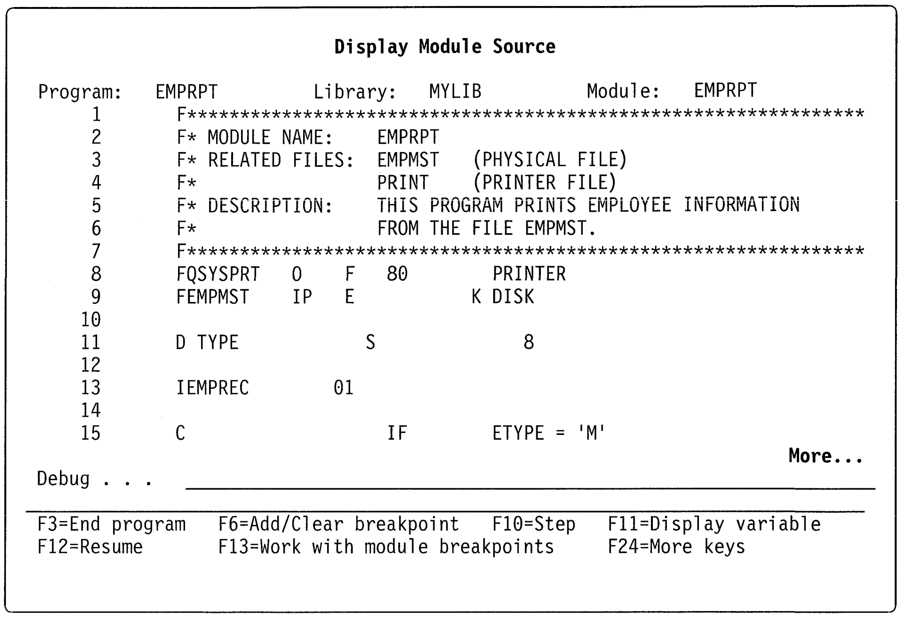
   ```
   CRTBNDRPG PGM(MYLIB/EMPRPT) DBGVIEW(*SOURCE)
   ```

   The program will be created in the library MYLIB with the same name as the source member on which it is based, namely, EMPRPT. Note that by default, it will be created with DFTACTGRP(*YES). This program object can be debugged using a source view.

2. To debug the program type:

   ```
   STRDBG EMPRPT
   ```

   Figure 18 on page 40 shows the screen which appears after entering the above command.



```
                             Display Module Source

 Program:   EMPRPT         Library:   MYLIB         Module:   EMPRPT
         F***************************************************************
         F* MODULE NAME:     EMPRPT
         F* RELATED FILES:   EMPMST   (PHYSICAL FILE)
         F*                  PRINT    (PRINTER FILE)
         F* DESCRIPTION:     THIS PROGRAM PRINTS EMPLOYEE INFORMATION
         F*                  FROM THE FILE EMPMST.
         F***************************************************************
         FQSYSPRT   O    F   80        PRINTER
         FEMPMST    IP   E            K DISK

         D TYPE            S             8

         IEMPREC       01

         C                   IF        ETYPE = 'M'
                                                                  More...
 Debug . . .   ___________________________________________________________

 F3=End program   F6=Add/Clear breakpoint   F10=Step   F11=Display variable
 F12=Resume       F13=Work with module breakpoints     F24=More keys
```

Figure 18. Display Module Source display for EMPRPT

From this screen (the Display Module Source display) you can enter debug commands to display or change field values and set breakpoints to control program flow while debugging.

For more information on debugging see Chapter 10, “Debugging Programs” on page 113.

Creating a Program with Static Binding

In this example you create a program COMPUTE using CRTBNDRPG to which you bind a service program at program creation time.

Assume that you want to bind the program COMPUTE to services which you have purchased to perform advanced mathematical computations. The binding directory to which you must bind your source is called MATH. This directory contains the name of a service program which contains the various procedures that make up the services.

To create the object, type:

```
CRTBNDRPG PGM(MYLIB/COMPUTE)
          DFTACTGRP(*NO) ACTGRP(GRP1) BNDDIR(MATH)
```

The source will be bound to the service program specified in the binding directory MATH at program creation time. This means that calls to the procedures in the service program will take less time than if they were dynamic calls.

When the program is called, it will run in the named activation group GRP1. The default value ACTGRP parameter on CRTBNDRPG is QILE. However, it is recommended that you run your application as a unique group to ensure that the associated resources are fully protected.

Note: DFTACTGRP must be set to *NO in order for you to enter a value for the ACTGRP and BNDDIR parameters.

For more information on service programs, see Chapter 7, "Creating a Service Program" on page 63.

Using a Compiler Listing

This section discusses how to obtain a listing and how to use it to help you:

- fix compilation errors
- fix run-time errors
- provide documentation for maintenance purposes.

See Appendix D, "Compiler Listings" on page 359 for more information on the different parts of the listing and for a complete sample listing.

Obtaining a Compiler Listing

To obtain a compiler listing specify OUTPUT(*PRINT) on either the CRTBNDRPG command or the CRTRPGMOD command. (This is their default setting.) The specification OUTPUT(*NONE) will suppress a listing.

Specifying OUTPUT(*PRINT) results in a compiler listing which consists *minimally* of the following sections:

- Prologue (command option summary)
- Source Listing, which includes:
  - In-Line diagnostic messages
  - Match-field table (if using the RPG cycle with match fields)
- Additional diagnostic messages
- Field Positions in Output Buffer
- /COPY Member Table
- Compile Time Data which includes:
  - Alternate Collating Sequence records and table or NLSS information and table
  - File translation records
  - Array records
  - Table records
- Message summary
- Final summary
- Code generation report (appears only if there are errors)
- Binding report (applies only to CRTBNDRPG; appears only if there are errors)

The following additional information is included in a compiler listing if the appropriate value is specified on the OPTION parameter of either create command:

***EXPDDS**	Specifications of externally-described files (appear in source section of listing)
***SHOWCPY**	Source records of /COPY members (appear in source section of listing)
***EXPDDS**	Key field information (separate section)
***XREF**	List of Cross references (separate section)
***EXT**	List of External references (separate section)
***SECLVL**	Second-level message text (appear in message summary section)

Note: Except for *SECLVL, all of the above values reflect the default settings on the OPTION parameter for both create commands. You do not need to change the OPTION parameter unless you do not want certain listing sections or unless you want second level text to be included.

Customizing a Compiler Listing

You can customize a compiler listing by either customizing the page heading or by customizing the spacing.

Customizing a Page Heading: The page heading information includes the product information line and the title supplied by a /TITLE directive. The product information line includes the ILE RPG/400 compiler and library copyright notice, the member, and library of the source program, the date and time when the module was created, and the page number of the listing.

You can specify heading information on the compiler listing through the use of the /TITLE compiler directive. This directive allows you to specify text which will appear at the top of each page of the compiler listing. This information will precede the usual page heading information. If the directive is the first record in the source member, then this information will also appear in the prologue section.

You can also change the date separator, date format, and time separator used in the page heading and other information boxes throughout the listing. Normally, the compiler determines these by looking at the job attributes. To change any of these, use the Change Job (CHGJOB) command. After entering this command you can:

- Select one of the following date separators: *SYSVAL, *BLANK, slash (/), hyphen (-) period (.) or comma (,)
- Select one of the following date formats: *SYSVAL, *YMD, *MDY, *DMY, or *JUL
- Select one of the following time separators: *SYSVAL, *BLANK, colon (:), comma (,) or period (.)

Anywhere a date or time field appears in the listing, these values are used.

Customizing the Spacing: Each section of a listing usually starts on a new page; Each page of the listing starts with product information, unless the source member contains a /TITLE directive. If it does, the product information appears on the second line and the title appears on the first line.

You can control the spacing and pagination of the compiler listing through the use of the /EJECT and /SPACE compiler directives. The /EJECT directive forces a page break. The /SPACE directive controls line spacing within the listing. For more information on these directives refer to the *ILE RPG/400 Reference.*

Indenting Structured Operations in the Compiler Listing

If your source specifications contain structured operations (such as DO-END or IF-ELSE-END), you may want to have these indented in the source listing. The INDENT parameter lets you specify whether to show indentation, and specify the character to mark the indentation. If you do not want indentation, specify INDENT(*NONE); this is the default. If you do want indentation, then specify up to two characters to mark the indentation.

For example, to specify that you want structured operations to be indented and marked with a vertical bar (|) followed by a space, you specify INDENT('| ').

If you request indentation, then some of the information which normally appears in the source listing is removed, so as to allow for the indentation. The following columns will **not** appear in the listing:

- Do Num
- Last Update
- PAGE/LINE

If you specify indentation and you also specify a listing debug view, the indentation will not appear in the debug view.

Figure 19 shows part of source listing which was produced with indentation. The indentation mark is '| '.

```
Line   <---------------------- Source Specifications -------------------------------------------><---- Comments ----> Src  Seq
Number ....1....+....2....+<-------- 26 - 35 --------->....4....+....5....+....6....+....7....+....8....+....9....+...10 Id   Number
    33 C********************************************************************                                    000000       002000
    34 C*    MAINLINE                                                      *                                    000000       002100
    35 C********************************************************************                                    000000       002200
    36 C                   WRITE                       FOOT1                                                                 002300
    37 C                   WRITE                       HEAD                                                                  002400
    38 C                   EXFMT                       PROMPT                                                                002500
    39 C*                                                                                                       000000       002600
    40 C                   DOW                         NOT *IN03                                                             002700
    41 C     CSTKEY        | SETLL                     CMLREC2                            ----20                             002800
    42 C                   | IF                        *IN02                                                                 002900
    43 C                   | | MOVE                    '1'           *IN61                                                   003000
    44 C                   | ELSE                                                                                            003100
    45 C                   | | EXSR                    SFLPRC                                                                003200
    46 C                   | END                                                                                             003300
    47 C                   | IF                        NOT *IN03                                                             003400
    48 C                   | | IF                      *IN04                                                                 003500
    49 C                   | | | IF                    *IN61                                                                 003600
    50 C                   | | | | WRITE               FOOT1                                                                 003700
    51 C                   | | | | WRITE               HEAD                                                                  003800
    52 C                   | | | ENDIF                                                                                       003900
    53 C                   | | | EXFMT                 PROMPT                                                                004000
    54 C                   | | ENDIF                                                                                         004100
    55 C                   | ENDIF                                                                                           004200
    56 C                   ENDDO                                                                                             004300
    57 C*                                                                                                       000000       004400
    58 C                   SETON                                                      LR----                             004500
```

Figure 19. Sample Source Part of the Listing with Indentation

Correcting Compilation Errors

The main sections of a compiler listing that are useful for fixing compilation errors are:

- The source section
- The Additional Messages section
- The /COPY table section
- The various summary sections.

In-line diagnostic messages, which are found in the source section, point to errors which the compiler can flag immediately. Other errors are flagged after additional information is received during compilation. The messages which flag these errors are in the source section and Additional Messages section.

To aid you in correcting any compilation errors, you may want to include the second-level message text in the listing — especially if you are new to RPG. To do this, specify OPTION(*SECLVL) on either create command. This will add second-level text to the messages listed in the message summary.

Finally, keep in mind that a compiler listing is a record of your program. Therefore, if there are any errors, you can use the listing to check that the source is coded the way you intended it to be. Parts of the listing, besides the source statements, which you may want to check include:

- Match field table

  If you are using the RPG cycle with match fields, then you can use this to confirm that the compiler is recognizing them as intended.

- Offset-defined output fields

  Lists the start and end positions along with the literal text or field names. Use this to confirm that the compiler is recognizing the input as intended.

- Compile-time data

  ALTSEQ and FTRANS records and tables are listed. NLSS information and tables are listed. Tables and arrays are explicitly identified. Use this to confirm that the compiler is recognizing them as intended.

Using In-Line Diagnostic Messages

There are two types of in-line diagnostic messages: finger and non-finger. Finger messages point out exactly where the error occurred. Figure 20 shows an example of finger in-line diagnostic messages.

```
Line   <---------------------- Source Specifications ---------------------------------><---- Comments ----> Do  Page Change Src  Seq
Number ....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8....+....9....+...10 Num Line Date   Id   Number

   63  C                    SETOFF                                          _12___                                         003100
======>                                                                     aabb
======>                                                                     cccc

*RNF5051 30 a Resulting-Indicator entry is not valid; defaults to blanks.
*RNF5051 30 b Resulting-Indicator entry is not valid; defaults to blanks.
*RNF5053 30 c Resulting-Indicators entry is blank for specified operation.
```

Figure 20. Sample Finger In-Line Diagnostic Messages

In this example, an indicator has been incorrectly placed in positions 72 - 73 instead of 71 - 72 or 73 - 74. The three fingers 'aa', 'bb', and 'cc' identify the parts of the line where there are errors. The actual columns are highlighted with variables which are further explained by the messages. In this case, message RNF5051 indicates that the fields marked by 'aa' and 'bb' do not contain a valid indicator. Since there is no valid indicator the compiler assumes that the fields are blank. However, since the SETOFF operation requires an indicator, another error arises, as pointed out by the field 'cccc' and message RNF5053.

Errors are listed in the order in which they are found. As a general rule, you should focus on correcting the first few errors found, rather than the later ones.

Non-finger in-line diagnostic messages also indicate errors. However, due to the nature of the error, they cannot point to a specific location in the source; they can only state a problem. Figure 21 on page 45 shows an example of the non-finger in-line diagnostic messages.

```
Line   <---------------------- Source Specifications ----------------------------><---- Comments ----> Do  Page  Change  Src  Seq
Number ....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8....+....9....+...10 Num Line  Date    Id   Number

    6  FINTERN    IS   F   72        DISK                                                              12345              000600
    7  FQSYSPRT   O    F  256        DISK                                                                                 000700
    8  FF7145A    CT   F   28        DISK                                                                                 000800
    9  FF7145B    CT   F   28        DISK                                                                                 000900
   10  F*                                                                                              12345 100792       001000

*RNF2025 30        6 INTERN defaults to primary file.


   11  DARRA             S             28    FROMFILE(F7145A) TOFILE(F7145A)                                              001100
```

Figure 21. Sample Non-Finger In-Line Diagnostic Messages

In this example, there are several file specifications, one of which defines a secondary file, but there is no primary file defined. Because a primary file is required, a message (RNF2025) is issued. The compiler cannot point to a particular line which should define a primary file. It can only assume, once the next specification section begins, that there is no such line.

Using Additional-Diagnostic Messages

The Additional Diagnostic Messages section identifies errors which arise when one or more lines of code are viewed collectively. These messages are not placed within the code where the problem is; in general, the compiler does not know at the time of writing that portion of the source that a problem exists. However, when possible, the message line includes the listing statement number of a source line which is related to the message.

Browsing a Compiler Listing Using SEU

The SEU Split/Browse session (F15) allows you to browse a compiler listing in the output queue. You can review the results of a previous compilation while making the required changes to your source code.

While browsing the compiler listing, you can scan for errors and correct those source statements that have errors. To scan for errors, type `F *ERR` on the SEU command line of the browse session. The line with the first (or next) error is highlighted and the first-level text of the same message appears at the bottom of the screen. You can see the second-level text by placing your cursor on the message at the bottom and then pressing F1 (Help).

When possible, the error messages in the listing identify the SEU sequence number of the line in error. The line number is found just before the message text.

For complete information on browsing a compiler listing, see *ADTS/400: Source Entry Utility*.

Correcting Run-time Errors

The source section of the listing is also useful for correcting run-time errors. Many run-time error messages identify a statement number where the error in question occurred. The line number on the *left* side of the compiler listing corresponds to the statement number in the run-time error message. The source ID number and the SEU sequence number on the *right* side of the compiler listing identifies the source member and record. You can use the two together, especially if you are editing the source using SEU, to determine which line needs to be examined.

Coordinating Listing Options with Debug View Options

Correcting run-time errors often involves debugging a program. The following considerations may help you when you go to debug your program:

- If you use the source debugger to debug your program you have a choice of debug views: *STMT, *SOURCE, *LIST, *COPY, *ALL.
- If you plan to use a compiler listing as an aid while debugging, then you can obtain one by specifying OUTPUT(*PRINT). A listing is important if you intend to debug using a statement (*STMT) view since the line numbers for setting breakpoints are those identified in the listing.
- If you know that you will have considerable debugging to do, you may want to compile the source with DBGVIEW(*ALL), OUTPUT(*PRINT) and OPTION(*SHOWCPY). This will allow you to use either a source or listing view, and it will include /COPY members.
- If you specify DBGVIEW(*LIST), the information available to you while debugging depends on what you specified for the OPTION parameter. The view will include /COPY members and externally-described files only if you specify OPTION(*SHOWCPY *EXPDDS).

Using a Compiler Listing for Maintenance

A compiler listing of an error-free program can be used as documentation when:

- teaching the program to a new programmer.
- updating the program at a later date.

In either case it is advisable to have a full listing, namely, one produced with OUTPUT(*PRINT) and with OPTION(*XREF *SHOWCPY *EXPDDS *EXT). Note that this is the default setting for each of these parameters for both create commands.

Of particular value for program maintenance is the Prologue section of the listing. This section tells you:

who compiled the module/program
what source was used to produce the module/program
what options were used when compiling the module/program

You may need to know about the command options (for example, the debug view selected, or the binding directory used) when you make later changes to the program.

The following specifications for the OPTION parameter provides additional information as indicated:

- *SHOWCPY and *EXPDDS provide a complete description of the program, including all specifications form /COPY members, and generated specifications from externally-described files.
- *XREF allows you to check the use of files, fields and indicators within the module/program.
- *EXT allows you to see which procedures and fields are imported or exported by the module/program. It also identifies the actual files which were used for generating the descriptions for externally-described files and data structures.

Accessing the RETURNCODE Data Area

Both the CRTBNDRPG and CRTRPGMOD (see "Using the CRTRPGMOD Command" on page 50) commands create and update a data area with the status of the last compilation. This data area is named RETURNCODE, is 400 characters long, and is placed into library QTEMP.

To access the RETURNCODE data area, specify RETURNCODE in factor 2 of a *DTAARA DEFINE statement.

The data area RETURNCODE has the following format:

Byte	**Content and Meaning**
1	For CRTRPGMOD, character '1' means a module was created in the specified library. For CRTBNDRPG, character '1' means a module with the same name as the program name was created in QTEMP.
2	Character '1' means the compilation failed because of compiler errors.
3	Character '1' means the compilation failed because of source errors.
4	Not set. Always '0'.
5	Character '1' means the translator was not called because either OPTION(*NOGEN) was specified on the CRTRPGMOD or CRTBNDRPG command; or the compilation failed before the translator was called.
6-10	Number of source statements
11-12	Severity level from command
13-14	Highest severity of diagnostic messages
15-20	Number of errors found in the module (CRTRPGMOD) or program (CRTBNDRPG).
21-26	Compile date
27-32	Compile time
33-100	Not set. Always blank
101-110	Module (CRTRPGMOD) name or program (CRTBNDRPG) name.
111-120	Module (CRTRPGMOD) library name or program (CRTBNDRPG) library name.
121-130	Source file name
131-140	Source file library name
141-150	Source file member name
151-160	Compiler listing file name
161-170	Compiler listing library name
171-180	Compiler listing member name
181-329	Not set. Always blank

330-334	Total elapsed compile time to the nearest 10th of a second
335	Not set. Always blank
336-340	Elapsed compile time to the nearest 10th of a second
341-345	Elapsed translator time to the nearest 10th of a second
346-379	Not set. Always blank
380-384	Total compile CPU time to the nearest 10th of a second
385	Not set. Always blank
386-390	CPU time used by compiler to the nearest 10th of a second
391-395	CPU time used by the translator to the nearest 10th of a second
396-400	Not set. Always blank

Chapter 6. Creating a Program with the CRTRPGMOD and CRTPGM Commands

The two-step process of program creation consists of compiling source into modules using CRTRPGMOD and then binding one or more module objects into a program using CRTPGM. With this process you can create permanent modules. This in turn allows you to modularize an application without recompiling the whole application. It also allows you to reuse the same module in different applications.

This chapter shows how to:

- Create a module object from RPG IV source
- Bind modules into a program using CRTPGM
- Read a binder listing
- Change a module or program

Creating a Module Object

A **module** is a nonrunnable object (type *MODULE) that is the output of an ILE compiler. It is the basic building block of an ILE program.

A module consists of one or more procedures. The number of procedures allowed is language dependent. An ILE RPG/400 module consists of *one* procedure which has its own LR semantics, cycle, file control blocks, and static storage.

Module creation consists of compiling a source member, and, if that is successful, creating a *MODULE object. The *MODULE object includes a list of imports and exports referenced within the module. It also includes debug data if you request this at compile time.

A module cannot be run by itself. You must bind one or more modules together to create a program object (type *PGM) which can then be run. You can also bind one or more modules together to create a service program object (type *SRVPGM). You then access the procedures within the bound modules through static procedure calls.

This ability to combine modules allows you to:

- Reuse pieces of code generally resulting in smaller programs. Smaller programs give you better performance and easier debugging capabilities.
- Maintain shared code with little chance of introducing errors to other parts of the overall program.
- Manage large programs more effectively. Modules allow you to divide your old program into parts which can be managed separately. If the program needs to be enhanced, you only need to recompile those modules which have been changed.
- Create mixed-language programs where you bind together modules written in the best language for the task required.

For more information about the concept of modules, refer to the *ILE Concepts*.

Using the CRTRPGMOD Command

You create a module using the Create RPG Module (CRTRPGMOD) command. You can use the command interactively, as part of a batch input stream, or from a Command Language (CL) program.

If you are using the command interactively and need prompting, type CRTRPGMOD and press F4 (Prompt). If you need help, type CRTRPGMOD and press F1 (Help).

Table 3 lists the parameters of the CRTRPGMOD command and their system-supplied defaults. The syntax diagram of the command and a description of the parameters are found in Appendix C, "The Create Commands" on page 333.

Table 3. CRTRPGMOD Parameters and Their Default Values Grouped by Function

Module Identification	
MODULE(*CURLIB/*CTLSPEC)	Determines created module name and library
SRCFILE(*LIBL/QRPGLESRC)	Identifies source file and library
SRCMBR(*MODULE)	Identifies file member containing source specifications
TEXT(*SRCMBRTXT)	Provides brief description of module.
Module Creation	
GENLVL(10)	Conditions module creation to error severity (0-20).
OPTION(*GEN)	*GEN/*NOGEN, determines if module is created.
DBGVIEW(*STMT)	Specify type of debug view, if any, to be included in module.
OPTIMIZE(*NONE)	Determine level of optimization, if any.
REPLACE(*YES)	Determine if module should replace existing module.
AUT(*LIBCRTAUT)	Specify type of authority for created module
TGTRLS(*CURRENT)	Specify the release level the object is to be run on.
Compiler Listing	
OUTPUT(*PRINT)	Determine if there is a compiler listing.
INDENT(*NONE)	Determine if indentation should show in listing, and identify character for marking it.
OPTION(*XREF *NOSECLVL *SHOWCPY *EXPDDS *EXT)	Specify contents of compiler listing.
Data Conversion Options	
CVTOPT(*NONE)	Specify how various data types from externally-described files are handled.
ALWNULL(*NO)	Determine if the module will accept values from null-capable fields.
FIXNBR(*NONE)	Determine if invalid zoned decimal data is to be fixed upon conversion to packed data.
Run-Time Considerations	
SRTSEQ(*HEX)	Specify the sort sequence table to be used.
LANGID(*JOBRUN)	Used with SRTSEQ to specify the language identifier for sort sequence.
TRUNCNBR(*YES)	Specify action to take when numeric overflow occurs.

When requested, the CRTRPGMOD command creates a compiler listing which is for the most part identical to the listing that is produced by the CRTBNDRPG command. (The listing created by CRTRPGMOD will never have a code generation or binding section.)

For information on using the compiler listing, see "Using a Compiler Listing" on page 41. A sample compiler listing is provided in Appendix D, "Compiler Listings" on page 359.

Creating a Module Using CRTRPGMOD Defaults

In this example you create an ILE RPG/400 module object INCALC using the CRTRPGMOD command and its default settings. The source for INCALC shown in Figure 22.

1. To create a module object, type:

   ```
   CRTRPGMOD MODULE(MYLIB/INCALC)  SRCFILE(MYLIB/QRPGLESRC)
   ```

   The module will be created in the library MYLIB with the name specified in the command, INCALC. The source for the module is the source member INCALC in file QRPGLESRC in the library MYLIB.

   This module object can be debugged using a statement view and a compiler listing for the module is produced.

2. Type one of the following CL commands to see the compiler listing.

   - DSPJOB and then select option 4 (*Display spooled files*)
   - WRKJOB
   - WRKOUTQ *queue-name*
   - WRKSPLF

```
 *================================================================*
 * MODULE NAME:    INCALC                                         *
 * RELATED FILES:  N/A                                            *
 * RELATED SOURCE: TRNSRPT                                        *
 * DESCRIPTION:    This source calculates the income for the      *
 *                 transaction using the data in the fields in    *
 *                 the parameter list.                            *
 *                 It returns to the TRNSRPT after all the        *
 *                 calculations are done.                         *
 *================================================================*
C     *ENTRY        PLIST
C                   PARM                    Prod            10
C                   PARM                    Qty              5 0
C                   PARM                    Disc             3 2
C                   PARM                    Inc             11 1
```

Figure 22 (Part 1 of 2). Source for INCALC member

```
C                   SELECT
C                   WHEN      Prod = 'Model 1'
C                   EVAL      Inc = 1500 * Qty * Disc
C                   WHEN      Prod = 'Model 2'
C                   EVAL      Inc = 3500 * Qty * Disc
C                   WHEN      Prod = 'Model 8'
C                   EVAL      Inc = 32000 * Qty * Disc
C                   WHEN      Prod = 'Model 12'
C                   EVAL      Inc = 28000 * Qty * Disc
C                   OTHER
C                   EVAL      Inc = 0
C                   ENDSL

 *  Return to the  caller TRNSRPT.                            *
C                   RETURN
```

Figure 22 (Part 2 of 2). Source for INCALC member

Creating a Module for Source Debugging

In this example, you create an ILE RPG/400 module object which you can debug using the source debugger. The source for this module is shown in Figure 23.

To create a module object, type:

```
CRTRPGMOD MODULE(MYLIB/TRNSRPT)  SRCFILE(MYLIB/QRPGLESRC)
           DBGVIEW(*SOURCE)
```

The module is created in the library MYLIB with the same name as the source file on which it is based, namely, TRNSRPT. This module object can be debugged using a source view. For information on the other views available, see "Preparing a Program for Debugging" on page 115.

A compiler listing for the TRNSRPT module will be produced.

```
 *================================================================*
 * MODULE NAME:    TRNSRPT                                         *
 * RELATED FILES:  TRNSDTA  (PF)                                   *
 * RELATED SOURCE: INCALC  (calculations procedure)                *
 * DESCRIPTION:    This program reads every tranasction record     *
 *                 stored in the physical file TRNSDTA. It calls   *
 *                 INCALC which performs calculations and          *
 *                 returns a value back.  This program then        *
 *                 prints the transaction record out.              *
 *================================================================*
FTRNSDTA   IP   E           DISK
FQSYSPRT   O    F   80      PRINTER
 *
ITRNREC       01
```

Figure 23 (Part 1 of 2). Source for TRNSRPT module

```
     *  Call the calculation procedure INCALC.                      *
     C   01              CALLB     'INCALC'
     C                   PARM                    PROD
     C                   PARM                    QTY
     C                   PARM                    DISCOUNT
     C                   PARM                    INCOME           11 1
     C
     OQSYSPRT   D    01                     1
     O                                       12 'PRODUCT: '
     O                       PROD            25
     O                                       40 'QUANTITY: '
     O                       QTY             45
     O                                       60 'INCOME: '
     O                       INCOME       2  75
```

Figure 23 (Part 2 of 2). Source for TRNSRPT module

The DDS for the module TRNSDTA is shown in Figure 24.

```
     A*****************************************************************
     A* RELATED FILES:  TRNSRPT                                       *
     A* DESCRIPTION:    This is the physical file TRNSDTA. It has     *
     A*                 one record format called TRNSREC.             *
     A*****************************************************************
     A* PARTS TRANSACTION FILE -- TRNSDTA
     A          R TRNREC
     A            PROD          10S 0       TEXT('Product')
     A            QTY            5S 0       TEXT('Quantity')
     A            DISCOUNT       3S 2       TEXT('Discount')
```

Figure 24. DDS for TRNSDTA

Additional Examples

For additional examples of creating modules, see:

- "Sample Service Program" on page 65, for an example of creating a module for a service program.
- "Binding to a Program" on page 69. for an example of creating a module to be used with a service program.
- "Dynamically Allocating Storage for a Run-Time Array" on page 84, for an example of creating a module for dynamically allocating storage for a run-time array
- "Sample Source for Debug Examples" on page 147, for example of creating an RPG and C module for use in a sample debug program.

Behavior of Bound ILE RPG/400 Modules

In ILE RPG/400, the procedure is the boundary for:

Scope of LR semantics
Scope of open files
Scope of cycle.

Since each ILE RPG/400 module consists of one and only one procedure, procedure scope is the same as module scope. When you bind together multiple modules, the resulting program will have multiple cycles, namely, one per ILE RPG/400 procedure.

Note: Procedure scope may not equal module scope in other ILE languages since they may treat the notion of procedure differently.

Related CL Commands

The following CL commands can be used with modules:

- Display Module (DSPMOD)
- Change Module (CHGMOD)
- Delete Module (DLTMOD)
- Work with Modules (WRKMOD)

For further information on these commands see the *CL Reference*.

Binding Modules into a Program

Binding is the process of creating a runnable ILE program by combining one or more modules and optional service programs, and resolving symbols passed between them. The system code that does this combining and resolving is called a **binder** on the AS/400 system.

As part of the binding process, a procedure must be identified as the startup procedure, or **program entry procedure**. When a program is called, the program entry procedure receives the parameters from the command line and is given initial control for the program. The user's code associated with the program entry procedure is the user entry procedure.

Every ILE RPG/400 module implicitly contains a program entry procedure. However, this may not be true of other ILE languages. For example, an ILE C/400 module contains a program entry procedure only if it contains a main() function.

Figure 25 gives an idea of the internal structure of a program object. It shows the program object TRPT, which was created by binding the two modules TRNSRPT and INCALC. TRNSRPT is the entry module.

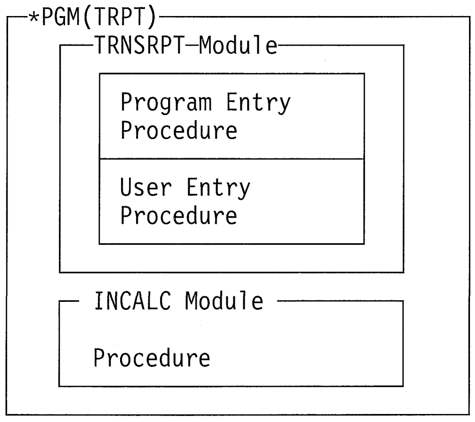


Figure 25. Structure of Program TRPT

Within a bound object, procedures can interrelate using static procedure calls. These bound calls are faster than external calls. Therefore, an application consisting of a single bound program with many bound calls should perform faster than a similar application consisting of separate programs with many external interapplication calls.

In addition to binding modules together, you can also bind them to service programs (type *SRVPGM). Service programs allow you to code and maintain modules separately from the program modules. Common routines can be created as service programs and if the routine changes, the change can be incorporated by binding the service program again. The programs that use these common routines do not have to be recreated. For information on creating service programs see Chapter 7, "Creating a Service Program" on page 63.

For information on the binding process and the binder, refer to the *ILE Concepts*.

Using the CRTPGM Command

The Create Program (CRTPGM) command creates a program object from one or more previously created modules and, if required, one or more service programs. You can bind modules created by any of the ILE Create Module commands, CRTRPGMOD, CRTCMOD, CRTCBLMOD or CRTCLMOD.

Note: The modules and/or service programs required must have been created previously.

Before you create a program object using the CRTPGM command, you should:

1. Establish a program name.
2. Identify the module or modules, and if required, service programs you want to bind into a program object.
3. Identify the module which has the program entry procedure.

   You indicate which module contains the program entry procedure through the ENTMOD parameter of CRTPGM. The default is ENTMOD(*FIRST), meaning that the module containing the first program entry procedure found in the list for the MODULE parameter is the entry module.

   If you are binding more than one ILE RPG/400 modules together, then you should specify *FIRST or else specify the module name with the program entry procedure. You can use ENTMOD(*ONLY) when you are binding only one module into a program object or if you are binding several modules, but only one contains a program entry procedure. For example, if you an RPG module to a C module without a main() function, then you can specify ENTMOD(*ONLY).
4. Identify the activation group that the program is to use.

   Specify the named activation group QILE if your program has no special requirements or if you are not sure which group to use. In general, it is a good idea to run an application in its own activation group. Therefore, you may want to name the activation group after the application.

   Note that the default activation group for CRTPGM is *NEW. This means that your program will run in its own activation group, and the activation group will terminate when the program does. This means that whether or not you set on LR, your program will have a fresh copy of its data the next time you call it. For more information on activation groups see "Specifying an Activation Group" on page 80.

To create a program object using the CRTPGM command, perform the following steps:

1. Enter the CRTPGM command.
2. Enter the appropriate values for the command parameter.

Table 4 lists the CRTPGM command parameters and their default values. For a full description of the CRTPGM command and its parameters, refer to the *CL Reference*.

Table 4. Parameters for CRTPGM Command and their Default Values

Parameter Group	**Parameter(Default Value)**

Identification	PGM(*library name/program name*)
	MODULE(*PGM)

Program access	ENTMOD(*FIRST)

Binding	BNDSRVPGM(*NONE)
	BNDDIR(*NONE)

Run time	ACTGRP(*NEW)

Miscellaneous	OPTION(*GEN *NODUPPROC *NODUPVAR *WARN *RSLVREF)
	DETAIL(*NONE)
	ALWUPD(*YES)
	ALWRINZ(*NO)
	REPLACE(*YES)
	AUT(*LIBCRTAUT)
	TEXT(*ENTMODTXT)
	TGTRLS(*CURRENT)
	USRPRF(*USER)

Once you have entered the CRTPGM command, the system performs the following actions:

1. Copies listed modules into what will become the program object and links any service programs to the program object.
2. Identifies the module containing the program entry procedure and locates the first import in this module.
3. Checks the modules in the order in which they are listed and matches the first import with a module export.
4. Returns to the first module and locates the next import.
5. Resolves all imports in the first module.
6. Continues to the next module and resolves all imports.
7. Resolves all imports in each subsequent module until all of the imports have been resolved.
8. If any imports cannot be resolved with an export, the binding process terminates without creating a program object.
9. Once all the imports have been resolved, the binding process completes and the program object is created.

Note: If you have specified that a variable is to be exported (using the EXPORT keyword), it is possible that the variable name will be identical to a variable in another procedure within the bound program object. In this case, the results may not be as expected. See *ILE Concepts* for information on how to handle this situation.

Binding Multiple Modules

This example shows you how to use the CRTPGM command to bind two ILE RPG/400 modules into a program TRPT. In this program, the following occurs:

- The module TRNSRPT reads each transaction record from a file TRNSDTA.
- It then calls the module INCALC using the bound procedure call mechanism CALLB.
- INCALC then calculates the income pertaining to each transaction.
- INCALC returns to TRNSRPT.
- TRNSRPT then prints the transaction record.

Source for TRNSRPT, INCALC, and TRNSDTA is shown in Figure 23 on page 52, Figure 22 on page 51 and Figure 24 on page 53 respectively.

1. First create the module TRNSRPT. Type:

   ```
   CRTRPGMOD MODULE(MYLIB/TRNSRPT)
   ```

2. Then create module INCALC by typing:

   ```
   CRTRPGMOD MODULE(MYLIB/INCALC)
   ```

3. To create the program object, type:

   ```
   CRTPGM PGM(MYLIB/TRPT) MODULE(TRNSRPT INCALC)
          ENTMOD(*FIRST) ACTGRP(TRPT)
   ```

The CRTPGM command creates a program object TRPT in the library MYLIB.

Note that TRNSRPT is listed first in the MODULE parameter; this means it will contain the program entry procedure. If INCALC had been listed first, it would contain the program entry procedure; however, TRNSRPT would never run since it would not be called by INCALC.

The program will run in the named activation group TRPT. It runs in a named group to ensure that no other programs can affect its resources.

Additional Examples

For additional examples of creating programs, see:

- "Binding to a Program" on page 69, for an example of binding a module and a service program.
- "Sample Source for Debug Examples" on page 147, for an example of creating a program consisting of an RPG and C module.

Related CL Commands

The following CL commands can be used with programs:

- Change Program (CHGPGM)
- Delete Program (DLTPGM)
- Display Program (DSPPGM)
- Display Program References (DSPPGMREF)
- Update Program (UPDPGM)
- Work with Program (WRKPGM)

For further information on these commands see the *CL Reference*.

Using a Binder Listing

The binding process can produce a listing that describes the resources used, symbols and objects encountered, and problems that were resolved or not resolved in the binding process. The listing is produced as a spooled file for the job you use to enter the CRTPGM command. The command default is to not produce this information, but you can choose a DETAIL parameter value to generate it at three levels of detail:

- *BASIC
- *EXTENDED
- *FULL

The binder listing includes the following sections depending on the value specified for DETAIL:

Table 5. Sections of the Binder Listing based on DETAIL Parameter

Section Name	*BASIC	*EXTENDED	*FULL
Command Option Summary	X	X	X
Brief Summary Table	X	X	X
Extended Summary Table		X	X
Binder Information Listing		X	X
Cross-Reference Listing			X
Binding Statistics			X

The information in this listing can help you diagnose problems if the binding was not successful, or give feedback about what the binder encountered in the process. You may want to store the listing for an ILE program in the file where you store the modules or the module source for a program. To copy this listing to a database file, you can use the Copy Spool File (CPYSPLF) command.

Note: The CRTBNDRPG command will not create a binder listing. However, if any binding errors occur during the binding phase, the errors will be noted in your job log, and the compiler listing will include a message to this effect.

For an example of a basic binder listing, see “Sample Binder Listing” on page 71.

For more information on binder listings see *ILE Concepts*.

Changing a Module or Program

An ILE object may need to be changed for enhancements or for maintenance reasons. You can isolate what needs to be changed by using debugging information or the binder listing from the CRTPGM command. From this information you can determine what module needs to change, and often, what procedure or field needs to change.

In addition, you may want to change the optimization level or observability of a module or program. This often happens when you want to debug an program or module, or when you are ready to put a program into production. Such changes can be performed more quickly and use fewer system resources than re-creating the object in question.

Finally, you may want to reduce the program size once you have completed an application. ILE programs have additional data added to them which may make them larger than a similar OPM program.

Each of the above requires different data to make the change. The resources you need may not be available to you for an ILE program.

The following sections tell you how to

- Update a program
- Change the optimization level
- Change observability
- Reduce the object size

Note: In the remainder of this section the term 'object' will be used to refer to either an ILE module or ILE program.

Using the UPDPGM Command

In general, you can update a program by replacing modules as needed. You do not have to re-create the program. This is helpful if you are supplying an application to other sites. You need only send the revised modules, and the receiving site can update the application using the UPDPGM or UPDSRVPGM command.

The UPDPGM command works with both program and module objects. The parameters on the command are very similar to those on the CRTPGM command. For example, to replace a module in a program, you would enter the module name for MODULE parameter and the library name. The UPDPGM command requires that the modules to be replaced be in the same libraries as when the program was created. You can specify that all modules are to be replaced, or some subset.

The UPDPGM command requires that the module object be present. Thus, it is easier to use the command when you have created the program using separate compile and bind steps. Since the module object already exists, you simply specify its name and library when issuing the command.

To update a program created by CRTBNDRPG command, you must ensure that the revised module is in the library QTEMP. This is because the temporary module used when the CRTBNDRPG command was issued, was created in QTEMP. Once the module is in QTEMP, you can issue the UPDPGM command to replace the module.

For more information, see *ILE Concepts* and *CL Reference*.

Changing the Optimization Level

Optimizing an object means looking at the compiled code, determining what can be done to make the run-time performance as fast as possible, and making the necessary changes. In general, the higher the optimizing request, the longer it takes to create an object. At run time the highly optimized program or service program should run faster than the corresponding nonoptimized program or service program.

However, at higher levels of optimization, the values of fields may not be accurate when displayed in a debug session, or after recovery from exception. In addition, optimized code may have altered breakpoints and step locations used by the source debugger, since the optimization changes may rearrange or eliminate some statements.

To ensure that the contents of a field reflect their most current value, especially after exception recovery, you can use the NOOPT keyword on the corresponding Definition specification. For more information, see “Optimization Considerations” on page 160.

To circumvent this problem while debugging, you can lower the optimization level of a module to display fields accurately as you debug a program, and then raise the level again afterwards to improve the program efficiency as you get the program ready for production.

To determine the current optimization level of a *program* object, use the DSPPGM command. Display 3 of this command indicates the current level. To change the optimization level of a program, use the CHGPGM command. On the Optimize program parameter you can specify one the following values: *FULL, *BASIC, *NONE. These are the same values which can be specified on the OPTIMIZE parameters of either create command. The program is automatically re-created when the command runs.

Similarly, to determine the current optimization level of a *module*, use the DSPMOD command. Display 1, page 2 of this command indicates the current level. To change the optimization level, use the CHGMOD command. You then need to re-create the program either using UPDPGM or CRTPGM.

Removing Observability

Observability involves two kinds of data that can be stored with an object, and that allow the object to be changed without recompiling the source. The addition of this data increases the size of the object. Consequently, before you may want to remove the data in order to reduce object size. But once the data is removed, observability is also removed. You must recompile the source and recreate the program to replace the data. The two types of data are:

Create Data Represented by the *CRTDTA value. This data is necessary to translate the code to machine instructions. The object must have this data before you can change the optimization level.

Debug Data Represented by the *DBGDTA value. This data is necessary to allow an object to be debugged.

Use the CHGPGM command or the CHGMOD command to remove either kind of data from a program or module respectively. remove both types, or remove none. Removing all observability reduces an object to its minimum size (without compression). It is not possible to change the object in any way unless you re-create it. Therefore, ensure that you have all source required to create the program or have a comparable program object with CRTDATA. To re-create it, you must have authorization to access the source code.

Reducing an Object's Size

reducing object size

The create data (*CRTDTA) associated with an ILE program or module may make up more than half of the object's size. By removing or compressing this data, you will reduce the secondary storage requirements for your programs significantly.

If you remove the data, ensure that you have all source required to create the program or have a comparable program object with CRTDATA. Otherwise you will not be able to change the object.

An alternative is to compress the object, using the Compress Object (CPROBJ) command. A compressed object takes up less system storage than an uncompressed one. If the compressed program is called, the part of the object containing the runnable code is automatically decompressed. You can also decompress a compressed object by using the Decompress Object (DCPOBJ) command.

For more information on these CL commands, see the *CL Reference*.

Chapter 7. Creating a Service Program

This chapter provides:

- An overview of the service program concept
- Strategies for creating service programs
- A brief description of the CRTSRVPGM command
- An example of a service program

Service Program Overview

A service program is a bound program (type *SRVPGM) consisting of a set of procedures that can be called by procedures in other bound programs.

Service programs are typically used for common functions that are frequently called within an application and across applications. For example, the ILE compilers use service programs to provide runtime services such as math functions and input/output routines. Service programs enable reuse, simplify maintenance, and reduce storage requirements.

A service program differs from a program in two ways:

- It does not contain a program entry procedure. This means that you cannot call a service program using the CALL operation.
- A service program is bound into a program or other service programs using binding by reference.

When you bind a service program to a program, the contents of the service program are not copied into the bound program. Instead, linkage information of the service program is bound into the program. This is called 'binding by reference' in contrast to the static binding process used to bind modules into programs.

Because a service program is bound by reference to a program, you can call the service program's exported procedures using the CALLB operation. The initial call has a certain amount of overhead because it is bound by reference. However, subsequent calls to any of its procedures are faster than program calls.

The set of exports contained in a service program are the interface to the services provided by it. You can use the Display Service Program (DSPSRVPGM) command or the service program listing to see what variable names are available for use the calling procedures.

Strategies for Creating Service Programs

When creating a service program, you should keep in mind:

1. whether you intend to update the program at a later date
2. whether any updates will involve changes to the interface (namely, the imports and exports used).

If the interface to a service program changes, then you may have to re-bind *any* programs bound to the original service program. However, if the changes required are upward-compatible, you may be able to reduce the amount of re-binding if you created the service program using binder language. In this case, after updating the binder language source to identify the new exports you need to re-bind only those programs that use them.

Binder language gives you control over the exports of a service program. This control can be very useful if you want to:

- Mask certain service program procedures from service-program users
- Fix problems
- Enhance function
- Reduce the impact of changes to the users of an application.

See “Sample Service Program” on page 65 for an example of using binder language to create a service program.

For information on binder language, masking exports, and other service program concepts, see *ILE Concepts*.

Creating a Service Program Using CRTSRVPGM

You create a service program using the Create Service Program (CRTSRVPGM) command. Any ILE module can be bound into a service program. The module(s) must exist before you can create a service program with it.

Table 6 lists the CRTSRVPGM parameters and their defaults. For a full description of the CRTSRVPGM command and its parameters, refer to the *CL Reference*.

Table 6. Parameters for CRTSRVPGM Command and their Default Values

Parameter Group	**Parameter(Default Value)**

Identification	SRVPGM(*library name/service program name*)
	MODULE(*SRVPGM)

Program access	EXPORT(*SRCFILE)
	SRCFILE(*LIBL/QSRVSRC)
	SRCMBR(*SRVPGM)

Binding	BNDSRVPGM(*NONE)
	BNDDIR(*NONE)

Run time	ACTGRP(*CALLER)

Miscellaneous	OPTION(*GEN *NODUPPROC *NODUPVAR *WARN *RSLVREF)
	DETAIL(*NONE)
	ALWUPD(*YES)
	ALWRINZ(*NO)
	REPLACE(*YES)
	AUT(*LIBCRTAUT)
	TEXT(*ENTMODTXT)
	TGTRLS(*CURRENT)
	USRPRF(*USER)

See “Creating the Service Program” on page 68 for an example of using the CRTSRVPGM command.

Changing A Service Program

You can update or change a service program in the same ways available to a program object. In other words, you can:

- Update the service program (using UPDSRVPGM)
- Change the optimization level (using CHGSRVPGM)
- Remove observability (using CHGSRVPGM)
- Reduce the size (using CPROBJ)

For more information on any of the above points, see "Changing a Module or Program" on page 58.

Related CL commands

The following CL commands are also used with service programs:

- Change Service Program (CHGSRVPGM)
- Display Service Program (DSPSRVPGM)
- Delete Service Program (DLTSRVPGM)
- Update Service Program (UPDSRVPGM)
- Work with Service Program (WRKSRVPGM)

Sample Service Program

The following example shows how to create a service program CVTTOHEX which converts character strings to their hexadecimal equivalent. Two parameters are passed to the service program:

1. a character field (InString) to be converted
2. a character field (HexString) which will contain the 2-byte hexadecimal equivalent

The field HexString is used to contain the result of the conversion and also to indicate the length of the string to be converted. For example, if a character string of 30 characters is passed, but you are only interested in converting the first ten, you would pass a second parameter of 20 bytes (2 times 10). Based on the length of the passed fields, the service program determines the length to handle.

Figure 26 on page 66 shows the source for the service program.

The basic logic of the procedure contained within the service program is listed below:

1. Operational descriptors are used to determine the length of the passed parameters.
2. The length to be converted is determined: it is the lesser of the length of the character string, or one-half the length of the hex string field.
3. Each character in the string is converted to a two-byte hexadecimal equivalent using the subroutine GetHex.
4. The procedure returns to its caller.

The service program makes use of operational descriptors, which is an ILE construct used when the precise nature of a passed parameter is not known ahead of time, in this case the length. The operational descriptors are created on a call to a procedure when you specify the operation extender (D) on the CALLB operation.

To use the operational descriptors, the service program must call the ILE bindable API, CEEDOD (Retrieve Operational Descriptor). This API requires certain parameters which must be defined for the CALLB operation. However, it is the last parameter which provides the information needed, namely, the length. For more information on operational descriptors, see "Using Operational Descriptors" on page 97.)

```
  *================================================================*
  * CVTTOHEX - convert input string to hex output string           *
  *                                                                *
  * Note: Operational descriptors must be passed                   *
  *                                                                *
  *================================================================*

  *----------------------------------------------------------------*
  * Program parameters                                             *
  * 1. Input:   string                  character(n)               *
  * 2. Output:  hex string              character(2 * n)           *
  *----------------------------------------------------------------*
 D InString        S          16383
 D HexString       S          32766
  *----------------------------------------------------------------*
  * Binary parameters for CEEDOD (Retrieve operational descriptor)  *
  *----------------------------------------------------------------*
 D ParmNum         S              9B 0
 D DescType        S              9B 0
 D DataType        S              9B 0
 D DescInfo1       S              9B 0
 D DescInfo2       S              9B 0
 D InLen           S              9B 0
 D HexLen          S              9B 0

  *----------------------------------------------------------------*
  * Other fields used by the program                               *
  *----------------------------------------------------------------*
 D HexDigits       C                   CONST('0123456789ABCDEF')
 D BinDs           DS
 D   BinNum                       4B 0 INZ(0)
 D   BinChar                      1    OVERLAY(BinNum:2)
 D HexDs           DS
 D   HexC1                        1
 D   HexC2                        1
 D InChar          S              1
 D Pos             S              5P 0
 D HexPos          S              5P 0

 C     *ENTRY        PLIST
 C                   PARM                    InString
 C                   PARM                    HexString
```

Figure 26 (Part 1 of 3). Source for Service Program CVTTOHEX

```
 *-----------------------------------------------------------------*
 * Use the operational descriptors to determine the lengths of     *
 * the parameters that were passed.                                *
 *-----------------------------------------------------------------*
C                   CALLB     'CEEDOD'
C                   PARM      1             ParmNum
C                   PARM                    DescType
C                   PARM                    DataType
C                   PARM                    DescInfo1
C                   PARM                    DescInfo2
C                   PARM                    InLen
C                   PARM                    *OMIT
C                   CALLB     'CEEDOD'
C                   PARM      2             ParmNum
C                   PARM                    DescType
C                   PARM                    DataType
C                   PARM                    DescInfo1
C                   PARM                    DescInfo2
C                   PARM                    HexLen
C                   PARM                    *OMIT

 *-----------------------------------------------------------------*
 * Determine the length to handle (minimum of the input length     *
 * and half of the hex length)                                     *
 *-----------------------------------------------------------------*
C                   IF        InLen > HexLen / 2
C                   EVAL      InLen = HexLen / 2
C                   ENDIF

 *-----------------------------------------------------------------*
 * For each character in the input string, convert to a 2-byte     *
 * hexadecimal representation (for example, '5' --> 'F5')         *
 *-----------------------------------------------------------------*
C                   EVAL      HexPos = 1
C                   DO        InLen         Pos
C                   EVAL      InChar = %SUBST(InString:Pos:1)
C                   EXSR      GetHex
C                   EVAL      %SUBST(HexString:HexPos:2) = HexDs
C                   EVAL      HexPos = HexPos + 2
C                   ENDDO

 *-----------------------------------------------------------------*
 * Done; return to caller.                                         *
 *-----------------------------------------------------------------*
C                   RETURN
```

Figure 26 (Part 2 of 3). Source for Service Program CVTTOHEX

```
 *=================================================================*
 * GetHex - subroutine to convert 'InChar' to 'HexDs'              *
 *                                                                 *
 * Use division by 16 to separate the two hexadecimal digits.      *
 * The quotient is the first digit, the remainder is the second.   *
 *=================================================================*
C     GetHex        BEGSR
C                   EVAL      BinChar = InChar
C     BinNum        DIV       16            X1                 5 0
C                   MVR                     X2                 5 0
 *-----------------------------------------------------------------*
 * Use the hexadecimal digit (plus 1) to substring the list of     *
 * hexadecimal characters '012...CDEF'.                            *
 *-----------------------------------------------------------------*
C                   EVAL      HexC1 = %SUBST(HexDigits:X1+1:1)
C                   EVAL      HexC2 = %SUBST(HexDigits:X2+1:1)
C                   ENDSR
```

Figure 26 (Part 3 of 3). Source for Service Program CVTTOHEX

When designing this service program, it was decided to make use of binder language to determine the interface, so that the program could be more easily updated at a later date. Figure 27 shows the binder language needed define the exports of the service program CVTTOHEX.

```
STRPGMEXP SIGNATURE(CVTHEX)
   EXPORT SYMBOL('CVTTOHEX')
ENDPGMEXP
```

Figure 27. Source for Binder Language for CVTTOHEX

The parameter SIGNATURE on STRPGMEXP identifies the interface that the service program will provide. In this case, the export identified in the binder language is the interface. Any program bound to CVTTOHEX will make use of this signature.

The binder language EXPORT statements identify the exports of the service program. You need one for each procedure whose exports you want to make available to the caller. In this case, the service program contains one module which contains one procedure. Hence, only one EXPORT statement is required.

For more information on binder language and signatures, see *ILE Concepts*.

Creating the Service Program

To create the service program CVTTOHEX, follow these steps:

1. Create the module CVTTOHEX from the source in Figure 26 on page 66, by entering:

   ```
   CRTRPGMOD MODULE(MYLIB/CVTTOHEX) SRCFILE(MYLIB/QRPGLESRC)
   ```

2. Create the service program using the module CVTTOHEX and the binder language shown in Figure 27.

   ```
   CRTSRVPGM SRVPGM(MYLIB/CVTTOHEX)  MODULE(*SRVPGM)
             EXPORT(*SRCFILE)  SRCFILE(MYLIB/QSRVSRC)
             SRCMBR(*SRVPGM)
   ```

The last three parameters in the above command identify the exports which the service program will make available. In this case, it is based on the source found in the member CVTTOHEX in the file QSRVSRC in the library MYLIB.

Note that a binding directory is not required here because all modules needed to create the service program have been specified with the MODULE parameter.

The service program CVTTOHEX will be created in the library MYLIB. It can be debugged using a statement view; this is determined by the default DBGVIEW parameter on the CRTRPGMOD command. No binder listing is produced.

Binding to a Program

To complete the example, we will create an 'application' consisting of a program CVTHEXPGM which is bound to the service program. It uses a seven-character string which it passes to CVTTOHEX twice, once where the value of the hex string is 10 (that is, convert 5 characters) and again where the value is 14, that is, the actual length.

Note that the program CVTHEXPGM serves to show the use of the service program CVTTOHEX. In a real application the caller of CVTTOHEX would have another primary purpose other than testing CVTTOHEX. Furthermore, a service program would normally be used by many other programs, or many times by a few programs; otherwise the overhead of initial call does not justify making it into a service program.

To create the application follow these steps:

1. Create the module from the source in Figure 28 on page 70, by entering:

   ```
   CRTRPGMOD MODULE(MYLIB/CVTHEXPGM) SRCFILE(MYLIB/QRPGLESRC)
   ```

2. Create the program by typing

   ```
   CRTPGM PGM(MYLIB/CVTHEXPGM)
          BNDSRVPGM(MYLIB/CVTTOHEX)
          DETAIL(*BASIC)
   ```

   When CVTHEXPGM is created, it will include information regarding the interface it uses to interact with the service program. This is the same as reflected in the binder language for CVTTOHEX.

3. Call the program, by typing:

   ```
   CALL CVTHEXPGM
   ```

   During the process of making CVTHEXPGM ready to run, the system verifies that:

   - The service program CVTTOHEX in library MYLIB can be found
   - The public interface used by CVTHEXPGM when it was created is still valid at run time.

   If either of the above is not true, then an error message is issued.

The output of CVTHEXPGM is shown below. (The input string is 'ABC123*'.)

```
Result14++++++
Result10++
C1C2C3F1F2        10 character output
C1C2C3F1F2F35C    14 character output
```

```
 *--------------------------------------------------------------*
 * Program to test Service Program CVTTOHEX                     *
 *                                                              *
 * 1. Use a 7-character input string                            *
 * 2. Convert to a 10-character hex string (only the first five *
 *    input characters will be used because the result is too   *
 *    small for the entire input string)                        *
 * 3. Convert to a 14-character hex string (all seven input     *
 *    characters will be used because the result is long enough)*
 *--------------------------------------------------------------*

FQSYSPRT   O    F   80        PRINTER

D ResultDS        DS
D   Result14              1     14
D   Result10              1     10
D InString        S              7
D Comment         S             25

C                   EVAL      InString = 'ABC123*'

 *--------------------------------------------------------------*
 * Pass character string and the 10-character result field      *
 * using a CALLB(D).  The operation extender (D) will create     *
 * operational descriptors for the passed parameters.            *
 * These are required by the called procedure in CVTTOHEX.       *
 *--------------------------------------------------------------*
C                   EVAL      Comment = '10 character output'
C                   CLEAR                   ResultDS
C                   CALLB(D)  'CVTTOHEX'
C                   PARM                    InString
C                   PARM                    Result10
C                   EXCEPT

 *--------------------------------------------------------------*
 * Pass character string and the 14-character result field      *
 * using a CALLB(D).                                             *
 *--------------------------------------------------------------*
C                   EVAL      Comment = '14 character output'
C                   CLEAR                   ResultDS
C                   CALLB(D)  'CVTTOHEX'
C                   PARM                    InString
C                   PARM                    Result14
C                   EXCEPT

C                   EVAL      *INLR = *ON

OQSYSPRT   H    1P
O                                             'Result14++++++'
OQSYSPRT   H    1P
O                                             'Result10++'
OQSYSPRT   E
O                       ResultDS
O                       Comment             +5
```

Figure 28. Source for Test Program CVTHEXPGM

Updating the Service Program

Because of the binder language, the service program could be updated and the program CVTHEXPGM would not have to be re-compiled. For example, you could add a new procedure to CVTTOHEX you would:

1. Create a module object for the new procedure.
2. Modify the binder language source to handle the interface associated with the new procedure. This would involve adding any new export statements *following* the existing ones.
3. Bind the new module to service program CVTTOHEX by re-creating the service program.

New programs can access the new function. Since the old exports are in the same order they can still be used by the existing programs. Until it is necessary to also update the existing programs, they do not have to be re-compiled.

For more information on updating service programs, see *ILE Concepts*.

Sample Binder Listing

Figure 29 on page 72 shows a sample binder listing for the CVTHEXPGM. The listing is an example of a basic listing. For more information on binder listings, see "Using a Binder Listing" on page 58 and also *ILE Concepts*.

```
                                                 Create Program                                                        Page     1
5763SS1 V3R1M0  940909                                                              MYLIB/CVTHEXPGM    AS400S01  09/22/94  23:24:00
 Program . . . . . . . . . . . . . . . . . . . . . :   CVTHEXPGM
   Library . . . . . . . . . . . . . . . . . . . . :     MYLIB
 Program entry procedure module . . . . . . . . . . :   *FIRST
   Library . . . . . . . . . . . . . . . . . . . . :
 Activation group . . . . . . . . . . . . . . . . . :   *NEW
 Creation options . . . . . . . . . . . . . . . . . :   *GEN        *NODUPPROC  *NODUPVAR   *WARN       *RSLVREF
 Listing detail . . . . . . . . . . . . . . . . . . :   *BASIC
 Allow Update . . . . . . . . . . . . . . . . . . . :   *YES
 User profile . . . . . . . . . . . . . . . . . . . :   *USER
 Replace existing program . . . . . . . . . . . . . :   *YES
 Authority  . . . . . . . . . . . . . . . . . . . . :   *LIBCRTAUT
 Target release . . . . . . . . . . . . . . . . . . :   *CURRENT
 Allow reinitialization . . . . . . . . . . . . . . :   *NO
 Text . . . . . . . . . . . . . . . . . . . . . . . :   *ENTMODTXT
 Module      Library                Module      Library               Module      Library               Module      Library
 CVTHEXPGM   MYLIB
 Service                            Service                           Service                           Service
 Program     Library                Program     Library               Program     Library               Program     Library
 CVTTOHEX    MYLIB
 Binding                            Binding                           Binding                           Binding
 Directory   Library                Directory   Library               Directory   Library               Directory   Library
 *NONE
                                                 Create Program                                                        Page     2
5763SS1 V3R1M0  940909                                                              MYLIB/CVTHEXPGM    AS400S01  09/22/94  23:24:00
                                                      Brief Summary Table
 Program entry procedures . . . . . . . . . . . :   1
   Symbol     Type       Library     Object      Identifier
              *MODULE    MYLIB       CVTHEXPGM   _QRNP_PEP_CVTHEXPGM
 Multiple strong definitions  . . . . . . . . . :   0
 Unresolved references  . . . . . . . . . . . . :   0

                                   * * * * *   E N D  O F  B R I E F  S U M M A R Y  T A B L E   * * * * *
                                                 Create Program                                                        Page     3
5763SS1 V3R1M0  940909                                                              MYLIB/CVTHEXPGM    AS400S01  09/22/94  23:24:00
                                                       Binding Statistics
 Symbol collection CPU time . . . . . . . . . . . . . . . . . :               .016
 Symbol resolution CPU time . . . . . . . . . . . . . . . . . :               .004
 Binding directory resolution CPU time  . . . . . . . . . . . :               .175
 Binder language compilation CPU time . . . . . . . . . . . . :               .000
 Listing creation CPU time  . . . . . . . . . . . . . . . . . :               .068
 Program/service program creation CPU time  . . . . . . . . . :               .234
 Total CPU time . . . . . . . . . . . . . . . . . . . . . . . :               .995
 Total elapsed time . . . . . . . . . . . . . . . . . . . . . :              3.531
                                    * * * * *   E N D  O F  B I N D I N G  S T A T I S T I C S   * * * * *
*CPC5D07 - Program CVTHEXPGM created in library MYLIB.
                               * * * * *   E N D  O F  C R E A T E  P R O G R A M  L I S T I N G   * * * * *
```

Figure 29. Basic Binder listing for CVTHEXPGM

Chapter 8. Running a Program

This chapter shows you how to:

- Run a program and pass parameters using the CL CALL command
- Run a program from a menu-driven application
- Run a program using a user-created command
- Manage activation groups
- Manage run-time storage.

In addition, you can run a program using:

- The Programmer Menu. *CL Programming* contains information on this menu.
- The Start Programming Development Manager (STRPDM) command. *ADTS/400: Programming Development Manager* contains information on this command.
- The QCMDEXC program. *CL Programming* contains information on this program.
- A high-level language. Chapter 9, "Calling Programs and Procedures" on page 91. provides information on running programs from another HLL or calling service programs and procedures,

Running a Program Using the CL CALL Command

You can use the CL CALL command to run a program (type *PGM). You can use the command interactively, as part of a batch job, or include it in a CL program. If you need prompting, type CALL and press F4 (Prompt). If you need help, type CALL and press F1 (Help).

For example, to call the program EMPRPT from the command line, type:

```
CALL EMPRPT
```

The program object specified must exist in a library and this library must be contained in the library list *LIBL. You can also explicitly specify the library in the CL CALL command as follows:

```
CALL MYLIB/EMPRPT
```

See the *CL Reference* for further information about using the CL CALL command.

Once you call your program, the OS/400 system performs the instructions found in the program.

Passing Parameters using the CL CALL Command

You use the PARM option of the CL CALL command to pass parameters to the ILE program when you run it.

```
CALL PGM(program-name)
     PARM(parameter-1 parameter-2 ... parameter-n)
```

You can also type the parameters without specifying any keywords:

```
CALL library/program-name (parameter-1 parameter-2 ... parameter-n)
```

Each parameter value can be specified as a CL program variable or as one of the following:

- a character string constant
- a numeric constant
- a logical constant

If you are passing parameters to a program where an ILE RPG/400 procedure is the program entry procedure, then that program must have one and only one *ENTRY PLIST specified. The parameters which follow (in the PARM statements) should correspond on a one-to-one basis to those passed through the CALL command.

Refer to the CALL Command in the *CL Reference* or to the section on "Passing Parameters between Programs" in *CL Programming* for a full description of how parameters are handled.

For example, the program EMPRPT2 requires the correct password to be passed to it when it first started; otherwise it will not run. Figure 30 shows the source.

1. To create the program, type:

   ```
   CRTBNDRPG PGM(MYLIB/EMPRPT2)
   ```

2. To run the program, type:

   ```
   CALL MYLIB/EMPRPT2 (HELLO)
   ```

   When the CALL command is issued, the contents of the parameter passed by the command is stored and the program parameter PSWORD points to its location. The program then checks to see if the contents of PSWORD matches the value stored in the program, ('HELLO'). In this case, the two values are the same, and so the program continues to run.

```
  *==============================================================*
  * PROGRAM NAME:   EMPRPT2                                      *
  * RELATED FILES:  EMPMST   (PHYSICAL FILE)                     *
  *                 PRINT    (PRINTER FILE)                      *
  * DESCRIPTION:    This program prints employee information     *
  *                 stored in the file EMPMST if the password    *
  *                 entered is correct.                          *
  *                 Run the program by typing "CALL library name/ *
  *                 EMPRPT2 (PSWORD)" on the command line, where *
  *                 PSWORD is the password for this program.     *
  *                 The password for this program is 'HELLO'.   *
  *==============================================================*
 FPRINT     O    F   80        PRINTER
 FEMPMST    IP   E            K DISK

 IEMPREC       01
```

Figure 30 (Part 1 of 2). ILE RPG/400 Program that Requires Parameters at Run Time

```
 *---------------------------------------------------------------*
 * The entry parameter list is specified in this program.        *
 * There is one parameter, called PSWORD, and it is a            *
 * character field 5 characters long.                            *
 *---------------------------------------------------------------*

C     *ENTRY        PLIST
C                   PARM                    PSWORD         5
 *---------------------------------------------------------------*
 * The password for this program is 'HELLO'.  The field PSWORD   *
 * is checked to see whether or not it contains 'HELLO'.         *
 * If it does not, the last record indicator (LR) and *IN99      *
 * are set on.  *IN99 controls the printing of messages.         *
 *---------------------------------------------------------------*
C     PSWORD        IFNE      'HELLO'
C                   SETON                                        LR99
C                   ENDIF

OPRINT    H    1P                  2  6
O                                     50 'EMPLOYEE INFORMATION'
O         H    1P
O                                     12 'NAME'
O                                     34 'SERIAL #'
O                                     45 'DEPT'
O                                     56 'TYPE'
O         D    01N99
O                      ENAME          20
O                      ENUM           32
O                      EDEPT          45
O                      ETYPE          55
O         D    99
O                                     16 '***'
O                                     40 'Invalid Password Entered'
O                                     43 '***'
```

Figure 30 (Part 2 of 2). ILE RPG/400 Program that Requires Parameters at Run Time

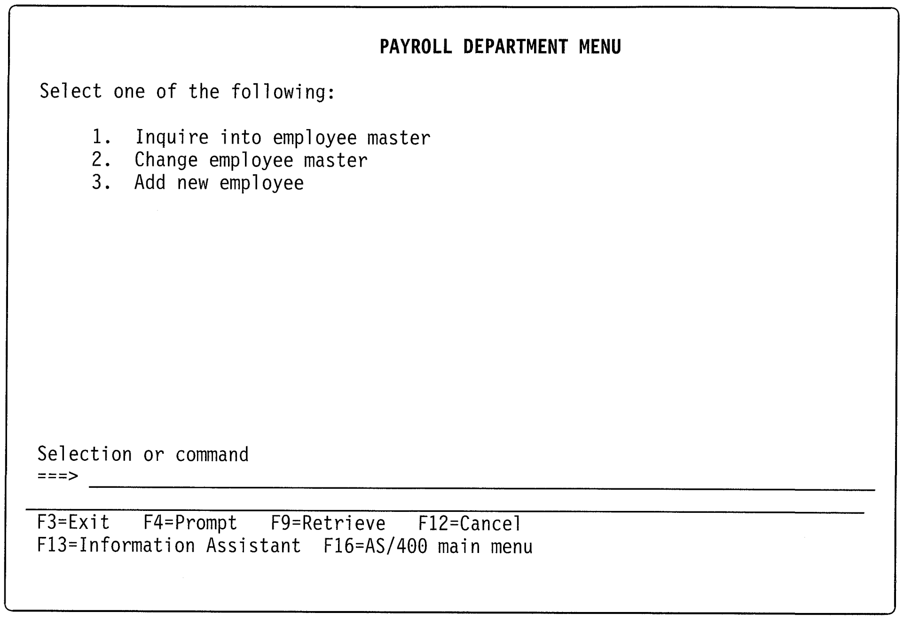
Figure 31 shows the DDS which is referenced by the EMPRPT source.

```
A*****************************************************************
A* DESCRIPTION:  This is the DDS for the physical file EMPMST.   *
A*               It contains one record format called EMPREC.    *
A*               This file contains one record for each employee *
A*               of the company.                                 *
A*****************************************************************
A*
A          R EMPREC
A            ENUM           5  0       TEXT('EMPLOYEE NUMBER')
A            ENAME         20          TEXT('EMPLOYEE NAME')
A            ETYPE          1          TEXT('EMPLOYEE TYPE')
A            EDEPT          3  0       TEXT('EMPLOYEE DEPARTMENT')
A            ENHRS          3  1       TEXT('EMPLOYEE NORMAL WEEK HOURS')
A          K ENUM
```

Figure 31. DDS for EMPRPT

Running a Program From a Menu-Driven Application

Another way to run an ILE program is from a menu-driven application. The work-station user selects an option from a menu, which in turn calls a particular program. Figure 32 illustrates an example of an application menu.



```
                                   PAYROLL DEPARTMENT MENU

 Select one of the following:

      1. Inquire into employee master
      2. Change employee master
      3. Add new employee










 Selection or command
 ===> ____________________________________________________________

 F3=Exit   F4=Prompt   F9=Retrieve   F12=Cancel
 F13=Information Assistant  F16=AS/400 main menu
```

Figure 32. Example of an Application Menu

The menu shown in Figure 32 is displayed by a menu program in which each option calls a separate ILE program. You can create the menu by using STRSDA and selecting option 2 ('Design menus').

Figure 33 on page 77 shows the DDS for the display file of the above PAYROLL DEPARTMENT MENU. The source member is called PAYROL and has a source type of MNUDDS. The file was created using SDA.

```
     A* Free Form Menu: PAYROL
     A*
     A                                      DSPSIZ(24 80 *DS3                -
     A                                             27 132 *DS4)
     A                                      CHGINPDFT
     A                                      INDARA
     A                                      PRINT(*LIBL/QSYSPRT)
     A          R PAYROL
     A                                      DSPMOD(*DS3)
     A                                      LOCK
     A                                      SLNO(01)
     A                                      CLRL(*ALL)
     A                                      ALWROL
     A                                      CF03
     A                                      HELP
     A                                      HOME
     A                                      HLPRTN
     A                                  1 34'PAYROLL DEPARTMENT MENU'
     A                                      DSPATR(HI)
     A                                  3  2'Select one of the following:'
     A                                      COLOR(BLU)
     A                                  5  7'1.'
     A                                  6  7'2.'
     A                                  7  7'3.'
     A* CMDPROMPT  Do not delete this DDS spec.
     A                                019  2'Selection or command            -
     A                                         '
     A                                  5 11'Inquire'
     A                                  5 19'into'
     A                                  5 24'employee'
     A                                  5 33'master'
     A                                  6 11'Change'
     A                                  6 18'employee'
     A                                  6 27'master'
     A                                  7 11'Add'
     A                                  7 15'new'
     A                                  7 19'employee'
```

Figure 33. Data Description Specification of an Application Menu

Figure 34 shows the source of the application menu illustrated in Figure 32 on page 76. The source member is called PAYROLQQ and has a source type of MNUCMD. It was also created using SDA.

```
PAYROLQQ,1
0001 call RPGINQ
0002 call RPGCHG
0003 call RPGADD
```

Figure 34. Source for Menu Program

You run the menu by entering:

```
GO library name/PAYROL
```

If the user enters 1, 2, or 3 from the application menu, the source in Figure 34 calls the programs RPGINQ, RPGCHG, or RPGADD respectively.

Running a Program Using a User-Created Command

You can create a command to run a program by using a command definition. A **command definition** is an object (type *CMD) that contains the definition of a command (including the command name, parameter descriptions, and validity-checking information), and identifies the program that performs the function requested by the command.

For example, you can create a command, PAY, that calls a program, PAYROLL, where PAYROLL is the name of an RPG program that you want to run. You can enter the command interactively, or in a batch job. See the *CL Programming* for further information about using command definitions.

Replying to Run-time Inquiry Messages

When you run a program with ILE RPG/400 procedures, run-time inquiry messages may be generated. They occur when there is no error indicator or error subroutine (*PSSR or INFSR) to handle the exception. The inquiry messages require a response before the program continues running.

You can add the inquiry messages to a system reply list to provide automatic replies to the messages. The replies for these messages may be specified individually or generally. This method of replying to inquiry messages is especially suitable for batch programs, which would otherwise require an operator to issue replies.

You can add the following ILE RPG/400 inquiry messages to the system reply list:

RNQ0100	RNQ0231	RNQ0501	RNQ1042	RNQ1251
RNQ0101	RNQ0232	RNQ0502	RNQ1051	RNQ1255
RNQ0102	RNQ0299	RNQ0600	RNQ1071	RNQ1261
RNQ0103	RNQ0333	RNQ0601	RNQ1201	RNQ1271
RNQ0112	RNQ0401	RNQ0802	RNQ1211	RNQ1281
RNQ0113	RNQ0402	RNQ0803	RNQ1215	RNQ1282
RNQ0114	RNQ0411	RNQ0804	RNQ1216	RNQ1284
RNQ0120	RNQ0412	RNQ0805	RNQ1217	RNQ1285
RNQ0121	RNQ0413	RNQ0907	RNQ1218	RNQ1286
RNQ0122	RNQ0414	RNQ1011	RNQ1221	RNQ1287
RNQ0123	RNQ0415	RNQ1021	RNQ1222	RNQ1299
RNQ0202	RNQ0421	RNQ1022	RNQ1231	RNQ1331
RNQ0211	RNQ0431	RNQ1031	RNQ1235	RNQ9998
RNQ0221	RNQ0432	RNQ1041	RNQ1241	RNQ9999
RNQ0222	RNQ0450			

Note: ILE RPG/400 inquiry messages have a message id prefix of RNQ.

To add inquiry messages to a system reply list using the Add Reply List Entry command enter:

```
ADDRPYLE sequence-no message-id
```

where *sequence-no* is a number from 1-9999, which reflects where in the list the entry is being added, and *message-id* is the message number you want to add. Repeat this command for each message you want to add.

Use the Change Job (CHGJOB) command (or other CL job command) to indicate that your job uses the reply list for inquiry messages. To do this, you should specify *SYSRPYL for the Inquiry Message Reply (INQMSGRPY) attribute.

The reply list is only used when an inquiry message is sent by a job that has the Inquiry Message Reply (INQMSGRPY) attribute specified as INQMSGRPY(*SYSRPYL). The INQMSGRPY parameter occurs on the following CL commands:

- Change Job (CHGJOB)
- Change Job Description (CHGJOBD)
- Create Job Description (CRTJOBD)
- Submit Job (SBMJOB).

You can also use the Work with Reply List Entry (WRKRPYLE) command to change or remove entries in the system reply list. See the *CL Reference* for details of the ADDRPYLE and WRKRPYLE commands.

Ending an ILE Program

When an ILE program ends normally, the system returns control to the caller. The caller could be a workstation user or another program (such as the menu-handling program).

If an ILE program ends abnormally and the program was running in a different activation group than its caller, then the escape message CEE9901

```
Error message-id caused program to end.
```

is issued and control is returned to the caller.

A CL program can monitor for this exception by using the Monitor Message (MONMSG) command. You can also monitor for exceptions in other ILE languages.

If the ILE program is running in the same activation group as its caller and it ends abnormally, then the message issued will depend on why the program ends. If it ends with a function check, then CPF9999 will be issued. If the exception is issued by an RPG procedure, then it will have a message prefix of RNX.

For more information on exception messages, see “Exception Handling Overview” on page 153.

Managing Activation Groups

An **activation group** is a substructure of a job and consists of system resources (for example, storage, commitment definitions, and open files) that are allocated to run one or more ILE or OPM programs. Activation groups make it possible for ILE programs running in the same job to run independently without intruding on each other (for example, commitment control and overrides). The basic idea is that all programs activated within one activation group are developed as one cooperative application.

You identify the activation group which your ILE program will run in at the time of program creation. The activation group is determined by the value specified on the

ACTGRP parameter when the program object was created. (OPM programs always run in the default activation group; you cannot change their activation group specification.) Once an ILE program (object type *PGM) is activated, it remains activated until the activation group is deleted.

The remainder of this section tells you how to specify an activation group and how to delete one. For more information on activation groups, refer to *ILE Concepts*.

Specifying an Activation Group

You control which activation group your ILE program will run in by specifying a value for the ACTGRP parameter when you create your program (using CRTPGM or CRTBNDRPG) or service program (using CRTSRVPGM).

Note: If you are using the CRTBNDRPG command, you can only specify a value for ACTGRP if the value of DFTACTGRP is *NO.

You can choose one of the following values:

- a named activation group

  A named activation group allows you to manage a collection of ILE programs and service programs as one application. The activation group is created when the first program which specified the activation group name on creation is called. It is then used by all programs and service programs that specify the same activation group name.

  A named activation group ends when it is deleted using the CL command RCLACTGRP. This command can only be used when the activation group is no longer in use. When it is ended, *all* resources associated with the programs and service programs of the named activation group are returned to the system.

  The named activation group QILE is the default value of the ACTGRP parameter on the CRTBNDRPG command. However, because activation groups are intended to correspond to applications, it is recommended that you specify a different value for this parameter. For example, you may want to name the activation group after the application name.

- *NEW

  When *NEW is specified, a new activation group is created whenever the program is called. The system creates a name for the activation group. The name is unique within your job.

  An activation group created with *NEW always ends when the program(s) associated with it end. For this reason, if you plan on returning from your program with LR OFF in order to keep your program active, then you should not specify *NEW for the ACTGRP parameter.

  Note: This value is not valid for service programs. A service program can only run in a named activation group or the activation group of its caller.

  *NEW is the default value for the ACTGRP parameter on the CRTPGM command.

  If you create an ILE RPG/400 program with ACTGRP(*NEW), you can then call the program as many times as you want without returning from earlier calls. With each call, there is a new copy of the program. Each new copy will have its own data, open its files, etc.. However, you must ensure that there is some way to end the calls to 'itself'; otherwise you will just keep creating new activation groups and the programs will never return.

- *CALLER

  The program or service program will be activated into the activation group of the calling program. If an ILE program created with ACTGRP(*CALLER) is called by an OPM program, then it will be activated into the OPM default activation group (*DFTACTGRP).

Running in the OPM Default Activation Group

When an OS/400 job is started, the system creates an activation group to be used by OPM programs. The symbol used to represent this activation group is *DFTACTGRP. You cannot delete the OPM default activation group. It is deleted by the system when your job ends.

OPM programs automatically run in the OPM default activation group. An ILE program will also run in the OPM default activation group when one of the following occurs:

- The program was created with DFTACTGRP(*YES) on the CRTBNDRPG command.
- The program was created with ACTGRP(*CALLER) at the time of program creation and the caller of the program runs in the default activation group. Note that you can only specify ACTGRP(*CALLER) on the CRTBNDRPG command if DFTACTGRP(*NO) is also specified.

Note: The resources associated with a program running in the OPM default activation group via *CALLER will not be deleted until the job ends.

Maintaining OPM RPG/400 and ILE RPG/400 Program Compatibility

If you have an OPM application which consists of several RPG programs, you can ensure that the migrated application will behave like an OPM one if you create the ILE application as follows:

1. Convert each OPM source member using the CVTRPGSRC command, making sure to convert the /COPY members.

   See “Converting Your Source” on page 310 for more information.

2. Using the CRTBNDRPG command, compile and bind each converted source member separately into a program object, specifying DFTACTGRP(*YES).

For more information on OPM-compatible programs. refer to “Strategy 1: OPM-Compatible Application” on page 19.

Deleting an Activation Group

When an activation group is deleted, its resources are reclaimed. The resources include static storage and open files. A *NEW activation group is deleted when the program it is associated with returns to its caller.

Named activation groups (such as QILE) are *persistent* activation groups in that they are not deleted unless explicitly deleted or unless the job ends. The storage associated with programs running in named activation groups is not released until these activation groups are deleted.

The OPM default activation group is also a persistent activation group. The storage associated with ILE programs running in the default activation group is released either when you sign off (for an interactive job) or when the job ends (for a batch job).

If many ILE RPG/400 programs are activated (that is called at least once) system storage may be exhausted. Therefore, you should avoid having ILE programs that use large amounts of static storage run in the OPM default activation group, since the storage will not be reclaimed until the job ends.

Note: An ILE RPG/400 program created DFTACTGRP(*YES) will have its storage released when it ends.

The storage associated with a service program is reclaimed only when the activation group it is associated with ends. If the service program is called into the default activation group, its resources are reclaimed when the job ends.

You can delete a named activation group using the RCLACTGRP command. Use this command to delete a nondefault activation group that is not in use. The command provides options to either delete all eligible activation groups or to delete an activation group by name.

For more information on RCLACTGRP refer to the *CL Reference*. For more information on the RCLACTGRP and activation groups, refer to *ILE Concepts*.

Reclaim Resources Command

The Reclaim Resources (RCLRSC) command is designed to free the resources for programs which are no longer active. The command works differently depending on how the program was created. If the program is an OPM program or was created with DFTACTGRP(*YES), then the RCLRSC command will close open files and free static storage.

For ILE programs which were activated into the OPM default activation group because they were created with *CALLER, files will be closed and storage re-initialized when the RCLRSC command is issued. However, the storage will not be released.

For ILE programs associated with a named activation group, the RCLRSC command has *no* effect. You must use the RCLACTGRP command to free resources in a named activation group.

For more information on the RCLRSC command, refer to *CL Reference*. For more information on the RCLRSC and activation groups, refer to *ILE Concepts*.

Managing Dynamically-Allocated Storage

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

You are not required to explicitly manage run-time storage. However, you may wish to do so if you want to make use of dynamically allocated run-time storage.

For example, you may want to do this if you do not know exactly how big an array or multiple-occurrence data structure should be. You could define the array or data structure as BASED, and acquire the actual storage for the array or data structure once your program determines how big it should be.

There are two types of heaps available on the system: a default heap and a user-created heap. The rest of this section explains how to use a default heap to manage run-time storage in an ILE RPG/400 program. For more information on user-created heaps and other ILE storage management concepts refer to *ILE Concepts*.

Managing the Default Heap

The first request for dynamic storage within an activation group results in the creation of a **default heap** from which the storage allocation takes place. Additional requests for dynamic storage are met by further allocations from the default heap. If there is insufficient storage in the heap to satisfy the current request for dynamic storage, the heap is extended and the additional storage is allocated.

Allocated dynamic storage remains allocated until it is explicitly freed or until the heap is discarded. The default heap is discarded only when the owning activation group ends.

Programs in the same activation group all use the same default heap. If one program accesses storage beyond what has be allocated, it can cause problems for another program. For example, assume that two programs, PGM A and PGM B are running in the same activation group. 10 bytes are allocated for PGM A, but 11 bytes are changed by PGM A. If the extra byte was in fact allocated for PGM B, problems may arise for PGM B.

You can isolate the dynamic storage used by some programs and procedures within an activation group. You do this by creating one or more user-created heaps. For information on creating a user-created heap refer to *ILE Concepts*.

You can use the following ILE bindable APIs on the default heap:

The Free Storage (CEEFRST) bindable API frees one previous allocation of heap storage.

The Get Heap Storage (CEEGTST) bindable API allocates storage within a heap.

The Reallocate Storage (CEECZST) bindable API changes the size of previously allocated storage.

Note: You cannot use these or any other ILE bindable APIs from within a program created with DFTACTGRP(*YES). This is because static binding (via the CALLB operation) is not allowed in this type of program.

See the System API Reference for specific information about the storage management bindable APIs.

Dynamically Allocating Storage for a Run-Time Array

The following example shows you how to manage dynamic storage belonging to the default heap from an ILE RPG/400 procedure. In this example, the procedure DYNARRAY dynamically allocates storage for a practically unbounded packed array. The caller of the procedure can request one of three actions from DYNARRAY on each call:

- Add an element to the array
- Return an element from the array
- Release the storage for the array.

DYNARRAY performs these actions using the three ILE bindable storage APIs, CEEGTST (Get Storage), CEECZST (Reallocate Storage), and CEEFRST (Free Storage).

Figure 35 shows the Definition specifications for DYNARRAY. The procedure has been defined for use with a (15,0) packed decimal array. It could easily be converted to handle a character array simply by changing the definition of Element to a character field.

```
 *=================================================================*
 * DYNARRAY :     Allocate a (practically) unbounded run-time      *
 *                Packed(15,0) array.  This procedure will         *
 *                Allocate the array, Return an Array value        *
 *                and Deallocate the array.                        *
 *=================================================================*

 *-----------------------------------------------------------------*
 * Procedure parameters                                            *
 * 1. Input: The function required.  Possible values               *
 *        1  meaning add the element to the array.                 *
 *        2  meaning return the element from the array             *
 *        3  meaning to release the storage for the array          *
 * 2. Input/Output: The Element to be added to or returned         *
 *        from the array.                                          *
 * 3. Input: The Index of element to be added or returned          *
 *-----------------------------------------------------------------*
D Operation       S              1P 0
D Element         S             15P 0
D Index           S              5P 0

 *-----------------------------------------------------------------*
 * Named constant definitions of the operation that this           *
 * procedure supports.                                             *
 *-----------------------------------------------------------------*
D AddToArr        C                   1
D ReturnElem      C                   2
D Terminate       C                   3
```

Figure 35 (Part 1 of 2). Definition specifications for DYNARRAY

```
 *-----------------------------------------------------------------*
 * Define variables to interface to the storage management        *
 * API's.                                                          *
 *  1) HeapId = Id of the heap.  We will allocate from the         *
 *             default heap and thus set this variable to 0.       *
 *  2) Size   = Number of bytes to allocate or reallocate          *
 *  3) RetAddr= Address of the storage in the heap.                *
 *-----------------------------------------------------------------*
D HeapId          S              9B 0 INZ(0)
D Size            S              9B 0
D RetAddr         S               *

 *-----------------------------------------------------------------*
 * Define the dynamic array.  We code the number of elements       *
 * as the maximum allowed, noting that no storage will actually    *
 * be declared for this definition (because it is BASED).          *
 *-----------------------------------------------------------------*
D DynArr          S                   DIM(32767) BASED(DynArr@)
D                                     LIKE(Element)
D DynArr@         S               *

 *-----------------------------------------------------------------*
 * Global to keep track of the current number of elements          *
 * in the dynamic array.                                           *
 *-----------------------------------------------------------------*
D NumElems        S              9B 0 INZ(0)

 *-----------------------------------------------------------------*
 * Initial number of elements that will be allocated for the       *
 * array, and minimum number of elements that will be added        *
 * to the array on subsequent allocations.                         *
 *-----------------------------------------------------------------*
D InitAlloc       C                   100
D SubsAlloc       C                   100
```

Figure 35 (Part 2 of 2). Definition specifications for DYNARRAY

Figure 36 on page 86 shows the Calculation specifications which define the entry parameter list for DYNARRAY and the parameter lists for the called APIs.

```
 *-----------------------------------------------------------------*
 * Procedure parameters                                            *
 *-----------------------------------------------------------------*
C     *ENTRY        PLIST
C                   PARM                    Operation
C                   PARM                    Element
C                   PARM                    Index

 *-----------------------------------------------------------------*
 * Interface to the CEEGTST API (Get heap Storage).                *
 *  1) HeapId = Id of the heap.  We will allocate from the         *
 *              default heap and thus set this variable to 0.      *
 *  2) Size   = Number of bytes to allocate                        *
 *  3) RetAddr= Return address of the allocated storage            *
 *  4) *OMIT  = The feedback parameter.  Specifying *OMIT here     *
 *              means that we will receive an exception from       *
 *              the API if it cannot satisfy our request.          *
 *              Since we do not monitor for it, the calling        *
 *              procedure will receive the exception.              *
 *-----------------------------------------------------------------*
C     CEEGTST_PL    PLIST
C                   PARM                    HeapId
C                   PARM                    Size
C                   PARM                    RetAddr
C                   PARM                    *OMIT

 *-----------------------------------------------------------------*
 * Interface to the CEECZST API (Reallocate Storage).              *
 *  1) RetAddr= The address of the storage that we want            *
 *              reallocated.                                       *
 *  2) Size   = The new size of the storage                        *
 *  3) *OMIT  = The feedback parameter.                            *
 *-----------------------------------------------------------------*
C     CEECZST_PL    PLIST
C                   PARM                    RetAddr
C                   PARM                    Size
C                   PARM                    *OMIT

 *-----------------------------------------------------------------*
 * Interface to the CEEFRST API (Free Storage).                    *
 *  1) RetAddr =The address of the storage that we want            *
 *              freed.                                             *
 *  2) *OMIT  = The feedback parameter.  Specifying *OMIT here     *
 *              means that we will receive an exception from       *
 *              the API if it cannot satisfy our request.          *
 *-----------------------------------------------------------------*
C     CEEFRST_PL    PLIST
C                   PARM                    RetAddr
C                   PARM                    *OMIT
```

Figure 36. Parameter Lists for DYNARRAY

The basic logic of the procedure is the following:

1. Run the initialization subroutine which in turn calls the subroutine ALLOCATE. This subroutine allocates heap storage based on initial value of the array (in this case 100) by calling the ILE bindable API CEEGTST (Get Heap Storage).

2. Determine which operation to perform (add or return an element of the array, or release storage).

   - Before adding an element to the array, the procedure checks to see if there is sufficient heap storage. If not, it calls a subroutine REALLOC which

acquires additional storage using the ILE bindable API CEECZST (Reallocate Storage).
- If a return is requested, the procedure returns to the caller either the element requested, or zeros. The latter occurs if the requested element has not actually been stored in the array.
- If a release of storage is requested, the procedure calls the subroutine DEALLOC which in turn calls the ILE bindable API CEEFRST (Free Heap Storage) and then returns to the caller with LR set on.

3. Return

Figure 37 shows the Calculation specifications which define the main logic, and Figure 38 on page 88 shows the Calculation specifications containing the subroutines.

```
     *-----------------------------------------------------------------*
     * Select which operation to perform                               *
     *-----------------------------------------------------------------*
     C                   SELECT

     *-----------------------------------------------------------------*
     * If the user selects to add to the array, then first check       *
     * if the array is large enough, if not then increase its          *
     * size. Add the element.                                          *
     *-----------------------------------------------------------------*

     C     Operation     WHENEQ    AddToArr
     C     Index         IFGT      NumElems
     C                   EXSR      REALLOC
     C                   ENDIF
     C                   EVAL      DynArr(Index) = Element

     *-----------------------------------------------------------------*
     * If the user selects to be returned an element from the          *
     * array then first check if the array is large enough             *
     * to satisfy the request.  If not, then simply clear the          *
     * element.  If the array is large enough then return the          *
     * element.                                                        *
     *-----------------------------------------------------------------*
     C     Operation     WHENEQ    ReturnElem
     C     Index         IFGT      NumElems
     C                   CLEAR                   Element
     C                   ELSE
     C                   EVAL      Element = DynArr(Index)
     C                   ENDIF
```

Figure 37 (Part 1 of 2). Main logic Portion of DYNARRAY

```
 *---------------------------------------------------------------*
 * If the user selects to terminate, then deallocate the         *
 * storage for the array and set on LR to terminate this         *
 * procedure.                                                    *
 *---------------------------------------------------------------*
C     Operation     WHENEQ    Terminate
C                   EXSR      DEALLOC
C                   MOVE      *ON           *INLR

C                   ENDSL

 *---------------------------------------------------------------*
 *   Done.                                                       *
 *---------------------------------------------------------------*
C                   RETURN
```

Figure 37 (Part 2 of 2). Main logic Portion of DYNARRAY

```
 *---------------------------------------------------------------*
 *                 S U B R O U T I N E S                         *
 *---------------------------------------------------------------*

 *===============================================================*
 * *INZSR: The Initialization Subroutine                         *
 *                                                               *
 *        Function: ALLOCATE an initial amount of                *
 *                  storage for the run time array.              *
 *===============================================================*
C     *INZSR        BEGSR

C                   Z-ADD     InitAlloc     NumElems
C                   EXSR      ALLOCATE

C                   ENDSR

 *===============================================================*
 * ALLOCATE: Allocate storage for the dynamic array              *
 *                                                               *
 *        Function: Allocate storage for the DynArr              *
 *                  array using the CEEGTST API.                 *
 *===============================================================*
C     ALLOCATE      BEGSR
 *                                                               *
 * Determine the number of bytes needed for the array.           *
 *                                                               *
C                   EVAL      Size = NumElems * %SIZE(DynArr)
```

Figure 38 (Part 1 of 3). Subroutines for DYNARRAY

```
 *                                                              *
 * Allocate the storage and set the array basing pointer        *
 * to the pointer returned from the API.                        *
 *                                                              *
C                   CALLB     'CEEGTST'     CEEGTST_PL
C                   MOVE      RetAddr       DynArr@
 *                                                              *
 * Initialize the storage for the array.                        *
 *                                                              *
C     1             DO        NumElems      i                 5 0
C                   CLEAR                   DynArr(i)
C                   ENDDO

C                   ENDSR

 *==============================================================*
 * REALLOC: Reallocate storage                                  *
 *                                                              *
 *        Function: Increase the size of the dynamic array      *
 *                  and initialize the new elements.            *
 *==============================================================*
C     REALLOC       BEGSR
 *
 *  Remember the old number of elements
 *
C                   Z-ADD     NumElems      OldElems          5 0
 *
 *  Calculate the new number of elements.  If the Index is
 *  greater than the current number of elements in the array
 *  plus the new allocation, then allocate up to the index,
 *  otherwise, add a new allocation amount onto the array.
 *
C                   IF        Index > NumElems + SubsAlloc
C                   Z-ADD     Index         NumElems
C                   ELSE
C                   ADD       SubsAlloc     NumElems
C                   ENDIF
 *                                                              *
 * Calculate the new size of the array                          *
 *                                                              *
C                   EVAL      Size =  NumElems * %size(DynArr)
 *                                                              *
 * Reallocate the storage and set the array basing pointer      *
 * to the pointer returned from the API.                        *
 *                                                              *
C                   CALLB     'CEECZST'     CEECZST_PL
C                   MOVE      RetAddr       DynArr@
```

Figure 38 (Part 2 of 3). Subroutines for DYNARRAY

```
 *                                                                *
 * Initialize the new elements for the array.                     *
 *                                                                *
C     1             ADD       OldElems      i
C     i             DO        NumElems      i
C                   CLEAR                   DynArr(i)
C                   ENDDO

C                   ENDSR

 *================================================================*
 * DEALLOC: Deallocate the storage for the array                  *
 *                                                                *
 *         Function: Relase the storage for the array.            *
 *================================================================*
C     DEALLOC       BEGSR

C                   CALLB     'CEEFRST'     CEEFRST_PL

C                   ENDSR
```

Figure 38 (Part 3 of 3). Subroutines for DYNARRAY

To create the procedure DYNARRAY, type:

```
CRTRPGMOD MODULE(MYLIB/DYNARRAY) SRCFILE(MYLIB/QRPGLESRC)
```

The procedure can then be bound with other modules using CRTPGM or CRTSRVPGM.

Chapter 9. Calling Programs and Procedures

In ILE, it is possible to call either a program or procedure. The caller must identify whether the target of the call statement is a program or a procedure. ILE RPG/400 provides the following operation codes to enable program or procedure calls and passing of parameters.

Operation Code	**Function**
CALL	Call a program object
CALLB	Call a bound procedure
RETURN	Return to the calling program or procedure
PLIST	Identify a parameter list
PARM	Identify the parameters

This chapter describes how to:

- Call a program or procedure
- Pass parameters between programs and procedures
- Return from a program or procedure
- Use ILE bindable APIs
- Call a Graphics routine
- Call special routines.

Program/Procedure Call Overview

Program processing within ILE occurs at the procedure level. (ILE programs consist of one or more modules which in turn consist of one or more procedures. An ILE RPG/400 module contains only one procedure; other ILE languages may allow more than one.) A 'program call' is a special form of procedure call; that is, it is a call to the program entry procedure. A program entry procedure is the procedure designated at program creation time to receive control when a program is called.

This section compares program call with procedure call. It also presents the notion of the call stack, in order to help you understand how a series of calls interact.

Calling Programs

You can call OPM or ILE programs using program calls. A **program call** is a call made to a program object (*PGM). The called program's name is resolved to an address at run time, just before the calling program passes control to the called program for the first time. For this reason, program calls are often referred to as dynamic calls.

Calls to an ILE program, an EPM program or an OPM program are all examples of program calls. A call to a non-bindable API is also an example of a program call.

You use the CALL operation to make a program call.

When an ILE program is called, the program entry procedure receives the program parameters and is given initial control for the program. In addition, all procedures within the program become available for procedure calls.

Calling Procedures

Unlike OPM programs, ILE programs are not limited to using program calls. ILE programs can also use static procedure calls or procedure pointer calls to call other procedures. Procedure calls are also referred to as bound calls.

A **static procedure call** is a call to an ILE procedure where the name of the procedure is resolved to an address during binding — hence, the term static. As a result, run-time performance of using static procedure calls is faster than run-time performance using program calls. Static calls allow operational descriptors, omitted parameters, and they extend the limit (to 399) on the number of parameters passed.

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

Using either type of procedure call, you can call a procedure in a separate module within the same ILE program or service program, or a procedure in a separate ILE service program. Any procedure that can be called using a static procedure call can also be called through a procedure pointer.

You use the CALLB operation to make a procedure call.

The Call Stack

The **call stack** is a list of call stack entries, in a last-in-first-out (LIFO) order. A **call stack entry** is a call (CALL or CALLB) to a program or procedure. There is one call stack per job.

When an ILE program is called, the program entry procedure is first added to the call stack. The system then automatically performs a procedure call, and the associated user's procedure is added. When a procedure is called, only the user's procedure is added; there is no overhead of a program entry procedure.

Figure 39 on page 93 shows a call stack for an application consisting of an OPM program which calls an ILE program consisting of two modules, an RPG module containing the program entry procedure and the associated user entry procedure, and a C module containing a regular procedure. Note that in the diagrams in this book, the most recent entry is at the bottom of the stack.

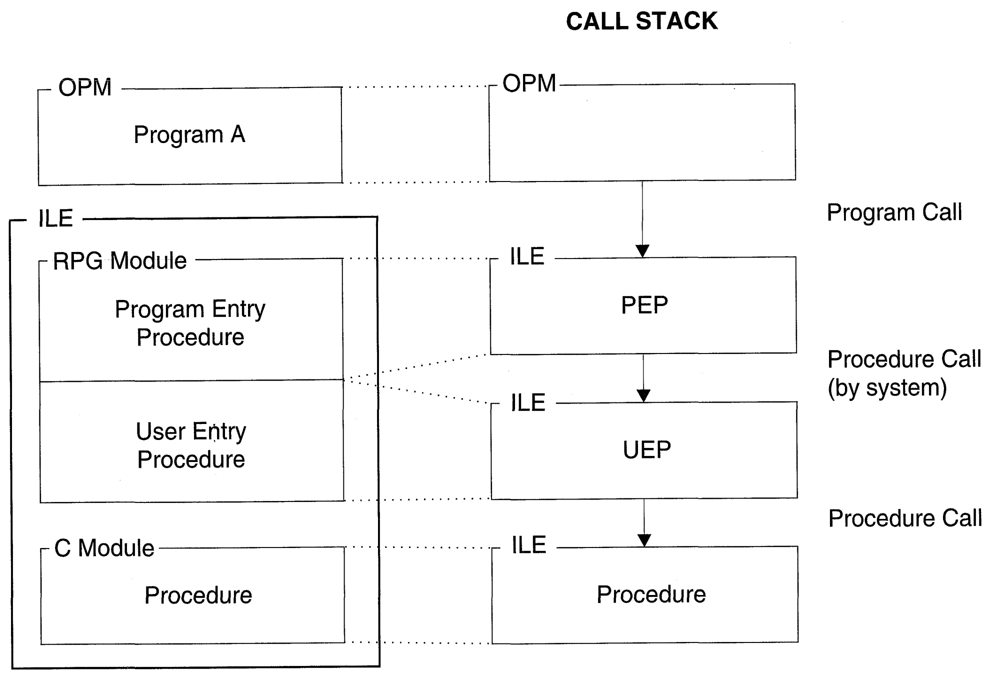


Figure 39. Program and Procedure Calls on the Call Stack

Note: In a program call, the calls to the program entry procedure and the user entry procedure (UEP) occur together, since the call to the UEP is automatic. Therefore, from now on, the two steps of a program call will be combined in later diagrams involving the call stack in this and remaining chapters,

An important point to understand about the call stack is that an ILE RPG/400 procedure which is on the call stack cannot be called until it returns to its caller. Therefore, be careful not to call another procedure which might call an already active ILE RPG/400 procedure.

For example, assume that procedures A, B and C are in the same program. If procedure A calls procedure B, then procedure B can call neither procedure A nor B. If procedure B returns (with or without LR set on) and if procedure A then calls procedure C, procedure C can call procedure B but not procedure A or C. See Figure 40.

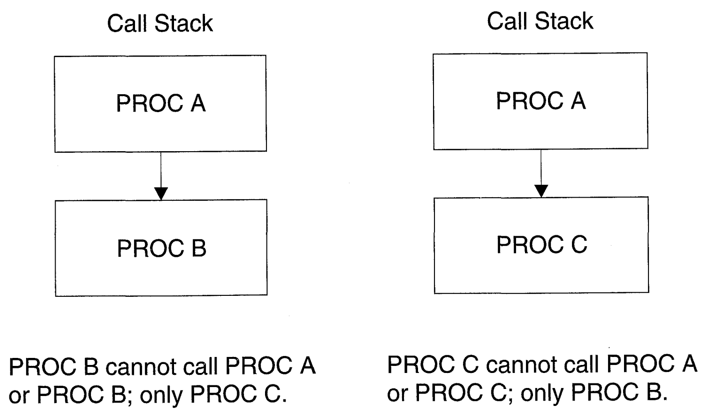

Call Stack
PROC A
PROC B
PROC B cannot call PROC A or PROC B; only PROC C.
Call Stack
PROC A
PROC C
PROC C cannot call PROC A or PROC C; only PROC B.

Figure 40. Procedure Cannot Call Other Active RPG Procedures

Similarly, OPM RPG programs already on the call stack cannot be called.

Using the CALL or CALLB Operations

You use the CALL (Call a Program) operation to make a program call and the CALLB (Call a Bound Procedure) operation to make a procedure call. The two call operations are very similar in their syntax and their use. To call a program or procedure, follow these general steps:

1. Identify the object to be called in the Factor 2 entry.
2. Optionally code an error indicator (positions 73 and 74) and/or an LR indicator (positions 75 and 76).

   When a called object ends in error the error indicator, if specified, is set on. Similarly, if the called object returns with LR on, the LR indicator, if specified, is set on.
3. To pass parameters to the called object, either specify a PLIST in the Result field of the call operation or follow the call operation immediately by PARM operations.

Either operation transfers control from the calling to the called object. After the called object is run, control returns to the first operation that can be processed after the call operation in the calling program or procedure.

The following considerations apply to either call operation:

- The Factor 2 entry can be a variable, literal, or named constant. Note that the entry is case-sensitive.

  For CALL only: The Factor 2 entry can be *library name/program name*, for example, MYLIB/PGM1. If no library name is specified, then the library list is used to find the program. The name of the called program can be provided at run time by specifying a character variable in the Factor 2 entry.

  For CALLB only: To make a procedure pointer call you specify the name of the procedure pointer which contains the address of the procedure to be called.
- A procedure can contain multiple calls to the same object with the same or different PLISTs specified.
- When an ILE RPG/400 procedure (including a program entry procedure) is first called, the fields are initialized and the procedure is given control. On subsequent calls to the same procedure, if it did not end on the previous call, then all fields, indicators, and files in the called procedure are the same as they were when it returned on the preceding call.
- The system records the names of all programs called from within an RPG procedure. When an RPG procedure is bound into a program (*PGM) you can query these names using DSPPGMREF, although you cannot tell which procedure is doing the call.

  For a module, you can query the names of procedures called using DSPMOD DETAIL(*IMPORT). Some procedures on this list will be system procedures; the names of these will usually contain underscores and you do not have to be concerned with these.

  If you call an object using a variable, you will see an entry with the name *VARIABLE (and no library name).

- **For CALLB only:** The compiler creates an operational descriptor indicating the number of parameters passed on the CALLB operation and places this value in the *PARMS field of the called procedure's program status data structure. This number includes any parameters which are designated as omitted (*OMIT on the PARM operation).

  If the (D) operation extender is used with the CALLB operation the compiler also creates an operational descriptor for each field and subfield.

  For more information on operational descriptors, see "Using Operational Descriptors" on page 97.

- There are further restrictions that apply when using the CALL or CALLB operation codes. For a detailed description of these restrictions, see the *ILE RPG/400 Reference*.

Examples of the Call Operations

For examples of using the CALL operation, see:

- "Sample Source for Debug Examples" on page 147, for example of calling an RPG program.
- "Using the *PARMS field in the PSDS" on page 100, for an example of calling an RPG program.

For examples of using the CALLB operation, see:

- "Sample Service Program" on page 65, for an example of calling a procedure in a service program.
- "Dynamically Allocating Storage for a Run-Time Array" on page 84, for an example of calling bindable APIs.
- "Sample Source for Debug Examples" on page 147, for example of calling a C procedure.
- "CUSMAIN: RPG Source" on page 272, for an example of a main inquiry program calling various RPG procedures.

Passing Parameters

ILE RPG/400 provides the PLIST and PARM operation codes to identify fields to be passed and received on call operations.

Keep in mind the following, especially when passing parameters to a different HLL than ILE RPG/400:

- Parameter Passing style of the HLLs

  Each HLL has its way of passing parameters, that is, whether it passes a pointer to the parameter value (as in ILE RPG/400), a copy of the value, or the value itself. For more information on these styles see "ILE Interlanguage Calls" on page 106.

- Interlanguage Data Compatibility

  Different HLLs support different ways of representing data. In general, you should only pass data which have a data type common to the calling and called program or procedure. However, sometimes you may not be sure of the exact format of the data that is being passed to you. In this case you may specify to the users of your procedure that an operational descriptor should be passed to provide additional information regarding the format of the passed parameters. To find out how, see "Using Operational Descriptors" on page 97.

- Number of parameters.

  In general, you should pass the same number of parameters as expected by the called program or procedure. If you pass fewer parameters than expected, and the called object references one for which no data was passed, then it will get an error. For information on omitting parameters when calling procedures, see "Omitting Parameters" on page 98.

  The ILE RPG/400 compiler always passes an operational descriptor describing the number of parameters passed to a called procedure. You can use this descriptor to verify that the same number are being used by both the called and calling procedures. See "Using Operational Descriptors" on page 97 for more information.

  Note: Other ILE languages may not pass operational descriptors as described above.

Using the PLIST Operation

The PLIST operation:

- Defines a name by which a list of parameters can be referenced. The list of parameters is specified by PARM operations immediately following the PLIST operation.
- Defines the entry parameter list (*ENTRY PLIST).

Factor 1 of the PLIST operation must contain the PLIST name. This name can be specified in the Result field of one or more call operations. If the parameter list is the entry parameter list of a called procedure, then Factor 1 must contain *ENTRY.

Multiple PLIST operations can appear in a procedure. However, only one *ENTRY PLIST can be specified.

For an example of the PLIST operation see Figure 30 on page 74, and Figure 42 on page 101.

Using the PARM operation

The PARM operation is used to identify the parameters which are passed from or received by a procedure. Each parameter is defined in a separate PARM operation. You specify the name of the parameter in the Result field; the name need not be the same as in the calling/called procedure.

The Factor 1 and Factor 2 entries are optional and indicate variables or literals whose value is transferred to or received from the Result Field entry depending on whether these entries are in the calling program/procedure or the called program/procedure. Table 7 on page 97 shows how Factor 1 and Factor 2 are used.

Table 7. Meaning of Factor 1 and Factor 2 Entries in PARM Operation

Status	Factor 1	Factor 2
In **calling** procedure	Value transferred from Result Field entry upon return.	Value placed in Result Field entry when call occurs.
In **called** procedure	Value transferred from Result Field entry when call occurs.	Value placed in Result Field entry upon return.

Note: The moves to either the Factor 1 entry or the Result Field entry occur only when the called procedure returns normally to its caller. If an error occurs while attempting to move data to either entry, then the move is not completed.

If insufficient parameters are specified when calling a procedure, an error occurs when an unresolved parameter is used by the called procedure. To avoid the error, you can specify *OMIT in the Result field of the PARM operations of the unpassed parameters. The called procedure can then check to see if the parameter has been omitted by checking to see if the parameter has value of *NULL. For more information, refer to “Omitting Parameters” on page 98.

Keep in mind the following when specifying a PARM operation:

- One or more PARM operations must immediately follow a PLIST operation.
- One or more PARM operations can immediately follow a CALL or CALLB operation.
- When a multiple occurrence data structure is specified in the Result field of a PARM operation, all occurrences of the data structure are passed as a single field.
- Factor 1 and the Result field of a PARM operation cannot contain a literal, a look-ahead field, a named constant, or a user-date reserved word.
- Factor 1 and Factor 2 must be blank if the Result field contains the name of a multiple occurrence data structure or if it contains *OMIT.
- There are other restrictions that apply when using the PARM operation code. For a detailed description of these restrictions, see the *ILE RPG/400 Reference*.

For examples of the PARM operation see:

- Figure 30 on page 74
- Figure 23 on page 52
- Figure 43 on page 104
- Figure 26 on page 66

Using Operational Descriptors

Sometimes it is necessary to pass a parameter to a procedure even though the data type is not precisely known to the called procedure, (for example, different types of strings). In these instances you can use **operational descriptors** to provide descriptive information to the called procedure regarding the form of the parameter. The additional information allows the procedure to properly interpret the string. You should only use operational descriptors when they are expected by the called procedure, usually an ILE bindable API.

Note: Initially, the ILE RPG/400 compiler only supports operational descriptors for fields and subfields. In other words, operational descriptors are not available for data structures, arrays, or tables.

To use operational descriptors, you specify (D) as the operation code extender of the CALLB operation. Operational descriptors are then built by the calling procedure and passed as hidden parameters to the called procedure. You can specify (D) even if the CALLB operation references omitted parameters. Operational descriptors will not be built for omitted parameters.

Operational descriptors have no effect on the parameters being passed or in the way that they are passed. When a procedure is passed operational descriptors which it does not expect, the operational descriptors are simply ignored.

You can retrieve information from an operational descriptor using the ILE bindable APIs Retrieve Operational Descriptor Information (CEEDOD) and Get Descriptive Information About a String Argument (CEESGI).

Note that operational descriptors are only allowed for procedure level calls. An error message will be issued by the compiler if the 'D' operation code extender is specified on a CALL operation.

For an example of using operational descriptors, see “Sample Service Program” on page 65. The example consists of a service program which converts character strings which are passed to it to their hexadecimal equivalent. The service program uses operational descriptors to determine the length of the character string and the length to be converted.

Omitting Parameters

When calling a procedure, you may sometimes need to pass fewer parameters than the called procedure is expecting. For example, this situation might arise when you are calling the ILE bindable APIs.

To indicate that a parameter is being omitted, specify *OMIT on the PARM operation. When *OMIT is specified, the compiler will generate the necessary code to indicate to the called procedure that the parameter has been omitted.

Note: The language of the called procedure must support passing by reference. Otherwise, the results are not predictable.

An omitted parameter cannot be referenced by the called procedure. If it is, an error will occur. The value of specifying *OMIT, is that it allows you to write a called procedure which can checks to see if a parameter has been omitted, and if it has, avoid referencing it.

To check to see if a parameter has been omitted in an ILE RPG/400 procedure, use the %ADDR built-in function to check the address of the parameter you wish to check. If the address is *NULL, then the parameter has been omitted. You can also use the CEESTA (Check for Omitted Argument) bindable API.

Keep in mind the following when specifying *OMIT:

- *OMIT is only allowed in PARM operations that immediately follows a CALLB operation or in a PLIST used with a CALLB.

- Factor 1 and Factor 2 of a PARM operation must be blank, if *OMIT is specified.
- *OMIT is not allowed in a PARM operation that is part of a *ENTRY PLIST.

The following is a simple example of how *OMIT can be used. In this example, a procedure calls the ILE bindable API CEEDOD in order to decompose an operational descriptor. The CEEDOD API expects to receive seven parameters; yet only six have been defined in the calling procedure. The last parameter of CEEDOD (and of most bindable APIs) is the feedback code which can be used to determine how the API ended. However, the calling procedure has been designed to receive any error messages via an exception rather than this feedback code. Consequently, on the call to CEEDOD, the procedure must indicate that the parameter for the feedback code has been omitted.

Figure 41 shows the coding for this situation.

```
*=================================================================*
* The following procedure calls an ILE API to decompose an        *
* operational descriptor.  It wishes to receive any error         *
* messages via exception. It does this by omitting the            *
* feedback code (the last parameter of the CEEDOD API).           *
*=================================================================*

*-----------------------------------------------------------------*
* CEEDOD parameters                                               *
*-----------------------------------------------------------------*

DName+++++++++++ETDsFrom+++To/L+++IDc.Functions++++++++++++++++++++
D posn            S              9B 0
D desctype        S              9B 0
D datatype        S              9B 0
D descinf1        S              9B 0
D descinf2        S              9B 0
D datalen         S              9B 0
```

*Figure 41 (Part 1 of 2). Omitting a parameter using the *OMIT keyword*

```
*-----------------------------------------------------------------*
* Call CEEDOD to decompose an operational descriptor, omitting the *
* feedback code.                                                  *
*-----------------------------------------------------------------*
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq
C                   EVAL      posn=1

C                   CALLB     CEEDOD
C                   PARM                    posn
C                   PARM                    desctype
C                   PARM                    datatype
C                   PARM                    descinf1
C                   PARM                    descinf2
C                   PARM                    datalen
C                   PARM                    *OMIT

C                   MOVE     '1'            *INLR
```

*Figure 41 (Part 2 of 2). Omitting a parameter using the *OMIT keyword*

A similar example of using *OMIT is provided earlier in this book in the example on creating a service program. See "Sample Service Program" on page 65 for the specific source.

Checking for the Number of Passed Parameters

You can use the *PARMS field in the program status data structure (PSDS) to determine the number of parameters passed to a called procedure. This number includes any parameters which are designated as omitted (*OMIT on the PARM operation).

Depending on how the procedure has been written, this number may allow you to avoid references to parameters that are not passed. For example, supposing you want to write a procedure which will sometimes be passed three parameters and sometimes four parameters. You can write the called procedure to process either number depending on the value in the *PARMS field.

This might arise when a new field is required. New procedures which use the field will pass a value for it. Old procedures can remain unchanged.

Because the number contained in the *PARMS field includes parameters passed using *OMIT, it is recommended that you check for *NULL values on parameters passed using the CALLB operation. For more information on using *OMIT, see "Omitting Parameters" on page 98.

Using the *PARMS field in the PSDS

In this example, a program FMTADDR has been modified several times to allow for a change in the address information for the employees of a company. FMTADDR is called by three different programs. The programs differ only in the number of parameters they use to process the employee information. That is, new requirements for the FMTADDR have arisen, and to support them, new parameters have been added. However, old programs calling FMTADDR are still supported and do not have to be changed or recompiled.

The changes to the employee address can be summarized as follows:

- Initially only the street name and number were required because all employees lived in the same city. Thus, the city and province could be supplied by default.
- At a later point, the company expanded, and so the city information became variable for some company-wide applications.
- Further expansion resulted in variable province information.

The program processes the information based on the number of parameters passed. The number may vary from 3 to 5. The number tells the program whether to provide default city and/or province values. Figure 42 shows the source for this program.

The main logic of the FMTADDR program is as follows:

1. Check to see how many parameters were passed by testing the value of NumofParms. This field is defined based on the *PARMS field in the program's PSDS.
   - If the number is greater than 4, then the default province is replaced with the actual province supplied by the fifth parameter P_Province.
   - If the number is greater than 3, then the default city is replaced with the actual city supplied by the fourth parameter P_City.
2. Correct the street number for printing using the subroutine GetStreet#.
3. Concatenate the complete address.
4. Return.

```
*=================================================================*
* FMTADDR - format an address                                     *
*                                                                 *
* Entry parameters                                                *
* 1. Address       character(70)                                  *
* 2. Street number packed(5,0)                                    *
* 3. Street name   character(20)                                  *
* 4. City          character(15)   (some callers do not pass)     *
* 5. Province      character(15)   (some callers do not pass)     *
*=================================================================*

D Address         S             70
D Street#         S              5  0
D Street          S             20
D P_City          S             15
D P_Province      S             15
```

```
 *-----------------------------------------------------------------*
 * Default values for parameters that might not be passed.         *
 *-----------------------------------------------------------------*
D City            S             15    INZ('Toronto')
D Province        S             15    INZ('Ontario')

 *-----------------------------------------------------------------*
 * Use *PARMS to determine how many parameters were passed         *
 *-----------------------------------------------------------------*
D Psds           SDS
D  NumOfParms        *PARMS

C     *ENTRY        PLIST
C                   PARM                    Address
C                   PARM                    Street#
C                   PARM                    Street
C                   PARM                    P_City
C                   PARM                    P_Province

 *-----------------------------------------------------------------*
 * Check whether the province parameter was passed.  If it was,    *
 * replace the default with the parameter value.
 *-----------------------------------------------------------------*
C                   IF        NumOfParms > 4
C                   EVAL      Province = P_Province
C                   ENDIF

 *-----------------------------------------------------------------*
 * Check whether the city parameter was passed.  If it was,        *
 * replace the default with the parameter value.                   *
 *-----------------------------------------------------------------*
C                   IF        NumOfParms > 3
C                   EVAL      City = P_City
C                   ENDIF

 *-----------------------------------------------------------------*
 * Set 'CStreet#' to be character form of 'Street#'                *
 *-----------------------------------------------------------------*
C                   EXSR      GetStreet#
 *-----------------------------------------------------------------*
 * Format the address as    Number Street, City, Province          *
 *-----------------------------------------------------------------*
C     CStreet#      CAT(P)    Street:1      Address
C                   CAT(P)    ',':0         Address
C                   CAT(P)    City:1        Address
C                   CAT(P)    ',':0         Address
C                   CAT(P)    Province:1    Address

C                   SETON                                        LR
```

```
 *===============================================================*
 * SUBROUTINE: GetStreet#
 * Get the character form of the street number, left-adjusted    *
 * and padded on the right with blanks.                          *
 *===============================================================*
C     GetStreet#    BEGSR

C                   MOVEL     Street#       CStreet#        10
 *---------------------------------------------------------------*
 * Find the first non-zero.                                      *
 *---------------------------------------------------------------*
C     '0'           CHECK     CStreet#      Non0             5 0
 *---------------------------------------------------------------*
 * If there was a non-zero, substring the number starting at     *
 * non-zero.                                                     *
 *---------------------------------------------------------------*
C                   IF        Non0 > 0
C                   SUBST(P)  CStreet#:Non0 CStreet#
 *---------------------------------------------------------------*
 * If there was no non-zero, just use '0' as the street number.  *
 *---------------------------------------------------------------*
C                   ELSE
C                   MOVEL(P)  '0'           CStreet#
C                   ENDIF

C                   ENDSR
```

*Figure 42 (Part 3 of 3). Source for program FMTADDR. Checks *PARMS to determine the number of parameters passed.*

Figure 43 on page 104 shows the source for the program PRTADDR. This program illustrates the use of FMTADDR. For convenience, the three programs which would each call FMTADDR are combined into this one program. Also, for the purposes of example, the data is program-described.

Since PRTADDR is 'three programs-in-one', it must define three different address data structures. Similarly, there are three parts in the Calculation specifications, each one corresponding to programs at each stage. After printing the address, the program PRTADDR returns.

```
 *=================================================================*
 * PRTADDR - Print an address                                      *
 *                                                                 *
 * Entry parameters                                                *
 * 1. Address       character(70)                                  *
 * 2. Street number packed(5,0)                                    *
 * 3. Street name   character(20)                                  *
 * 4. City          character(15)   (some callers do not pass)     *
 * 5. Province      character(15)   (some callers do not pass)     *
 *=================================================================*
FQSYSPRT   O    F  80        PRINTER

 *-----------------------------------------------------------------*
 * Stage1: Original address data structure.                        *
 * Only street and number are variable information.                *
 *-----------------------------------------------------------------*
D Stage1          DS
D   Street#1                      5P 0 DIM(2) CTDATA
D   StreetNam1                   20    DIM(2) ALT(Street#1)

 *-----------------------------------------------------------------*
 * Stage2: Revised address data structure as city information     *
 * now variable.                                                   *
 *-----------------------------------------------------------------*
D Stage2          DS
D   Street#2                      5P 0 DIM(2) CTDATA
D   Addr2                        35    DIM(2) ALT(Street#2)
D     StreetNam2                 20    OVERLAY(Addr2:1)
D     City2                      15    OVERLAY(Addr2:21)

 *-----------------------------------------------------------------*
 * Stage3: Revised address data structure as provincial            *
 * information now variable.                                       *
 *-----------------------------------------------------------------*
D Stage3          DS
D   Street#3                      5P 0 DIM(2) CTDATA
D   Addr3                        50    DIM(2) ALT(Street#3)
D     StreetNam3                 20    OVERLAY(Addr3:1)
D     City3                      15    OVERLAY(Addr3:21)
D     Province3                  15    OVERLAY(Addr3:36)

 *-----------------------------------------------------------------*
 * 'Program 1'- Use of FMTADDR before city parameter was added.    *
 *-----------------------------------------------------------------*
C                   DO        2             X                 5 0
C                   CALL      'FMTADDR'
C                   PARM                    Address          70
C                   PARM                    Street#1(X)
C                   PARM                    StreetNam1(X)

C                   EXCEPT
C                   ENDDO
```

```
     *---------------------------------------------------------------*
     * 'Program 2'- Use of FMTADDR before province parameter was added.*
     *---------------------------------------------------------------*
    C                   DO        2             X                 5 0
    C                   CALL      'FMTADDR'
    C                   PARM                    Address          70
    C                   PARM                    Street#2(X)
    C                   PARM                    StreetNam2(X)
    C                   PARM                    City2(X)

    C                   EXCEPT
    C                   ENDDO

     *---------------------------------------------------------------*
     * 'Program 3' - Use of FMTADDR after province parameter was added.*
     *---------------------------------------------------------------*
    C                   DO        2             X                 5 0
    C                   CALL      'FMTADDR'
    C                   PARM                    Address          70
    C                   PARM                    Street#3(X)
    C                   PARM                    StreetNam3(X)
    C                   PARM                    City3(X)
    C                   PARM                    Province3(X)

    C                   EXCEPT
    C                   ENDDO

    C                   SETON                                        LR

     *---------------------------------------------------------------*
     * Print the address.                                            *
     *---------------------------------------------------------------*
    OQSYSPRT   E
    O                       Address
**
00123Bumble Bee Drive
01243Hummingbird Lane
**
00003Cowslip Street      Toronto
01150Eglinton Avenue     North York
**
00012Jasper Avenue       Edmonton       Alberta
00027Avenue Road         Sudbury        Ontario
```

Figure 43 (Part 2 of 2). Source for program PRTADDR. Calls the program FMTADDR

To create these programs, follow these steps:

1. To create FMTADDR, using the source in Figure 42 on page 101, type:

   ```
   CRTBNDRPG PGM(MYLIB/FMTADDR)
   ```

2. To create PRTADDR, using the source in Figure 43 on page 104, type:

   ```
   CRTBNDRPG PGM(MYLIB/PRTADDR)
   ```

3. Call PRTADDR. The output is shown below:

   ```
   123 Bumble Bee Drive, Toronto, Ontario
   1243 Hummingbird Lane, Toronto, Ontario
   3 Cowslip Street, Toronto, Ontario
   1150 Eglinton Avenue, North York, Ontario
   12 Jasper Avenue, Edmonton, Alberta
   27 Avenue Road, Sudbury, Ontario
   ```

ILE Interlanguage Calls

ILE RPG/400 uses the same calling mechanisms for calling any ILE HLL program or procedure: CALL and CALLB respectively. Similarly, ILE RPG/400 passes and receives parameters using one passing method: by reference. In other words, ILE RPG/400 passes and receives parameters via a pointer to the actual data object. Other ILE languages may have different methods of passing data. Table 8 shows the common parameter passing methods for the ILE languages.

Table 8. Default Parameter Passing Style for ILE Languages

ILE HLL	Pass Data Object	Receive Data Object
ILE RPG/400	By reference	By reference
ILE C/400	By value, directly or By reference	By value, directly or By reference
ILE COBOL/400	By reference or BY CONTENT	By reference By value, indirectly
ILE CL	By reference	By reference

In short, to pass or receive parameters to or from procedure calls involving other ILE languages, especially ILE C/400 or ILE COBOL/400, you must ensure that the other procedure is set up to accept data by reference.

Returning from a Called Program/Procedure

A return from a procedure (including a program entry procedure) involves the removal of its call stack entry from the call stack along with other closing or cleanup operations.

An ILE RPG/400 procedure returns control to the calling procedure in one of the following ways:

- With a normal end
- With an abnormal end
- Without an end.

A description of the ways to return from a called procedure follows.

For a detailed description of where the LR, H1 through H9, and RT indicators are tested in the RPG program cycle, see the section on the RPG program cycle in the *ILE RPG/400 Reference*.

Normal End

A procedure ends normally and control returns to the calling procedure when the LR indicator is on and the H1 through H9 indicators are not on. The LR indicator can be set on:

- implicitly, as when the last record is processed from a primary or secondary file during the RPG program cycle
- explicitly, as when you set LR on.

A procedure also ends normally if:

- The RETURN operation is processed, the H1 through H9 indicators are not on, and the LR indicator is on.
- The RT indicator is on, the H1 through H9 indicators are not on, and the LR indicator is on.

When a procedure ends normally, the following occurs:

- The Factor-2-to-Result-field move of a PARM operation is performed.
- All arrays and tables with a 'To file name' specified on the Definition specifications, and all locked data area data structures are written out.
- Any data areas locked by the procedure are unlocked.
- All files that are open are closed.
- A return code is set to indicate to the calling procedure that the procedure has ended normally, and control then returns to the calling procedure.

On the next call to the procedure, a fresh copy is available for processing.

Note: If you are accustomed to ending with LR on to cause storage to be released, and you are running in a named (persistent) activation group, you may want to consider returning without an end. The reasons are:

- The storage is not freed until the activation group ends so there is no storage advantage to ending with LR on.
- Call performance is improved if the program is not re-initialized for each call.

You would only want to do this if you did not need your program re-initialized each time.

Abnormal End

A procedure ends abnormally and control returns to the calling procedure when one of the following occurs:

- The cancel option is taken when an ILE RPG/400 error message is issued.
- An ENDSR *CANCL operation in a *PSSR or INFSR error subroutine is processed. (For further information on the *CANCL return point for the *PSSR and INFSR error subroutines, see “Specifying a Return Point in the ENDSR Operation” on page 170).
- An H1 through H9 indicator is on when a RETURN operation is processed.
- An H1 through H9 indicator is on when last record (LR) processing occurs in the RPG cycle.

A procedure also ends abnormally when something outside the procedure ends its invocation. For example, this would occur if an ILE RPG/400 procedure X calls another procedure (such as a CL procedure) that issues an escape message directly to the procedure calling X.

When a procedure ends abnormally, the following occurs:

- All files that are open are closed.
- Any data areas locked by the procedure are unlocked.

- If there is a function check, then it is percolated to the caller. If not, the escape message RNX9001 is issued directly to the caller of of X (unless another escape message instead).

On the next call to the procedure, a fresh copy is available for processing. (For more information on the procedure status data structure, see "Using RPG-Specific Handlers" on page 161.)

Returning without Ending

A procedure can return control to the calling procedure without ending when none of the LR or H1 through H9 indicators are on and one of the following occurs:

- The RETURN operation is processed.
- The RT indicator is on and control reaches the *GETIN part of the RPG cycle, in which case control returns immediately to the calling procedure. (For further information on the RT indicator, see the *ILE RPG/400 Reference*)

If you call a procedure and it returns without ending, when you call the procedure again, all fields, indicators, and files in the procedure will hold the same values they did when you left the procedure.

Note:

- This is not true if the program is running in a *NEW activation group, since the activation group is deleted when the program returns. In that case, the next time you call your program will be the same as if you had ended with LR on.
- If you are sharing files, the state of the file may be different from the state it held when you left the procedure.

You can use either the RETURN operation code or the RT indicator in conjunction with the LR indicator and the H1 through H9 indicators. Be aware of the testing sequence in the RPG program cycle for the RETURN operation, the RT indicator, the LR indicator, and the H1 through H9 indicators.

Returning using ILE Bindable APIs

You can end a procedure normally by using the ILE bindable API CEETREC. However, the API will end *all* call stack entries that are in the same activation group up to the control boundary. When a procedure is ended using CEETREC it follows normal LR processing, as described in "Normal End" on page 106. On the next call to the procedure, a fresh copy is available for processing.

Similarly, you can end a procedure abnormally using the ILE bindable API CEE4ABN. The procedure will end as described in "Abnormal End" on page 107.

Note: You cannot use either of these APIs in a procedure created with DFTACTGRP(*YES), since procedure calls are not allowed in these procedures.

For more information on CEETREC and CEE4ABN refer to the System API Reference.

Using Bindable APIs

Bindable application programming interfaces (APIs) are available to all ILE languages. In some cases they provide additional function beyond that provided by a specific ILE language. They are also useful for mixed-language applications because they are HLL independent.

The bindable APIs provide a wide range of functions including:

- Activation group and control flow management
- Storage management
- Condition management
- Message services
- Source Debugger
- Math functions
- Call management
- Operational descriptor access

You access ILE bindable APIs using the same call mechanisms used by ILE RPG/400 to call procedures, that is, the CALLB operation. Figure 44 shows a sample 'call' to a bindable API.

```
C                   CALLB    'CEExxxx'
C                   PARM                  parm1
C                   PARM                  parm2
                    ...
C                   PARM                  parmn
C                   PARM                  feedback
```

Figure 44. Sample Call Syntax for ILE Bindable APIs

where

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

The following restrictions apply:

- APIs that require a return value cannot be called.
- APIs that require value parameters cannot be used.
- APIs cannot be used if DFTACTGRP(*YES) is specified on the CRTBNDRPG command.

For more information on bindable APIs, refer to the System API Reference.

Examples of Using Bindable APIs

For examples of using bindable APIs, see:

- “Sample Service Program” on page 65, for an example of using CEEDOD
- “Dynamically Allocating Storage for a Run-Time Array” on page 84. for an example of using CEEGTST, CEEFRST, and CEECZST.
- “Using Cancel Handlers” on page 176, for an example of using CEEHDLR and CEEHDLU.

Calling a Graphics Routine

ILE RPG/400 supports the use of the CALL operation to call OS/400 Graphics, which includes the Graphical Data Display Manager (GDDM*, a set of graphics primitives for drawing pictures), and Presentation Graphics Routines (a set of business charting routines). Factor 2 must contain the literal or named constant 'GDDM' (not a field name or array element). Use the PLIST and PARM operations to pass the following parameters:

- The name of the graphics routine you want to run.
- The appropriate parameters for the specified graphics routine. These parameters must be of the data type required by the graphics routine.

The procedure that processes the CALL does not implicitly start or end OS/400 graphics routines.

For more information on OS/400 Graphics, graphics routines and parameters, see *GDDM Programming* and the *GDDM Reference*.

Note: You can call OS/400 Graphics using the CALL operation; you cannot use the CALLB operation. You cannot pass Date, Time, Timestamp, or Graphic fields to GDDM, nor can you pass pointers to it.

Calling Special Routines

ILE RPG/400 supports the use of the following special routines using the CALL and PARM operations:

- Message-retrieving routine (SUBR23R3)
- Moving Bracketed Double-byte Data and Deleting Control Characters (SUBR40R3)
- Moving Bracketed Double-byte Data and Adding Control Characters (SUBR41R3).

Note: You cannot use the CALLB operation to call these special subroutines.

While the message retrieval routine is still supported, it is recommended that you use the QMHRTVM message API, which is more powerful.

Similarly, the routines SUBR40R3 and SUBR41R3 are being continued for compatibility reasons only. They will not be updated to reflect the level of graphic support provided by RPG IV via the new graphic data type.

Debugging and Exception Handling

This section describes how to:

- debug an ILE application using the ILE source debugger
- write programs that handle exceptions
- obtain a dump.

Chapter 10. Debugging Programs

Debugging allows you to detect, diagnose, and eliminate run-time errors in a program. You can debug ILE programs using the ILE source debugger.

This chapter describes how to use the ILE source debugger to:

- Prepare your ILE RPG/400 program for debugging
- Start a debug session
- Add and remove programs from a debug session
- View the program source from a debug session
- Set and remove conditional and unconditional breakpoints
- Step through a program
- Display the value of fields
- Change the value of fields
- Display the attributes of fields
- Equate a shorthand name to a field, expression, or debug command.

While debugging and testing your programs, ensure that your library list is changed to direct the programs to a test library containing test data so that any existing real data is not affected.

You can prevent database files in production libraries from being modified unintentionally by using one of the following commands:

- Use the Start Debug (STRDBG) command and retain the default *NO for the UPDPROD parameter.
- Use the Change Debug (CHGDBG) command.

See the appendix on debugging in *CL Reference* for more information on preventing unintended modification of production files.

See the chapter on debugging in *ILE Concepts*, for more information on the ILE source debugger (including authority required to debug a program or service program and the effects of optimization levels).

The ILE Source Debugger

The ILE source debugger is used to detect errors in and eliminate errors from program objects and service programs. Using debug commands with any ILE program, you can:

- View the program source or change the debug view.
- Set and remove conditional and unconditional breakpoints.
- Step through a specified number of statements.
- Display or change the value of fields, structures, and arrays.
- Equate a shorthand name with a field, expression, or debug command.

Before you can use the source debugger, you must select a debug view when you create a module object or program object using CRTRPGMOD or CRTBNDRPG. After starting the debugger you can set breakpoints and then call the program.

When a program stops because of a breakpoint or a step command, the pertinent module object's view is shown on the display at the point where the program stopped. At this point you can perform other actions such as displaying or changing field values.

Note: If your program has been optimized, you can still display fields, but their values may not be reliable. To ensure that the content of fields or data structures contain their correct (current) values, specify the NOOPT keyword on the appropriate Definition specification. To change the optimization level, see “Changing the Optimization Level” on page 59.

Debug Commands

Many debug commands are available for use with the ILE source debugger. The debug commands and their parameters are entered on the debug command line displayed on the bottom of the Display Module Source and Evaluate Expression displays. These commands can be entered in uppercase, lowercase, or mixed case.

Note: The debug commands entered on the debug command line are not CL commands.

The online help for the ILE source debugger describes the debug commands, explains their allowed abbreviations, and provides syntax diagrams for each command. It also provides examples in each of the ILE languages of displaying and changing variables using the source debugger. You can access the help while in a debug session by pressing F1(Help).

The debug commands are listed below.

Command	Description
ATTR	Permits you to display the attributes of a variable. The attributes are the size and type of the variable as recorded in the debug symbol table.
BREAK	Permits you to enter either an unconditional or conditional breakpoint at a position in the program being tested. Use BREAK *line-number* WHEN *expression* to enter a conditional breakpoint.
CLEAR	Permits you to remove conditional and unconditional breakpoints.
DISPLAY	Allows you to display the names and definitions assigned by using the EQUATE command. It also allows you to display a different source module than the one currently shown on the Display Module Source display. The module object must exist in the current program object.
EQUATE	Allows you to assign an expression, variable, or debug command to a name for shorthand use.
EVAL	Allows you to display or change the value of a variable or to display the value of expressions, records, structures, or arrays.
QUAL	Allows you to define the scope of variables that appear in subsequent EVAL commands. This command applies only to languages, such as ILE C/400, which have local variables. It does not apply to ILE RPG/400.
STEP	Allows you to run one or more statements of the program being debugged.

FIND	Searches forwards or backwards in the module currently displayed for a specified line number or string or text.
UP	Moves the displayed window of source towards the beginning of the view by the amount entered.
DOWN	Moves the displayed window of source towards the end of the view by the amount entered.
LEFT	Moves the displayed window of source to the left
RIGHT	Moves the displayed window of source to the right by the number of characters entered.
TOP	Positions the view to show the first line.
BOTTOM	Positions the view to show the last line.
NEXT	Positions the view to the next breakpoint in the source currently displayed.
PREVIOUS	Positions the view to the previous breakpoint in the source currently displayed.
HELP	Shows the online help information for the available source debugger commands.

Preparing a Program for Debugging

A program or module must have debug data available if you are to debug it. Since debug data is created during compilation, you specify whether a module is to contain debug data when you create it using CRTBNDRPG or CRTRPGMOD. You use the DBGVIEW parameter on either of these commands to indicate what type of data (if any) is to be created during compilation.

The type of debug data that can be associated with a module is referred to as a **debug view**. You can create one of the following views for each module that you want to debug. They are:

- Root source view
- COPY source view
- Listing view
- Statement view.

The default value for both CRTBNDRPG and CRTRPGMOD is to create a statement view. This view provides the closest level of debug support to previous releases.

If you do not want debug data to be included with the module or if you want faster compilation time, specify DBGVIEW(*NONE) when the module is created. However, the DUMP utility is not supported when no debug data is available.

Note also that the storage requirements for a module or program will vary somewhat depending on the type of debug data included with it. The following values for the DBGVIEW parameter are listed in increasing order based on their effect on secondary storage requirements:

1. *NONE
2. *STMT
3. *SOURCE

4. *COPY
5. *LIST
6. *ALL

Once you have created a module with debug data and bound it into a program object (*PGM), you can start to debug your program.

Creating a Root Source View

A **root source view** contains text from the root source member. This view does not contain any /COPY members. Furthermore, it is not available if the root source member is a DDM file.

You create a root source view to debug a module by using the *SOURCE, *COPY or *ALL options on the DBGVIEW parameter for either the CRTRPGMOD or CRTBNDRPG commands when you create the module.

The compiler creates the root source view while the module object (*MODULE) is being compiled. The root source view is created using references to locations of text in the root source member rather than copying the text of the member into the module object. For this reason, you should not modify, rename, or move root source members between the module creation of these members and the debugging of the module created from these members. If you do, the views for these source members may not be usable.

For example, to create a root source view for a program DEBUGEX when using CRTBNDRPG, type:

```
CRTBNDRPG PGM(MYLIB/DEBUGEX) SRCFILE(MYLIB/QRPGLESRC)
          TEXT('ILE RPG/400 program DEBUGEX')
          DBGVIEW(*SOURCE)
```

To create a root source view for a module DBGEX when using CRTRPGMOD, type:

```
CRTRPGMOD MODULE(MYLIB/DBGEX) SRCFILE(MYLIB/QRPGLESRC)
          TEXT('Entry module for program DEBUGEX')
          DBGVIEW(*SOURCE)
```

Specifying DBGVIEW(*SOURCE) with either create command creates a root source view for debugging module DBGEX. By default, a compiler listing with /COPY members and expanded DDS, as well as other additional information is produced.

Creating a COPY Source View

A **COPY source view** contains text from the root source member, as well as the text of all /COPY members expanded into the text of the source. When you use the COPY view, you can debug the root source member of the program using the root source view and the /COPY members of the program using the COPY source view.

The view of the root source member generated by DBGVIEW(*COPY) is the same view generated by DBGVIEW(*SOURCE). As with the root source view, a COPY source view is not available if the source file is a DDM file.

You create a COPY source view to debug a module by using the *COPY or *ALL option on the DBGVIEW parameter.

The compiler creates the COPY view while the module object (*MODULE) is being compiled. The COPY view is created using references to locations of text in the source members (both root source member and /COPY members) rather than copying the text of the members into the view. For this reason, you should not modify, rename, or move source members between the time the module object is created and the debugging of the module created from these members. If you do, the views for these source members may not be usable.

For example, to create a source view of a program TEST1 that contains /COPY members type:

```
CRTBNDRPG PGM(MYLIB/TEST1) SRCFILE(MYLIB/QRPGLESRC)
          TEXT('ILE RPG/400 program TEST1')
          DBGVIEW(*COPY)
```

Specifying DBGVIEW(*COPY) with either create command creates a root source view with /COPY members for debugging module TEST1. By default, a compiler listing is produced. The compiler listing will include /COPY members as well, since OPTION(*SHOWCPY) is a default value.

Creating a Listing View

A **listing view** contains text similar to the text in the compiler listing produced by the ILE RPG/400 compiler. The information contained in the listing view is dependent on whether OPTION(*SHOWCPY) or OPTION(*EXPDDS) are specified for either create command. OPTION(*SHOWCPY) includes /COPY members in the listing and OPTION(*EXPDDS) includes externally-described files.

Note that if you specify indentation in the compiler listing (via the INDENT parameter), the indentation will not appear in the listing view.

You create a listing view to debug a module by using the *LIST or *ALL options on the DBGVIEW parameter for either the CRTRPGMOD or CRTBNDRPG commands when you create a module.

The compiler creates the listing view while the module object (*MODULE) is being generated. The listing view is created by copying the text of the appropriate source members into the module object. There is no dependency on the source members upon which it is based, once the listing view is created.

For example, to create a listing view for a program TEST1 that contains expanded DDS type:

```
CRTBNDRPG PGM(MYLIB/TEST1) SRCFILE(MYLIB/QRPGLESRC)
          SRCMBR(TEST1)  OUTPUT(*PRINT)
          TEXT('ILE RPG/400 program TEST1')
          OPTION(*EXPDDS) DBGVIEW(*LIST)
```

Specifying DBGVIEW(*LIST) for the DBGVIEW parameter and *EXPDDS for the OPTION parameter on either create command creates a listing view with expanded DDS for debugging the source for TEST1. Note that OUTPUT(*PRINT) and OPTION(*EXPDDS) are both default values.

Creating a Statement View

A **statement view** allows the module object to be debugged using debug commands. Since source will not be displayed you must make use of statement numbers which are shown on the left-most column of the source section of the compiler listing. In other words, to effectively use this view, you will need a compiler listing.

You create a statement view to debug a module by using the *STMT option on the DBGVIEW parameter for either the CRTRPGMOD or CRTBNDRPG commands when you create a module.

Use this view when:

- You have storage constraints, but do not want to recompile the module or program if you need to debug it.
- You are sending compiled objects to other users and want to be able to diagnose problems in your code using the debugger, but you do not want these users to see your actual code.

For example, to create a statement view for the program DEBUGEX using CRTBNDRPG, type:

```
CRTBNDRPG PGM(MYLIB/DEBUGEX) SRCFILE(MYLIB/QRPGLESRC)
          TEXT('ILE RPG/400 program DEBUGEX')
```

To create a statement view for a module using CRTRPGMOD, type:

```
CRTRPGMOD MODULE(MYLIB/DBGEX) SRCFILE(MYLIB/QRPGLESRC)
          TEXT('Entry module for program DEBUGEX')
```

By default a compiler listing and a statement view are produced. Using a compiler listing to obtain the statement numbers, you debug the program using the debug commands.

If the default values for either create command have been changed, you must explicitly specify DBGVIEW(*STMT) and OUTPUT(*PRINT).

Starting the ILE Source Debugger

Once you have created the debug view (statement, source, COPY, or listing), you can begin debugging your application. To start the ILE source debugger, use the Start Debug (STRDBG) command. Once the debugger is started, it remains active until you enter the End Debug (ENDDBG) command.

Initially you can add as many as ten program objects to a debug session by using the Program (PGM) parameter on the STRDBG command. They can be any combination of OPM or ILE programs. You must have *CHANGE authority to a program object to include it in a debug session.

Note: If debugging a program using the COPY or root source view, the source code must be on the same system as the program object being debugged. In addition, the source code must be in a library/file(member) with the same name as when it was compiled.

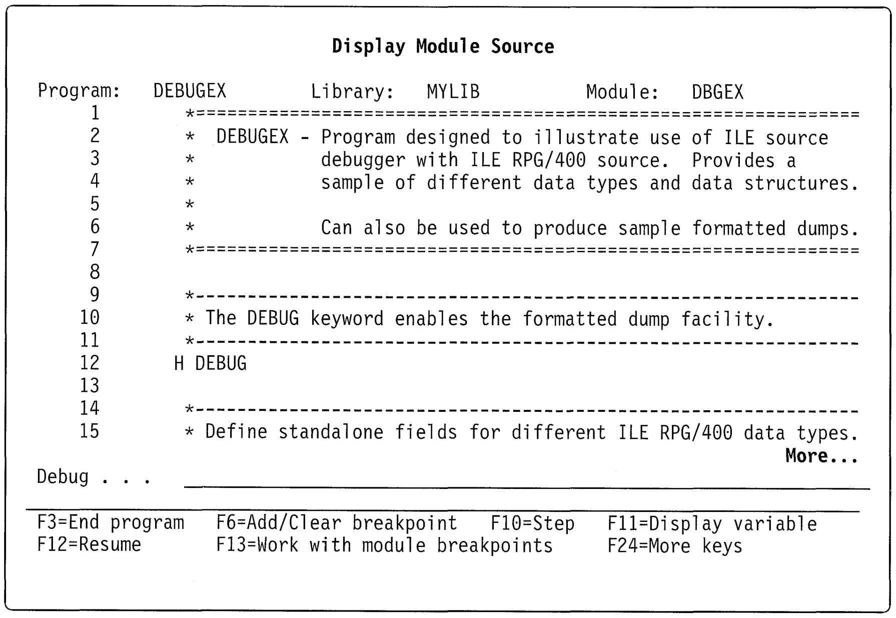
If an ILE RPG/400 program with debug data is in a debug session, the entry module is shown (if it has a debug view.) Otherwise the first module bound to the ILE RPG/400 program with debug data is shown.

For example, to start a debug session for the sample debug program DEBUGEX and a called program RPGPGM, type:

```
STRDBG PGM(MYLIB/DEBUGEX MYLIB/RPGPGM)
```

The Display Module Source display appears as shown in Figure 45. DEBUGEX consists of two modules, an RPG module DBGEX and a C module cproc. See "Sample Source for Debug Examples" on page 147 for the source for DBGEX, cproc, and RPGPGM.

If the entry module has a root source, COPY, or listing view, then the display will show the source of the entry module of the first program. In this case, the program was created using DBGVIEW(*ALL) and so the source for the main module, DBGEX, is shown.



```
                              Display Module Source

Program:   DEBUGEX       Library:   MYLIB         Module:   DBGEX
     1        *===============================================================
     2        *  DEBUGEX - Program designed to illustrate use of ILE source
     3        *            debugger with ILE RPG/400 source.  Provides a
     4        *            sample of different data types and data structures.
     5        *
     6        *            Can also be used to produce sample formatted dumps.
     7        *===============================================================
     8
     9        *---------------------------------------------------------------
    10        * The DEBUG keyword enables the formatted dump facility.
    11        *---------------------------------------------------------------
    12       H DEBUG
    13
    14        *---------------------------------------------------------------
    15        * Define standalone fields for different ILE RPG/400 data types.
                                                                        More...
Debug . . .   _________________________________________________________________

F3=End program   F6=Add/Clear breakpoint   F10=Step   F11=Display variable
F12=Resume       F13=Work with module breakpoints     F24=More keys
```

Figure 45. Display Module Source display for program DEBUGEX

Note: ILE service programs cannot be specified on the STRDBG command. You can add ILE service programs to a debug session by using option 1 (Add) on the Work with Module List display (F14) or by letting the source debugger add it as part of a STEP INTO debug command.

Setting Debug Options

You can set debug options after you start a debug session. Specifically, you can indicate whether database files can be updated while debugging your program. This option affects the UPDPROD parameter of the STRDBG command. The UPDPROD parameter specifies whether database files in a production library can be opened for updating records, or for adding new records, when the job is in debug mode. If not, the files must be copied into a test library before trying to run a program that uses the files.

To set debug options, follow these steps:

1. After entering STRDBG, if the current display is *not* the Display Module Source display, type:

   ```
   DSPMODSRC
   ```

   The Display Module Source display appears.
2. Press F16 (Set debug options) to show the Set Debug Options display.
3. On this display type `Y` (Yes) for the *Update production files* field, and press Enter to return to the Display Module Source display. The database files in production libraries are updated while the job is in debug mode.

Adding/Removing Programs from a Debug Session

You can add more programs to, and remove programs from a debug session, after starting a debug session. You must have *CHANGE authority to a program to add it to or remove it from a debug session.

For ILE programs, you use option 1 (Add program) on the Work with Module List display of the DSPMODSRC command. To remove an ILE program or service program, use option 4 (Remove program) on the same display. When an ILE program or service program is removed, all breakpoints for that program are removed. There is no limit to the number of ILE programs or service programs that can be in or removed from a debug session at one time.

For OPM programs, you use the Add Program (ADDPGM) command or the Remove Program (RMVPGM) command. Only ten OPM programs can be in a debug session at one time.

Example of Adding a Service Program to a Debug Session

In this example you add the service program CVTTOHEX to the debug session which already previously started. (See "Sample Service Program" on page 65 for a discussion of the service program).

1. If the current display is *not* the Display Module Source display, type:

   ```
   DSPMODSRC
   ```

   The Display Module Source display appears.
2. Press F14 (Work with module list) to show the Work with Module List display as shown in Figure 46 on page 121.
3. To add service program CVTTOHEX, on the first line of the display, type: `1` (Add program), `CVTTOHEX` for the *Program/module* field, `MYLIB` for the *Library* field. Change the default program type from *PGM to *SRVPGM and press Enter.
4. Press F12 (Cancel) to return to the Display Module Source display.

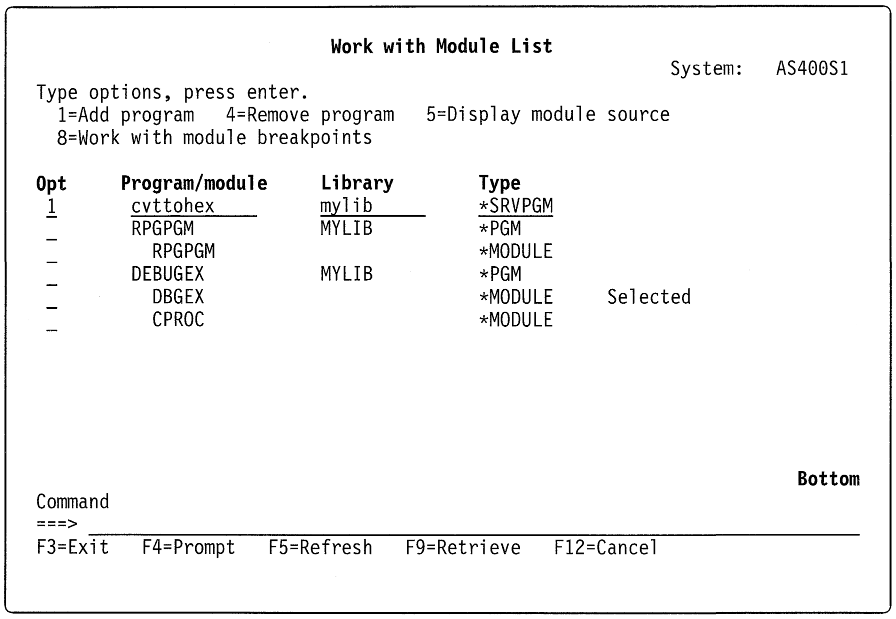

```
                          Work with Module List
                                                      System:   AS400S1
Type options, press enter.
  1=Add program   4=Remove program   5=Display module source
  8=Work with module breakpoints

Opt     Program/module     Library        Type
 1       cvttohex____      mylib_____     *SRVPGM
 _       RPGPGM            MYLIB          *PGM
 _         RPGPGM                         *MODULE
 _       DEBUGEX           MYLIB          *PGM
 _         DBGEX                          *MODULE     Selected
 _         CPROC                          *MODULE







                                                                 Bottom
Command
===> ____________________________________________________________________
F3=Exit   F4=Prompt   F5=Refresh   F9=Retrieve   F12=Cancel
```

Figure 46. Adding an ILE Service Program to a Debug Session

Example of Removing ILE Programs from a Debug Session

In this example you remove the ILE program CVTHEXPGM and the service program CVTTOHEX from a debug session.

1. If the current display is *not* the Display Module Source display, type:

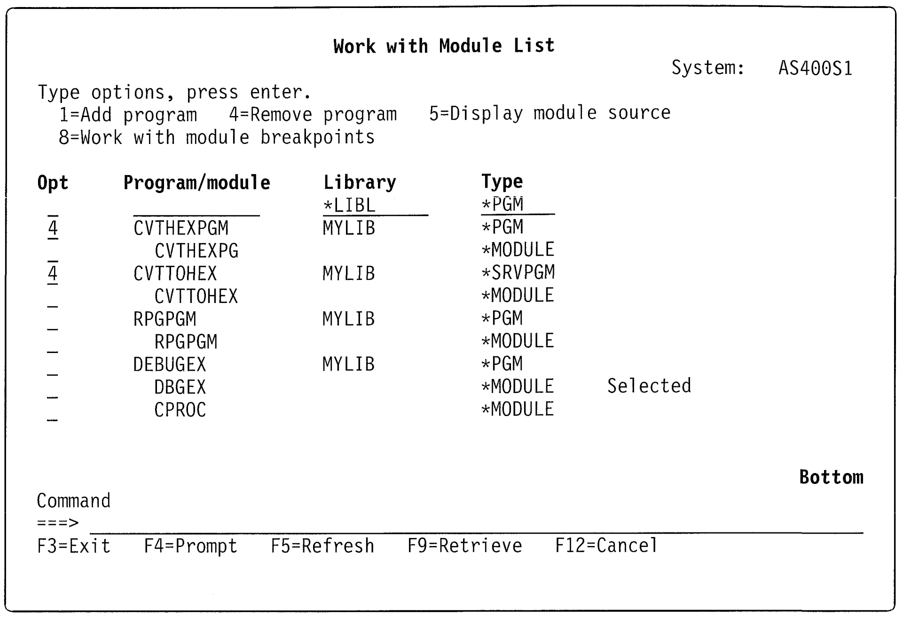
   DSPMODSRC

   The Display Module Source display appears.

2. Press F14 (Work with module list) to show the Work with Module List display as shown in Figure 47 on page 122.

3. On this display type 4 (Remove program) on the line next to CVTHEXPGM and CVTTOHEX, and press Enter.

4. Press F12 (Cancel) to return to the Display Module Source display.



```
                              Work with Module List
                                                                System:   AS400S1
 Type options, press enter.
   1=Add program   4=Remove program   5=Display module source
   8=Work with module breakpoints

 Opt      Program/module      Library        Type
  _       __________          *LIBL____      *PGM___
  4       CVTHEXPGM           MYLIB          *PGM
  _         CVTHEXPG                         *MODULE
  4       CVTTOHEX            MYLIB          *SRVPGM
  _         CVTTOHEX                         *MODULE
  _       RPGPGM              MYLIB          *PGM
  _         RPGPGM                           *MODULE
  _       DEBUGEX             MYLIB          *PGM
  _         DBGEX                            *MODULE    Selected
  _         CPROC                            *MODULE

                                                                        Bottom
 Command
 ===> __________________________________________________________________________
 F3=Exit   F4=Prompt   F5=Refresh   F9=Retrieve   F12=Cancel
```

Figure 47. Removing an ILE Program from a Debug Session

Viewing the Program Source

The Display Module Source display shows the source of a program object one module object at a time. A module object's source can be shown if the module object was compiled using one of the following debug view options:

- DBGVIEW(*SOURCE)
- DBGVIEW(*COPY)
- DBGVIEW(*LIST)
- DBGVIEW(*ALL)

There are two methods to change what is shown on the Display Module Source display:

- Change to a different module
- Change the view of a module

When you change a view, the ILE source debugger maps to equivalent positions in the view you are changing to. When you change the module, the runnable statement on the displayed view is stored in memory and is viewed when the module is displayed again. Line numbers that have breakpoints set are highlighted. When a breakpoint, step, or message causes the program to stop, and the display to be shown, the statement where the breakpoint occurred is highlighted.

Viewing a Different Module

To change the module object that is shown on the Display Module Source display, use option 5 (Display module source) on the Work with Module List display. You access the Work with Module List display from the Display Module Source display by pressing F14 (Work with Module List).

If you use this option with a program object, the entry module with a root source, COPY, or listing view is shown (if it exists). Otherwise the first module object bound to the program object with debug data is shown.

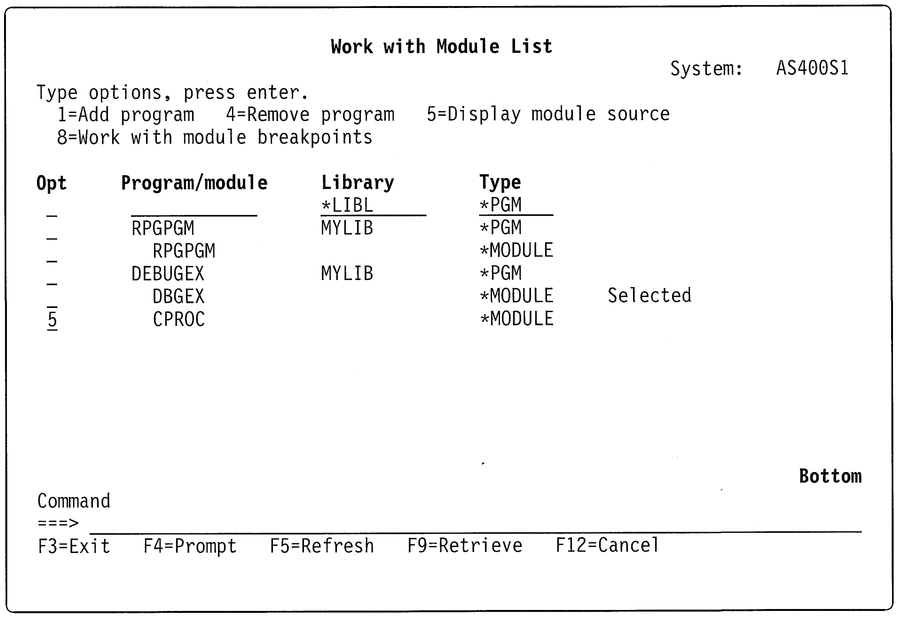
An alternate method of viewing a different module object is to use the DISPLAY debug command. On the debug command line, type:

```
DISPLAY MODULE module-name
```

The module object *module-name* is shown. The module object must exist in a program object that has been added to the debug session.

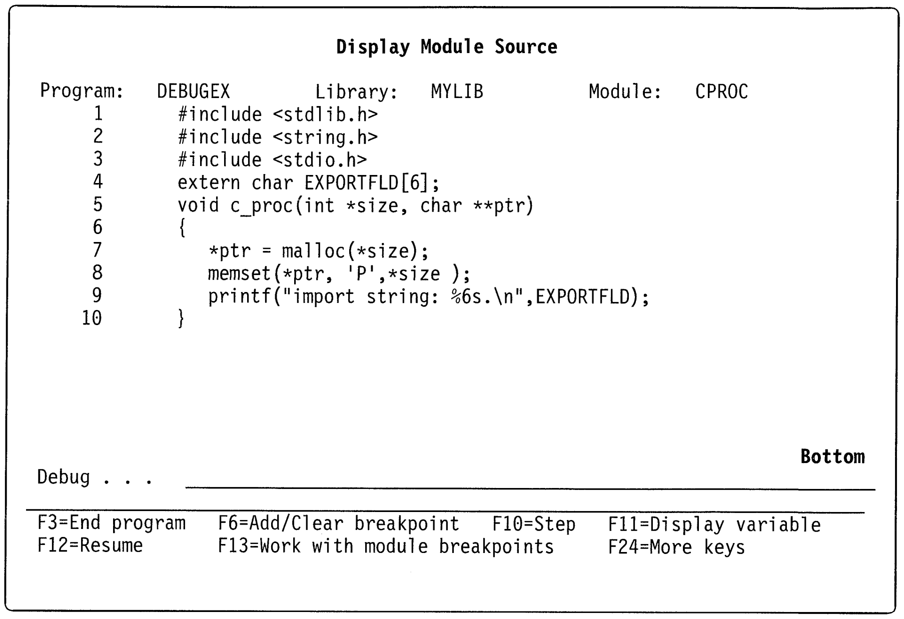
For example, to change from the module DBGEX in Figure 45 on page 119 to the module cproc using the Display module source option, follow these steps:

1. To work with modules type DSPMODSRC, and press Enter. The Display Module Source display is shown.
2. Press F14 (Work with module list) to show the Work with Module List display. Figure 48 shows a sample display.
3. To select cproc, type 5 (Display module source) next to it and press Enter. Since a root source view is available, it is shown, as in Figure 49 on page 124. If a root source was not available, the first module object bound to the program object with debug data is shown.



```
                              Work with Module List
                                                              System:   AS400S1
 Type options, press enter.
   1=Add program   4=Remove program   5=Display module source
   8=Work with module breakpoints

 Opt     Program/module     Library        Type
  _      ____________       *LIBL______    *PGM___
  _       RPGPGM            MYLIB          *PGM
  _         RPGPGM                         *MODULE
  _       DEBUGEX           MYLIB          *PGM
  _         DBGEX                          *MODULE      Selected
  5         CPROC                          *MODULE




                                                                        Bottom
 Command
 ===> _____________________________________________________________________
 F3=Exit   F4=Prompt   F5=Refresh   F9=Retrieve   F12=Cancel
```

Figure 48. Changing to a Different Module



```
                            Display Module Source

Program:   DEBUGEX        Library:   MYLIB          Module:   CPROC
             #include <stdlib.h>
             #include <string.h>
             #include <stdio.h>
             extern char EXPORTFLD[6];
             void c_proc(int *size, char **ptr)
             {
                *ptr = malloc(*size);
                memset(*ptr, 'P',*size );
                printf("import string: %6s.\n",EXPORTFLD);
             }




                                                                     Bottom
Debug . . .   _________________________________________________________________

F3=End program   F6=Add/Clear breakpoint   F10=Step   F11=Display variable
F12=Resume       F13=Work with module breakpoints     F24=More keys
```

Figure 49. Source View of ILE C procedure cproc

Changing the View of a Module

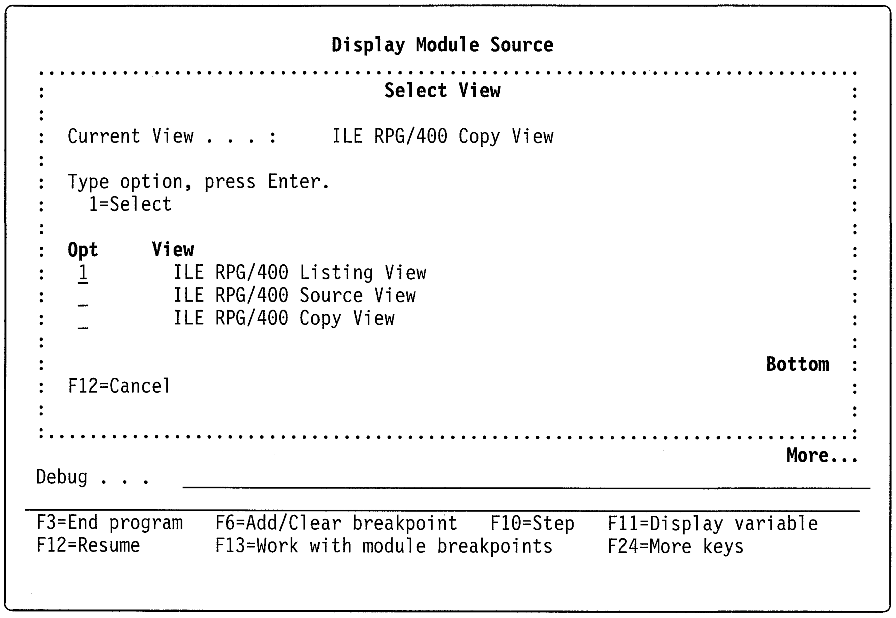
Several different views of an ILE RPG/400 module can be displayed depending on the values you specify when you create the module. They are:

- Root source view
- COPY source view
- Listing view

You can change the view of the module object that is shown on the Display Module Source display through the Select View display. The Select View display can be accessed from the Display Module Source display by pressing F15 (Select View). The Select View display is shown in Figure 50 on page 125. The current view is listed at the top of the window, and the other views that are available are shown below. Each module object in a program object can have a different set of views available, depending on the debug options used to create it.

For example, to change the view of the module from root source to listing, follow these steps:

1. Type DSPMODSRC, and press Enter. The Display Module Source display is shown.
2. Press F15 (Select view). The Select View window is shown in Figure 50 on page 125.



```
                              Display Module Source
 ........................................................................
 :                                 Select View                           :
 :                                                                       :
 :  Current View . . . :      ILE RPG/400 Copy View                      :
 :                                                                       :
 :  Type option, press Enter.                                            :
 :    1=Select                                                           :
 :                                                                       :
 :  Opt      View                                                        :
 :   1         ILE RPG/400 Listing View                                  :
 :   _         ILE RPG/400 Source View                                   :
 :   _         ILE RPG/400 Copy View                                     :
 :                                                                       :
 :                                                               Bottom  :
 :  F12=Cancel                                                           :
 :                                                                       :
 :.......................................................................:
                                                                  More...
 Debug . . .   _________________________________________________________

 F3=End program   F6=Add/Clear breakpoint   F10=Step    F11=Display variable
 F12=Resume       F13=Work with module breakpoints      F24=More keys
```

Figure 50. Changing a View of a Module

The current view is listed at the top of the window, and the other views that are available are shown below. Each module in a program can have a different set of views available, depending on the debug options used to create it.

Note: If a module is created with DBGVIEW(*ALL), the Select View window will show three views available: root source, COPY, and listing. If the module has no /COPY members, then the COPY view is identical to the root source view.

3. Type a 1 next to the listing view, and press Enter. The Display Module Source display appears showing the module with a listing view.

Setting and Removing Breakpoints

You can use breakpoints to halt a program object at a specific point when it is running. An **unconditional breakpoint** stops the program object at a specific statement. A **conditional breakpoint** stops the program object when a specific condition at a specific statement is met.

You set the breakpoints prior to running the program. When the program object stops, the Display Module Source display is shown. The appropriate module object is shown with the source positioned at the line where the breakpoint occurred. This line is highlighted. At this point, you can evaluate fields, set more breakpoints, and run any of the debug commands.

You should know the following characteristics about breakpoints before using them:

- When a breakpoint is set on a statement, the breakpoint occurs *before* that statement is processed.
- When a statement with a conditional breakpoint is reached, the conditional expression associated with the breakpoint is evaluated *before* the statement is

processed. If the expression is true, the breakpoint takes effect and the program stops on that line.

- If the line on which you want to set a breakpoint is not a runnable statement, the breakpoint will be set on the next runnable statement.
- If a breakpoint is bypassed that breakpoint is not processed.
- Breakpoint functions are specified through debug commands. These functions include:
  - Adding breakpoints to program objects
  - Removing breakpoints from program objects
  - Displaying breakpoint information
  - Resuming the running of a program object after a breakpoint has been reached.

If you change the view of the module after setting breakpoints, then the line numbers of the breakpoints are mapped to the new view by the source debugger.

If you are debugging a module or program created with a statement view, then you can set or remove breakpoints using statement numbers obtained from the compiler listing. For more information on using statement numbers, see "Setting and Removing Breakpoints Using Statement Numbers" on page 132.

Setting and Removing Unconditional Breakpoints

You can set or remove an unconditional breakpoint by using:

- F6 (Add/Clear breakpoint) from the Display Module Source display
- F13 (Work with module breakpoints) from the Display Module Source display
- The BREAK debug command to set a breakpoint
- The CLEAR debug command to remove a breakpoint
- The Work with Module Breakpoints display.

The simplest way to set and remove an unconditional breakpoint is to use F6 (Add/Clear breakpoint). The function key acts as a toggle and so it will remove a breakpoint from the line your cursor is on, if a breakpoint is already set on that line.

To remove an unconditional breakpoint using F13 (Work with module breakpoints), place your cursor on the line from which you want to remove the breakpoint and press F13 (Work with module breakpoints). A list of options appear which allow you to set or remove breakpoints. If you select 4 (Clear), a breakpoint is removed from the line.

An alternate method of setting and removing unconditional breakpoints is to use the BREAK and CLEAR debug commands. To set an unconditional breakpoint using the BREAK debug command, type:

```
BREAK line-number
```

on the debug command line. The variable *line-number* is the line number in the currently displayed view of the module object on which you want to set a breakpoint.

To remove an unconditional breakpoint using the CLEAR debug command, type:

```
CLEAR line-number
```

on the debug command line. The variable *line-number* is the line number in the currently displayed view of the module object from which you want to remove a breakpoint.

Example of Setting an Unconditional Breakpoint

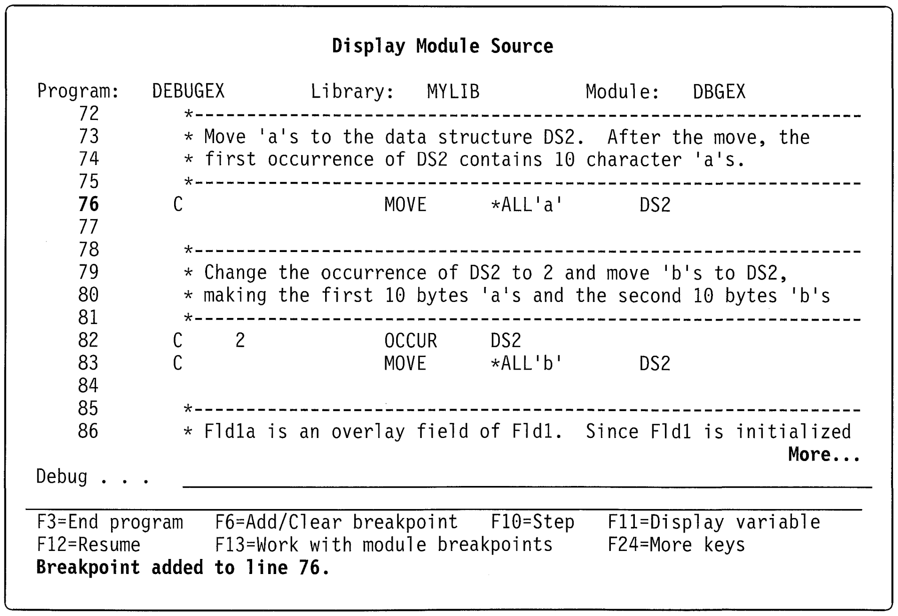
In this example you set an unconditional breakpoint using F6 (Add/Clear breakpoint). The breakpoint is to be set on the first runnable Calculation specification so that the various fields and data structures can be displayed.

1. To work with a module type DSPMODSRC and press Enter. The Display Module Source display is shown.
2. If you want to set the breakpoint in the module shown, continue with step 3. If you want to set a breakpoint in a different module, type:

   ```
   DISPLAY MODULE module-name
   ```

   on the debug command line where *module-name* is the name of the module that you want to display.
3. To set an unconditional breakpoint on the first Calculation specification, place the cursor on line 76.
4. Press F6 (Add/Clear breakpoint). If there is no breakpoint on the line 76, then an unconditional breakpoint is set on that line, as shown in Figure 51. If there is a breakpoint on the line, it is removed.

   Note: Because we want the breakpoint on the *first* Calculation specification, we could have placed the cursor on any line before the start of the Calculation specifications and the breakpoint would still have been placed on line 76, since it is the first runnable statement.



```
                               Display Module Source

 Program:   DEBUGEX        Library:   MYLIB          Module:   DBGEX
     72       *---------------------------------------------------------------
     73       * Move 'a's to the data structure DS2.  After the move, the
     74       * first occurrence of DS2 contains 10 character 'a's.
     75       *---------------------------------------------------------------
     76      C                    MOVE      *ALL'a'       DS2
     77
     78       *---------------------------------------------------------------
     79       * Change the occurrence of DS2 to 2 and move 'b's to DS2,
     80       * making the first 10 bytes 'a's and the second 10 bytes 'b's
     81       *---------------------------------------------------------------
     82      C     2              OCCUR     DS2
     83      C                    MOVE      *ALL'b'       DS2
     84
     85       *---------------------------------------------------------------
     86       * Fld1a is an overlay field of Fld1.  Since Fld1 is initialized
                                                                       More...
 Debug . . .   _______________________________________________________________

 F3=End program   F6=Add/Clear breakpoint   F10=Step   F11=Display variable
 F12=Resume       F13=Work with module breakpoints     F24=More keys
 Breakpoint added to line 76.
```

Figure 51. Setting an Unconditional Breakpoint

5. After the breakpoint is set, press F3 (Exit) to leave the Display Module Source display. The breakpoint is not removed.

6. Call the program. When a breakpoint is reached, the program stops and the Display Module Source display is shown again, with the line containing the breakpoint highlighted. At this point you can step through the program or resume processing.

Setting and Removing Conditional Breakpoints

You can set or remove a conditional breakpoint by using:

- The Work with Module Breakpoints display
- The BREAK debug command to set a breakpoint
- The CLEAR debug command to remove a breakpoint

Note: The relational operators supported for conditional breakpoints are <, >, =, <=, >=, and <> (not equal).

One way you can set or remove conditional breakpoints is through the Work with Module Breakpoints display. You access the Work with Module Breakpoints display from the Display Module Source display by pressing F13 (Work with module breakpoints). The display provides you with a list of options which allow you to either add or remove conditional and unconditional breakpoints. An example of the display is shown in Figure 52 on page 129.

To make the breakpoint conditional, specify a conditional expression in the *Condition* field. If the line on which you want to set a breakpoint is not a runnable statement, the breakpoint will be set at the next runnable statement.

Once you have finished specifying all of the breakpoints, you call the program. You can use F21 (Command Line) from the Display Module Source display to call the program object from a command line or call the program after exiting from the display.

When a statement with a conditional breakpoint is reached, the conditional expression associated with the breakpoint is evaluated before the statement is run. If the result is false, the program object continues to run. If the result is true, the program object stops, and the Display Module Source display is shown. At this point, you can evaluate fields, set more breakpoints, and run any of the debug commands.

An alternate method of setting and removing conditional breakpoints is to use the BREAK and CLEAR debug commands.

To set a conditional breakpoint using the BREAK debug command, type:

```
BREAK line-number WHEN expression
```

on the debug command line. The variable *line-number* is the line number in the currently displayed view of the module object on which you want to set a breakpoint and *expression* is the conditional expression that is evaluated when the breakpoint is encountered. The relational operators supported for conditional breakpoints are noted at the beginning of this section.

In non-numeric conditional breakpoint expressions, the shorter expression is implicitly padded with blanks before the comparison is made. This implicit padding occurs before any National Language Sort Sequence (NLSS) translation. See "National Language Sort Sequence (NLSS)" on page 130 for more information on NLSS.

To remove a conditional breakpoint using the CLEAR debug command, type:

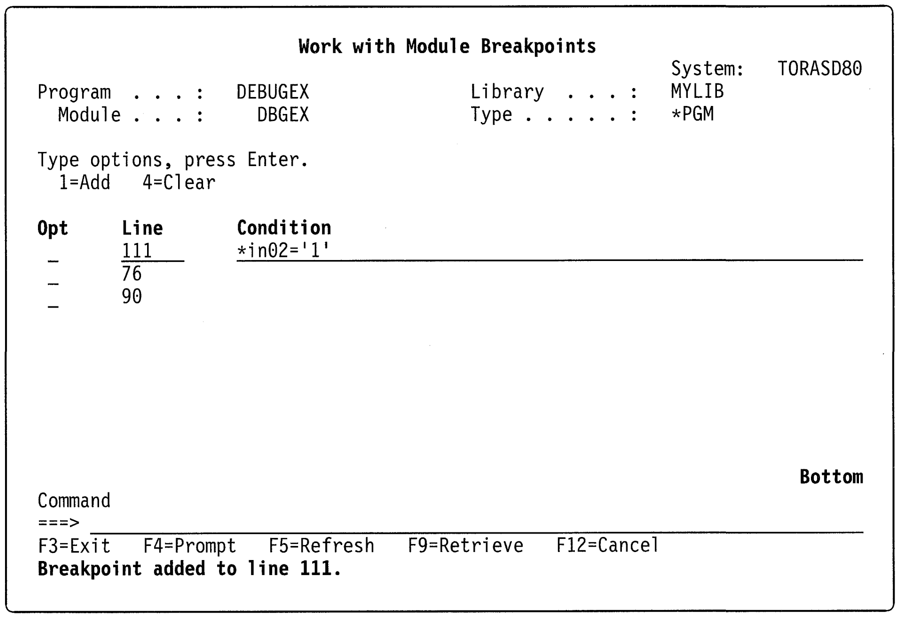
```
CLEAR line-number
```

on the debug command line. The variable *line-number* is the line number in the currently displayed view of the module object from which you want to remove a breakpoint.

Example of Setting a Conditional Breakpoint Using F13

In this example you set a conditional breakpoint using F13 (Work with module breakpoints).

1. To set a conditional breakpoint press F13 (Work with module breakpoints). The Work with Module Breakpoints display is shown.
2. On this display type 1 (Add) on the first line of the list to add a conditional breakpoint.
3. To set a conditional breakpoint at line 111 when *IN02='1', type 111 for the *Line* field, *IN02='1' for the *Condition* field, and press Enter. Figure 52 shows the Work with Module Breakpoints display after adding the conditional breakpoint.



```
                         Work with Module Breakpoints
                                                         System:   TORASD80
 Program  . . . :   DEBUGEX              Library  . . . :   MYLIB
   Module . . . :     DBGEX              Type . . . . . :   *PGM

 Type options, press Enter.
   1=Add   4=Clear

 Opt     Line       Condition
  _      111        *in02='1'
  _      76
  _      90










                                                                     Bottom
 Command
 ===>
 F3=Exit   F4=Prompt   F5=Refresh   F9=Retrieve   F12=Cancel
 Breakpoint added to line 111.
```

Figure 52. Setting a Conditional Breakpoint

A conditional breakpoint is set on line 111. The expression is evaluated before the statement is run. If the result is true (in the example, if *IN02='1'), the program stops, and the Display Module Source display is shown. If the result is false, the program continues to run.

An existing breakpoint is always replaced by a new breakpoint entered at the same location.

4. After the breakpoint is set, press F12 (Cancel) to leave the Work with Module Breakpoints display. Press F3 (End Program) to leave the ILE source debugger. Your breakpoint is not removed.

5. Call the program. When a breakpoint is reached, the program stops, and the Display Module Source display is shown again. At this point you can step through the program or resume processing.

Example of Setting a Conditional Breakpoint Using the BREAK Command

In this example, we want to stop the program when the date field BigDate has a certain value. To specify the conditional breakpoint using the BREAK command:

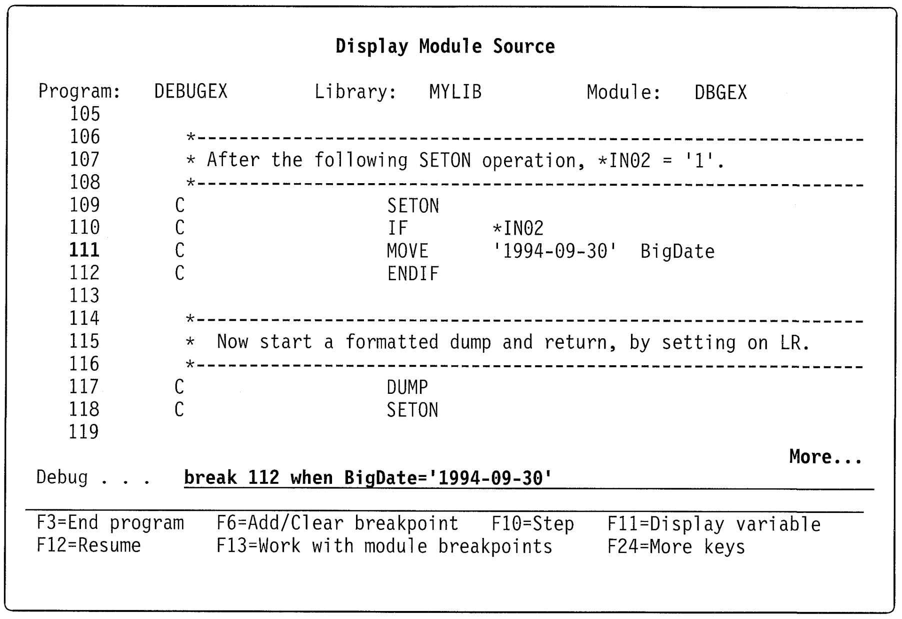
1. From the Display Module Source display, enter:

   ```
   break 112 when BigDate='1994-09-30'
   ```

   A conditional breakpoint is set on line 112.

2. After the breakpoint is set, press F3 (End Program) to leave the ILE source debugger. Your breakpoint is not removed.

3. Call the program. When a breakpoint is reached, the program stops, and the Display Module Source display is shown again.



```
                          Display Module Source

Program:   DEBUGEX        Library:   MYLIB          Module:   DBGEX
   105
   106        *---------------------------------------------------------------
   107        * After the following SETON operation, *IN02 = '1'.
   108        *---------------------------------------------------------------
   109       C                   SETON
   110       C                   IF        *IN02
   111       C                   MOVE      '1994-09-30'  BigDate
   112       C                   ENDIF
   113
   114        *---------------------------------------------------------------
   115        *  Now start a formatted dump and return, by setting on LR.
   116        *---------------------------------------------------------------
   117       C                   DUMP
   118       C                   SETON
   119
                                                                       More...
Debug . . .   break 112 when BigDate='1994-09-30'

F3=End program   F6=Add/Clear breakpoint   F10=Step   F11=Display variable
F12=Resume       F13=Work with module breakpoints     F24=More keys
```

Figure 53. Setting a Conditional Breakpoint Using the BREAK Command

National Language Sort Sequence (NLSS)

Non-numeric conditional breakpoint expressions are divided into the following two types:

- Char- 8: each character contains 8 bits

  This corresponds to the RPG data types of character, date, time, and timestamp.

- Char-16: each character contains 16 bits (DBCS)

  This corresponds to the RPG graphic data type.

NLSS applies only to non-numeric conditional breakpoint expressions of type Char-8. See Table 9 on page 131 for the possible combinations of non-numeric conditional breakpoint expressions.

The sort sequence table used by the source debugger for expressions of type Char-8 is the sort sequence table specified on the SRTSEQ parameter for the CRTRPGMOD or CRTBNDRPG commands.

If the resolved sort sequence table is *HEX, no sort sequence table is used. Therefore, the source debugger uses the hexadecimal values of the characters to determine the sort sequence. Otherwise, the specified sort sequence table is used to assign weights to each byte before the comparison is made. Bytes between, and including, shift-out/shift-in characters are **not** assigned weights. This differs from the way ILE RPG/400 handles comparisons; all characters, including the shift-out/shift-in characters, are assigned weights.

Notes:

1. The alternate sequence specified by ALTSEQ (*SRC) on the Control specification is not available to the ILE source debugger. Instead the source debugger uses the *HEX sort sequence table.
2. The name of the sort sequence table is saved during compilation. At debug time, the source debugger uses the name saved from the compilation to access the sort sequence table. If the sort sequence table specified at compilation time resolves to something other than *HEX or *JOBRUN, it is important the sort sequence table does *not* get altered before debugging is started. If the table cannot be accessed because it is damaged or deleted, the source debugger uses the *HEX sort sequence table.

Table 9. Non-numeric Conditional Breakpoint Expressions.

Type	Possible
Char-8	• Character field compared to character field • Character field compared to character literal 1 • Character field compared to hex literal 2 • Character literal 1 compared to character field • Character literal 1 compared to character literal 1 • Character literal 1 compared to hex literal 2 • Hex literal 2 compared to character field 1 • Hex literal 2 compared to character literal 1 • Hex literal 2 compared to hex literal 2
Char-16	• Graphic field compared to graphic field • Graphic field compared to graphic literal 3 • Graphic field compared to hex literal 2 • Graphic literal 3 compared to graphic field • Graphic literal 3 compared to graphic literal 3 • Graphic literal 3 compared to hex literal 2 • Hex literal 2 compared to graphic field • Hex literal 2 compared to graphic literal 3

1 Character literal is of the form 'abc'.

2 Hexadecimal literal is of the form X'hex digits'.

3 Graphic literal is of the form G'oK1K2i'. Shift-out is represented as o and shift-in is represented as i.

Setting and Removing Breakpoints Using Statement Numbers

You set and remove conditional or unconditional breakpoints using the statement numbers found in the compiler listing for the module in question. This is necessary if you want to debug a module which was created with DBGVIEW(*STMT).

To set an unconditional breakpoint using the BREAK debug command, type:

```
BREAK procedure-name/statement-number
```

on the debug command line. The variable *procedure-name* is the name of the procedure in which you are setting the breakpoint, which for ILE RPG/400 is the module name. (ILE RPG/400 only allows one procedure per module.) The variable *statement-number* is the statement number from the compiler listing on which you want to set a breakpoint.

To set a conditional breakpoint using the BREAK debug command, type:

```
BREAK procedure-name/statement-number WHEN expression
```

on the debug command line. The variables *procedure-name* and *statement-number* are the same as for unconditional breakpoints. The variable *expression* is the conditional expression that is evaluated when the breakpoint is encountered.

To remove an unconditional or conditional breakpoint using the CLEAR debug command, type:

```
CLEAR statement number
```

on the debug command line.

Removing All Breakpoints

You can remove all breakpoints, conditional and unconditional, from a program object that has a module object shown on the Display Module Source display by using the CLEAR PGM debug command. To use the debug command, type:

```
CLEAR PGM
```

on the debug command line. The breakpoints are removed from all of the modules bound to the program.

Stepping Through the Program Object

After a breakpoint is encountered, you can run a specified number of statements of a program object, then stop the program again and return to the Display Module Source display. You do this by using the step function of the ILE source debugger. The program object resumes running on the next statement of the module object in which the program stopped. Typically, a breakpoint is used to stop the program object.

You can step through a program object by using:

- F10 (Step) or F22 (Step into) on the Display Module Source display
- The STEP debug command

The simplest way to step through a program object one statement at a time is to use F10 (Step) or F22 (Step into) on the Display Module Source display. When you press F10 (Step) or F22 (Step into), then next statement of the module object shown in the Display Module Source display is run, and the program object is stopped again.

Note: You cannot specify the number of statements to step through when you use F10 (Step) or F22 (Step into). Pressing F10 (Step) or F22 (Step into) performs a single step.

Another way to step through a program object is to use the STEP debug command. The STEP debug command allows you to run more than one statement in a single step. The default number of statements to run, using the STEP debug command, is one. To step through a program object using the STEP debug command, type:

```
STEP number-of-statements
```

on the debug command line. The variable *number-of-statements* is the number of statements of the program object that you want to run in the next step before the program object is halted again. For example, if you type

```
STEP 5
```

on the debug command line, the next five statements of your program object are run, then the program object is stopped again and the Display Module Source display is shown.

When a CALL statement to another program or procedure is encountered in a debug session, you can:

- Step over the called program or procedure, or
- Step into the called program or procedure.

If you choose to **step over** the called program object, then the CALL statement and the called program object are run as a single step. The called program object is run to completion before the calling program object is stopped at the next step. Step over is the default step mode.

If you choose to **step into** the called program object, then each statement in the called program object is run as a single step. If the next step at which the running program object is to stop falls within the called program object, then the called program object is halted at this point and the called program object is shown in the Display Module Source display.

Note: You cannot step over or step into RPG subroutines.

Stepping Over Program Objects

You can step over program objects by using:

- F10 (Step) on the Display Module Source display
- The STEP OVER debug command

You can use F10 (Step) on the Display Module Source display to step over a called program object in a debug session. If the next statement to be run is a CALL statement to another program object, then pressing F10 (Step) will cause the called program object to run to completion before the calling program object is stopped again.

Alternately, you can use the STEP OVER debug command to step over a called program object in a debug session. To use the STEP OVER debug command, type:

```
STEP number-of-statements OVER
```

on the debug command line. The variable *number-of-statements* is the number of statements of the program object that you want to run in the next step before the program object is halted again. If this variable is omitted, the default is 1. If one of the statements that are run contains a CALL statement to another program object, the ILE source debugger will step over the called program object.

Stepping Into Program Objects

You can step into program objects by using:

- F22 (Step into) on the Display Module Source display
- The STEP INTO debug command

You can use F22 (Step into) on the Display Module Source display to step into a called program object in a debug session. If the next statement to be run is a CALL statement to another program object, then pressing F22 (Step into) will cause the first runnable statement in the called program object to be run. The called program object will then be shown in the Display Module Source display.

Note: The called program object must have debug data associated with it in order for it to be shown in the Display Module Source display.

Alternately, you can use the STEP INTO debug command to step into a called program object in a debug session. To use the STEP INTO debug command, type:

```
STEP number-of-statements INTO
```

on the debug command line. The variable *number-of-statements* is the number of statements of the program object that you want to run in the next step before the program object is halted again. If this variable is omitted, the default is 1.

If one of the statements that are run contains a CALL statement to another program object, the debugger will step into the called program object. Each statement in the called program object will be counted in the step. If the step ends in the called program object then the called program object will be shown in the Display Module Source display. For example, if you type

```
STEP 5 INTO
```

on the debug command line, the next five statements of the program object are run. If the third statement is a CALL statement to another program object, then two statements of the calling program object are run and the first three statements of the called program object are run.

The STEP INTO command works with the CL CALL command as well. You can take advantage of this to step through your program after calling it. After starting the source debugger, from the initial Display Module Source display, enter

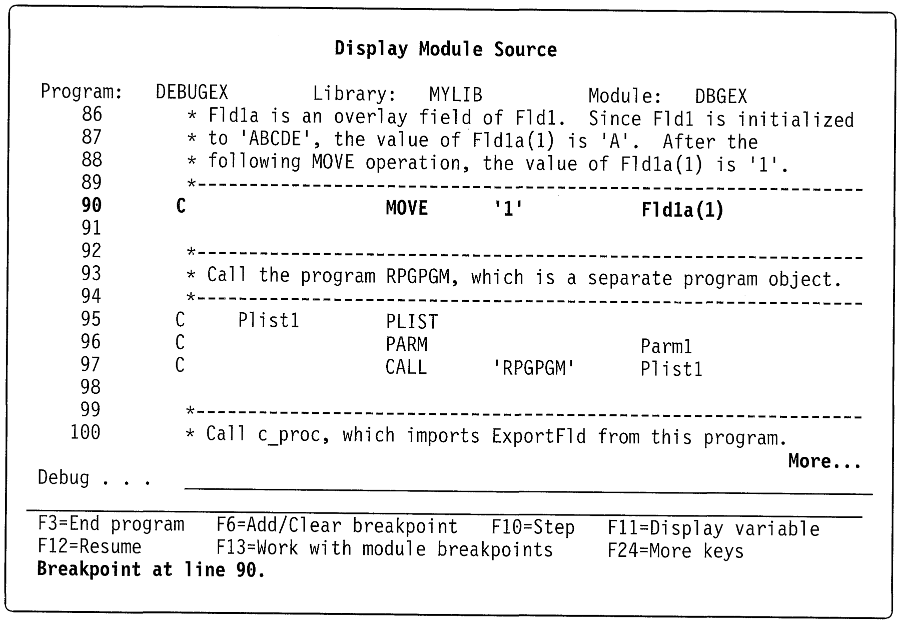
```
STEP 1 INTO
```

This will set the step count to 1. Use the F12 key to return to the command line and then call the program. The program will stop at the first statement with debug data.

Example of Stepping Into a Program Using F22

In this example, you use the F22 (Step Into) to step into the program RPGPGM from the program DEBUGEX.

1. Ensure that the Display Module Source display shows the source for DBGEX.
2. To set an unconditional breakpoint at line 90, which is the last runnable statement before the CALL operation, type `Break` and press Enter.
3. Press F3 (End program) to leave the Display Module Source display.
4. Call the program. The program stops at breakpoint 90, as shown in Figure 54.



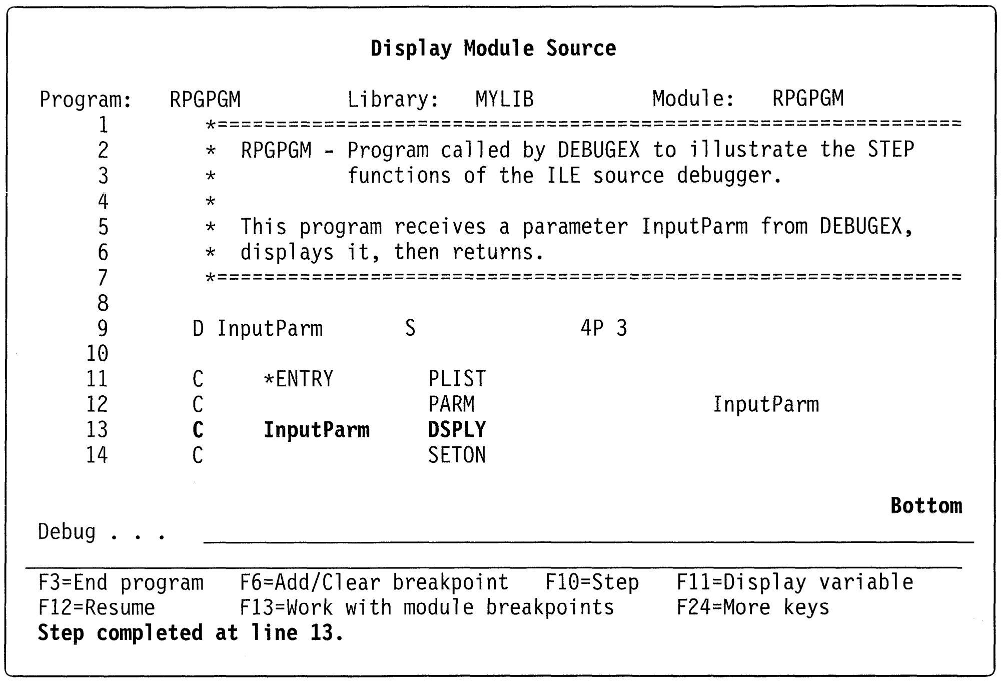
```
                          Display Module Source

Program:   DEBUGEX        Library:   MYLIB          Module:   DBGEX
    86        * Fld1a is an overlay field of Fld1.  Since Fld1 is initialized
    87        * to 'ABCDE', the value of Fld1a(1) is 'A'.  After the
    88        * following MOVE operation, the value of Fld1a(1) is '1'.
    89        *-------------------------------------------------------------
    90       C                   MOVE      '1'           Fld1a(1)
    91
    92        *-------------------------------------------------------------
    93        * Call the program RPGPGM, which is a separate program object.
    94        *-------------------------------------------------------------
    95       C     Plist1        PLIST
    96       C                   PARM                    Parm1
    97       C                   CALL      'RPGPGM'      Plist1
    98
    99        *-------------------------------------------------------------
   100        * Call c_proc, which imports ExportFld from this program.
                                                                      More...
Debug . . .   _________________________________________________________________

F3=End program   F6=Add/Clear breakpoint   F10=Step   F11=Display variable
F12=Resume       F13=Work with module breakpoints     F24=More keys
Breakpoint at line 90.
```

Figure 54. Display Module Source display of DBGEX Before Stepping Into RPGPGM

5. Press F22 (Step into). One statement of the program runs, and then the Display Module Source display of RPGPGM is shown, as in Figure 55 on page 136.

   In this case, the first runnable statement of RPGPGM is processed (line 13) and then the program stops.

   Note: You cannot specify the number of statements to step through when you use F22. Pressing F22 performs a single step.



```
                           Display Module Source

 Program:   RPGPGM         Library:   MYLIB          Module:   RPGPGM
      1       *===============================================================
      2       *  RPGPGM - Program called by DEBUGEX to illustrate the STEP
      3       *           functions of the ILE source debugger.
      4       *
      5       *  This program receives a parameter InputParm from DEBUGEX,
      6       *  displays it, then returns.
      7       *===============================================================
      8
      9       D InputParm       S              4P 3
     10
     11       C     *ENTRY        PLIST
     12       C                   PARM                    InputParm
     13       C     InputParm     DSPLY
     14       C                   SETON
                                                                          Bottom
 Debug . . .   _________________________________________________________________

 F3=End program   F6=Add/Clear breakpoint   F10=Step    F11=Display variable
 F12=Resume       F13=Work with module breakpoints      F24=More keys
 Step completed at line 13.
```

Figure 55. Stepping Into RPGPGM

Stepping Over Procedures

If you specify *over* on the step debug command, calls to procedures, and functions count as single statements. This is the default step mode. You can start the step-over function by using:

- The Step Over debug command
- F10 (Step)

Stepping Into Procedures

If you specify *into* on the STEP debug command, each statement in a procedure or function that is called counts as a single statement. Stepping through four statements of a program could result in running 20 statements if one of the four is a call to a procedure with 16 statements. You can start the step into function by using:

- The Step Into debug command
- F22 (Step into)

The called procedure must have debug data associated with it in order for it to be shown in the Display Module Source display.

Displaying Data and Expressions

You can display the contents of fields, data structures, and arrays, and you can evaluate expressions. There are two ways to display or evaluate:

- F11 (Display Variable)
- EVAL debug command

The scope of the fields used in the EVAL command is defined by using the QUAL command. However, you do not need to specifically define the scope of the fields contained in an ILE RPG/400 module because they are all of global scope.

The easiest way to display data or an expression is to use F11 (Display variable) on the Display Module Source display. To display a field using F11 (Display variable), place your cursor on the field that you want to display and press F11 (Display variable). The current value of the field is shown on the message line at the bottom of the Display Module Source display.

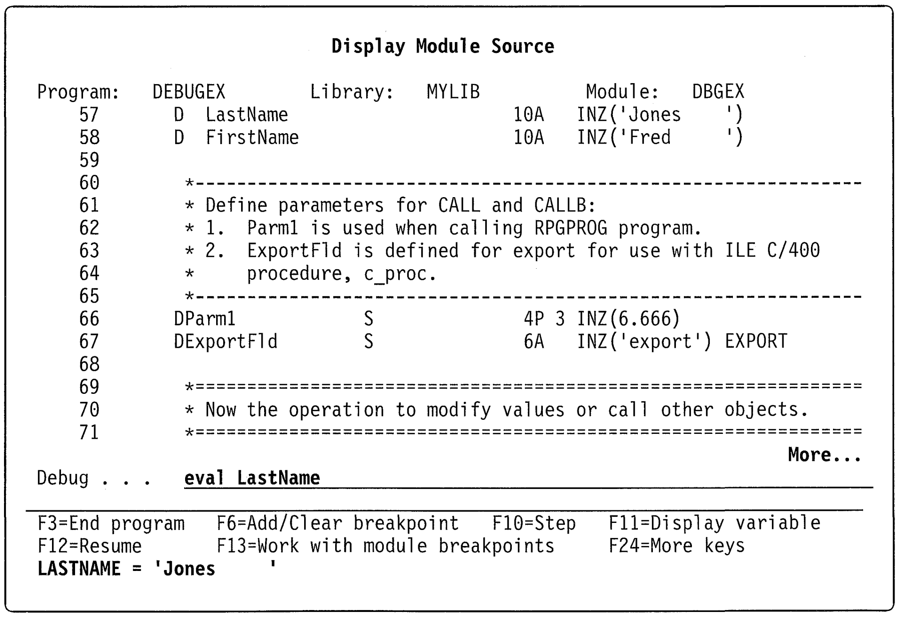
In cases where you are evaluating structures, records, or arrays, the message returned when you press F11 (Display variable) may span several lines. Messages that span several lines are shown on the Evaluate Expression display to show the entire text of the message. Once you have finished viewing the message on the Evaluate Expression display, press Enter to return to the Display Module Source display.

To display data using the EVAL debug command, type:

```
EVAL field-name
```

on the debug command line. The variable *field-name* is the name of the field, data structure, or array that you want to display or evaluate. The value is shown on the message line if the EVAL debug command is entered from the Display Module Source display and the value can be shown on a single line. Otherwise, it is shown on the Evaluate Expression display.

Figure 56 shows an example of using the EVAL debug command to display the contents of a subfield LastName.



```
                               Display Module Source
Program:   DEBUGEX       Library:   MYLIB          Module:   DBGEX
             D  LastName                   10A   INZ('Jones     ')
             D  FirstName                  10A   INZ('Fred      ')

             *---------------------------------------------------------------
             * Define parameters for CALL and CALLB:
             * 1.  Parml is used when calling RPGPROG program.
             * 2.  ExportFld is defined for export for use with ILE C/400
             *     procedure, c_proc.
             *---------------------------------------------------------------
             DParml           S             4P 3 INZ(6.666)
             DExportFld       S             6A   INZ('export') EXPORT

             *===============================================================
             * Now the operation to modify values or call other objects.
             *===============================================================
                                                                        More...
Debug . . .   eval LastName

F3=End program   F6=Add/Clear breakpoint   F10=Step   F11=Display variable
F12=Resume       F13=Work with module breakpoints     F24=More keys
LASTNAME = 'Jones     '
```

Figure 56. Displaying a Field using the EVAL debug command

Figure 57 on page 138 shows the use of the EVAL command with different types of RPG fields. The fields are based on the source in Figure 65 on page 148. Additional examples are also provided in the source debugger online help.

```
Scalar Fields                                      RPG Definition

> EVAL String                                      6A   INZ('ABCDEF')
  STRING = 'ABCDEF'
> EVAL Packed1D0                                   5P 2 INZ(-93.4)
  PACKED1D0 = -093.40
> EVAL ZonedD3D2                                   3S 2 INZ(-3.21)
  ZONEDD3D2 = -3.21
> EVAL Bin4D3                                      4B 3 INZ(-4.321)
  BIN4D3 = -4.321
> EVAL DBCSString                                  3G   INZ(G' BBCCDD ')
  DBCSSTRING = ' BBCCDD '
> EVAL NullPtr                                      *   INZ(*NULL)
  NULLPTR = SYP:*NULL

Based Fields
> EVAL String                                      6A   INZ('ABCDEF')
  STRING = 'ABCDEF'
> EVAL BasePtr                                      *   INZ(%ADDR(String))
  BASEPTR = SPP:C01947001218
> EVAL BaseString                                  6A   BASED(BasePtr)
  BASESTRING = 'ABCDEF'

Date, Time, Timestamp Fields
> EVAL BigDate                                      D   INZ(D'9999-12-31')
  BIGDATE = '9999-12-31'
> EVAL BigTime                                      T   INZ(T'12.00.00')
  BIGTIME = '12.00.00'
> EVAL BigTstamp                                    Z   INZ(Z'9999-12-31-12.00.00.000000
  BIGTSTAMP = '9999-12-31-12.00.00.000000'
```

Figure 57. Sample EVAL commands based on Module DBGEX

Displaying the Contents of an Array

Specifying an array name with EVAL will display the full array. To display one element of an array, specify the index of the element you wish to display in parentheses. You can also use the %INDEX debug built-in function.

To display a range of elements use the following range notation:

```
EVAL field-name (n...m)
```

The variable *field-name* is the name of the array, the variable *n* is a number representing the start of the range, and the variable *m* is a number representing the end of the range.

Figure 58 on page 139 shows the use of EVAL with the array in DBGEX.

```
> EVAL Arry                                 3S 2 DIM(2) INZ(1.23)
  ARRY(1) = 1.23    ** Display full array **
  ARRY(2) = 1.23

> EVAL Arry(2)      ** Display second element **
  ARRY(2) = 1.23

> EVAL Arry(1..2)   ** Display range of elements **
  ARRY(1) = 1.23
  ARRY(2) = 1.23
```

Figure 58. Sample EVAL commands for an Array

Displaying the Contents of a Table

Using EVAL on a table will result in a display of the current table element. You can display the whole table using the range notation. For example, to display a 3-element table, type:

```
EVAL TableA(1..3)
```

You can change the current element using the %INDEX built-in function. Figure 59 shows the use of EVAL with the table in DBGEX.

```
                                            3    DIM(3) CTDATA

                                            Compile-time data:  **
> EVAL TableA             ** Show value at                      aaa
  TABLEA = 'aaa'             current index                      bbb
                                                                ccc
> EVAL TableA(1)          ** Specify index 1 **
  TABLEA(1) = 'aaa'
> EVAL TableA(2)          ** Specify index 2 **
  TABLEA(2) = 'bbb'

> EVAL TableA(1..3)       ** Specify the whole table **
  TABLEA(1) = 'aaa'
  TABLEA(2) = 'bbb'
  TABLEA(3) = 'ccc'

> EVAL TableA=%INDEX(3)   ** Change current index to 3 **
> EVAL TableA
  TABLEA = 'ccc'
```

Figure 59. Sample EVAL commands for a Table

Displaying Data Structures

You display the contents of a data structure or its subfields as you would any standalone field. You simply use the data structure name after EVAL to see the entire contents, or the subfield name to see a subset.

When displaying a multiple-occurrence data structure, an EVAL on the data structure name will show the subfields using the current index. To specify a particular occurrence, specify the index in parentheses following the data structure name. For example, to display the contents of the second occurrence of DS1, type:

```
EVAL DS1(2)
```

Similarly, to view the contents of a particular occurrence of a subfield, use the index notation.

If a subfield is defined as an array overlay of another subfield, to see the contents of the overlay subfield, you can use the %INDEX built-in function to specify the occurrence, and the index notation to specify the array.

An alternative way of displaying a subfield which is an array overlay is to use the following notation:

```
EVAL subfield-name(occurrence-index,array-index)
```

where the variable *subfield-name* is the name of the subfield you wish to display, *occurrence-index* is the number of the array occurrence to display, and *array-index* is the number of the element to display.

Figure 60 shows some examples of using EVAL with the the data structures defined in DBGEX.

```
** Note that you can enter the data structure name or a subfield name. **

> EVAL DS3
  TITLE OF DS3 = 'Mr.  '                 5A   INZ('Mr.  ')
  LASTNAME OF DS3 = 'Jones     '        10A   INZ('Jones     ')
  FIRSTNAME OF DS3 = 'Fred      '       10A   INZ('Fred      ')

> EVAL LastName
  LASTNAME = 'Jones     '

> EVAL DS1                                    OCCURS(3)
  FLD1 OF DS1 = 'ABCDE'                  5A   INZ('ABCDE')
  FLD1A OF DS1(1) = 'A'                  1A   DIM(5) OVERLAY(Fld1)
  FLD1A OF DS1(2) = 'B'                  5B 2 INZ(123.45)
  FLD1A OF DS1(3) = 'C'
  FLD1A OF DS1(4) = 'D'
  FLD1A OF DS1(5) = 'E'
  FLD2 OF DS1 = 123.45

> EVAL DS1=%INDEX(2)      ** Change the occurrence of DS1 **
  DS1=%INDEX(2) = 2
```

Figure 60 (Part 1 of 2). Using EVAL with Data Structures

```
> EVAL Fld1                 ** Display a Subfield  **
  FLD1 = 'ABCDE'               (current occurrence)
> EVAL fld1(2)
  FLD1(2) = 'ABCDE'            (second occurrence)

> EVAL Fld1a                ** Display an Array Overlay Subfield **
  FLD1A OF DS1(1) = 'A'        (current occurrence)
  FLD1A OF DS1(2) = 'B'
  FLD1A OF DS1(3) = 'C'
  FLD1A OF DS1(4) = 'D'
  FLD1A OF DS1(5) = 'E'

> EVAL Fld1a(2,1)           ** Display 2nd occurrence, 1st element  **
  FLD1A(2,1) = 'A'

> EVAL Fld1a(2,1..2)        ** Display 2nd occurrence, 1st - 2nd elements **
  FLD1A(2,1) = 'A'
  FLD1A(2,2) = 'B'
```

Figure 60 (Part 2 of 2). Using EVAL with Data Structures

To display a data structure for which no subfields have been defined, you must use the character display function of EVAL which is discussed below.

Displaying Indicators

Indicators are defined as 1-byte character fields. Except for indicators such as *INLR, you can display indicators either as '*INxx' or '*IN(xx)'. Because the system stores indicators as an array, you can display them all or some subset of them using the range notation. For example, if you enter EVAL *IN, you will get a list of indicators 01 to 99. To display indicators *IN01 to *IN06 you would enter EVAL *IN(1..6).

Figure 61 shows each of these ways using the indicators as they were set in DBGEX.

```
> EVAL IN02
  Identifier does not exist.
> EVAL *IN02
  *IN02 = '1'
> EVAL *IN(02)
  *IN(02) = '1'

> EVAL *INLR
  *INLR = '0'
> EVAL *IN(LR)
  Identifier does not exist.

> EVAL *IN(1..6)          ** To display a range of indicators **
  *IN(1) = '0'
  *IN(2) = '1'
  *IN(3) = '0'
  *IN(4) = '1'
  *IN(5) = '0'
  *IN(6) = '1'
```

Figure 61. Sample EVAL commands for an Array

Displaying Fields as Hexadecimal Values

You can use the EVAL debug command to display the value of fields in hexadecimal format. To display a variable in hexadecimal format, type:

```
EVAL field-name: x number-of-bytes
```

on the debug command line. The variable *field-name* is the name of the field that you want to display in hexadecimal format. 'x' specifies that the field is to be displayed in hexadecimal format. The variable *number-of-bytes* indicates the number of bytes displayed. If no length is specified after the 'x', the size of the field is used as the length. A minimum of 16 bytes is always displayed. If the length of the field is less than 16 bytes, then the remaining space *is filled with zeroes* until the 16 byte boundary is reached.

For example, the field String is defined as six-character string. To find out the hexadecimal equivalent of the first 3 characters, you would enter:

```
EVAL String: x 3
   Result:
   00000     C1C2C300 00000000 00000000 00000000   - ABC.............
```

Displaying Fields in Character Format

You can use the EVAL debug command to display a field in character format. To display a variable in character format, type:

```
EVAL field-name: c number-of-characters
```

on the debug command line. The variable *field-name* is the name of the field that you want to display in character format. 'c' specifies the number of characters to display.

For example, in the program DEBUGEX, data structure DS2 does not have any subfields defined. Several MOVE operations move values into the subfield.

Because there are no subfields defined, you cannot display the data structure. Therefore, to view its contents you can use the character display function of EVAL.

```
EVAL DS2:C 20              Result:   DS2:C 20 = 'aaaaaaaaaabbbbbbbbbb'
```

Using Debug Built-In Functions

The following built-in functions are available while using the ILE source debugger:

%SUBSTR	Substring a string field.
%ADDR	Retrieve the address of a field.
%INDEX	Change the index of a table or multiple-occurrence data structure.

The %SUBSTR built-in function allows you to substring a string variable. The first parameter must be a string identifier, the second parameter is the starting position, and the third parameter is the number of single-byte or double-byte characters. In addition. the second and third parameters must be positive, integer literals. Parameters are delimited by one or more spaces.

Use the %SUBSTR built-in function to:

- Display a portion of a character field
- Assign a portion of a character field

- Use a portion of a character field on either side of a conditional break expression.

Figure 62 shows some examples of the use of %SUBSTR based on the source in Figure 65 on page 148.

```
 > EVAL String
   STRING = 'ABCDE '
** Display the first two characters of String **
 > EVAL %substr (String 1 2)
   %SUBSTR (STRING 1 2) = 'AB'

 > EVAL TableA
   TABLEA = 'aaa'
** Display the first character in the first table element **
 > EVAL %substr(TableA 1 1)
   %SUBSTR(TABLEA 1 1) = 'a'

 > EVAL BigDate
   BIGDATE = '1994-10-23'
** Set String equal to the first four characters of BigDate **
 > EVAL String=%substr(BigDate 1 4)
   STRING=%SUBSTR(BIGDATE 1 4) = '1994  '

 > EVAL Fld1          (5 characters)
   FLD1 = 'ABCDE'
 > EVAL String        (6 characters)
   STRING = '123456'
** Set the characters 2-5 of String equal to the
              first four characters of Fld1 **
 > EVAL %substr(String 2 4) = %substr(Fld1 1 4)
   %SUBSTR(STRING 2 4) = %SUBSTR(FLD1 1 4) = 'ABCD'
 > EVAL String
   STRING = '1ABCD6'

** You can only use %SUBSTR on character or graphic strings! **
 > EVAL %substr (Packed1D0 1 2)
   String type error occurred.
```

Figure 62. Examples of %SUBSTR using DBGEX

To change the current index, you can use the %INDEX built-in function, where the index is specified in parentheses following the function name. An example of %INDEX is found in the table section of Figure 59 on page 139 and Figure 60 on page 140.

Note: %INDEX will change the current index to the one specified. Therefore, any source statements which refer to the table or multiple-occurrence data structure subsequent to the EVAL statement may be operating with a different index than expected.

Changing the Value of Fields

You can change the value of fields by using the EVAL command with an assignment operator (=).

The scope of the fields used in the EVAL command is defined by using the QUAL command. However, you do not need to specifically define the scope of the fields contained in an ILE RPG/400 module because they are all of global scope.

To change the value of the field, type:

```
EVAL field-name = value
```

on the debug command line. *field-name* is the name of the variable that you want to change and *value* is an identifier, literal, or constant value that you want to assign to variable *field-name*. For example,

```
EVAL COUNTER=3
```

changes the value of *COUNTER* to 3 and shows

```
COUNTER=3 = 3
```

on the message line of the Display Module Source display.

Use the EVAL debug command to assign numeric, alphabetic, and alphanumeric data to fields. You can also use the %SUBSTR built-in function in the assignment expression.

When you assign values to a character field, the following rules apply:

- If the length of the source expression is less than the length of the target expression, then the data is left justified in the target expression and the remaining positions are filled with blanks.
- If the length of the source expression is greater than the length of the target expression, then the data is left justified in the target expression and truncated to the length of the target expression.

Note: Graphic fields can be assigned any of the following:

- Another graphic field
- A graphic literal of the form G'oK1K2i'
- A hexadecimal literal of the form X'hex digits'

When assigning literals to fields, the normal RPG rules apply:

- Character literals should be in quotes.
- Graphic literals should be specified as G'oDDDDi', where o is shift-out and i is shift-in.
- Hexadecimal literals should be in quotes, preceded by an 'x'.
- Numeric literals should not be in quotes.

Note: You cannot assign a figurative constant to a field using the EVAL debug command. Figurative constants are not supported by the EVAL debug command.

Figure 63 on page 145 shows some examples of changing field values based on the source in Figure 65 on page 148. Additional examples are also provided in the source debugger online help.

```
** Target Length = Source Length **

 > EVAL String='123456'     (6 characters)
   STRING='123456' = '123456'
 > EVAL ExportFld           (6 characters)
   EXPORTFLD = 'export'
 > EVAL String=ExportFld
   STRING=EXPORTFLD = 'export'

** Target Length < Source Length **

 > EVAL String              (6 characters)
   STRING = 'ABCDEF'
 > EVAL LastName            (10 characters)
   LASTNAME='Williamson' = 'Williamson'
 > EVAL String=LastName
   STRING=LASTNAME = 'Willia'

** Target Length > Source Length **

 > EVAL String               (6 characters)
   STRING = '123456'
 > EVAL TableA               (3 characters)
   TABLEA = 'aaa'
 > EVAL String=TableA
   STRING=TABLEA = 'aaa   '

** Using %SUBSTR **

 > EVAL BigDate
   BIGDATE = '1994-10-23'
 > EVAL String=%SUBSTR(BigDate 1 4)
   STRING=%SUBSTR(BIGDATE 1 4) = '1994  '

** Substring Target Length > Substring Source Length **
 > EVAL string = '123456'
   STRING = '123456' = '123456'
 > EVAL LastName='Williamson'
   LASTNAME='Williamson' = 'Williamson'
 > EVAL String = %SUBSTR(Lastname 1 8)
   STRING = %SUBSTR(LASTNAME 1 8) = 'Willia'

** Substring Target Length < Substring Source Length **
 > EVAL TableA
   TABLEA = 'aaa'
 > EVAL String
   STRING = '123456'
 > EVAL String=%SUBSTR(TableA 1 4)
   Substring extends beyond end of string.      ** Error **
 > EVAL String
   STRING = '123456'
```

Figure 63. Examples of Changing the Values of Fields based on DBGEX

Displaying Attributes of a Field

You can display the attributes of a field using the Attribute (ATTR) debug command. The attributes are the size (in bytes) and type of the variable as recorded in the debug symbol table.

Figure 64 shows some examples of displaying field attributes based on the source in Figure 65 on page 148. Additional examples are also provided in the source debugger online help.

```
> ATTR NullPtr
  TYPE = PTR, LENGTH = 16 BYTES
> ATTR ZonedD3D2
  TYPE = ZONED(3,2), LENGTH = 3 BYTES
> ATTR Bin4D3
  TYPE = BINARY, LENGTH = 2 BYTES
> ATTR Arry
  TYPE = ARRAY, LENGTH = 6 BYTES
> ATTR tablea
  TYPE = FIXED LENGTH STRING, LENGTH = 3 BYTES
> ATTR tablea(2)
  TYPE = FIXED LENGTH STRING, LENGTH = 3 BYTES
> ATTR BigDate
  TYPE = FIXED LENGTH STRING, LENGTH = 10 BYTES
> ATTR DS1
  TYPE = RECORD, LENGTH = 9 BYTES
> ATTR SpcPtr
  TYPE = PTR, LENGTH = 16 BYTES
> ATTR String
  TYPE = FIXED LENGTH STRING, LENGTH = 6 BYTES
> ATTR *IN02
  TYPE = CHAR, LENGTH = 1 BYTES
> ATTR DBCSString
  TYPE = FIXED LENGTH STRING, LENGTH = 6 BYTES
```

Figure 64. Examples of Displaying the Attributes of Fields based on DBGEX

Equating a Name with a Field, Expression, or Command

You can use the EQUATE debug command to equate a name with a field, expression or debug command for shorthand use. You can then use that name alone or within another expression. If you use it within another expression, the value of the name is determined before the expression is evaluated. These names stay active until a debug session ends or a name is removed.

To equate a name with a field, expression or debug command, type:

```
EQUATE shorthand-name definition
```

on the debug command line. *shorthand-name* is the name that you want to equate with a field, expression, or debug command, and *definition* is the field, expression, or debug command that you are equating with the name.

For example, to define a shorthand name called *DC* which displays the contents of a field called *COUNTER*, type:

```
EQUATE DC EVAL COUNTER
```

on the debug command line. Now, each time *DC* is typed on the debug command line, the command EVAL *COUNTER* is performed.

The maximum number of characters that can be typed in an EQUATE command is 144. If a definition is not supplied and a previous EQUATE command defined the name, the previous definition is removed. If the name was not previously defined, an error message is shown.

To see the names that have been defined with the EQUATE debug command for a debug session, type:

```
DISPLAY EQUATE
```

on the debug command line. A list of the active names is shown on the Evaluate Expression display.

Source Debug National Language Support for ILE RPG/400

You should be aware of the following conditions that exist when you are working with source debug National Language Support for ILE RPG/400

- When a view is displayed on the Display Module Source display, the source debugger converts all data to the Coded Character Set Identifier (CCSID) of the debug job.
- When assigning literals to fields, the source debugger will not perform CCSID conversion on quoted literals (for example, 'abc'). Also, quoted literals are case sensitive.

See the chapter on debugging in *ILE Concepts* for more information on NLS restrictions.

Sample Source for Debug Examples

Figure 65 on page 148 shows the source for the main procedure of the program DEBUGEX. Most of the examples and screens shown in this chapter are based on this source. Figure 66 on page 151 and Figure 67 on page 151 show the source for the called program RPGPGM and procedure cproc respectively.

The program DEBUGEX is designed to show the different aspects of the ILE source debugger and ILE RPG/400 formatted dumps. The sample dumps are provided in the next chapter.

The following steps describe how the program DEBUGEX was created for use in these examples:

1. To create the module DBGEX using the source in Figure 65 on page 148, type:

   ```
   CRTRPGMOD MODULE(MYLIB/DBGEX) SRCFILE(MYLIB/QRPGLESRC) DBGVIEW(*ALL)
             TEXT('Main module for Sample Debug Program')
   ```

   DBGVIEW(*ALL) was chosen in order to show the different views available.
2. To create the C module using the source in Figure 67 on page 151, type:

   ```
   CRTCMOD MODULE(MYLIB/cproc) SRCFILE(MYLIB/QCLESRC) DBGVIEW(*SOURCE)
             TEXT('C procedure for Sample Debug Program')
   ```
3. To create the program DEBUGEX, type:

```
CRTPGM PGM(MYLIB/DEBUGEX) MODULE(MYLIB/DBGEX MYLIB/CPROC)
          TEXT('Sample Debug Program')
```

The first module DBGEX is the entry module for this program. The program will run in a new activation group (that is, *NEW) when it is called.

4. To create the called RPG program using the source in Figure 66 on page 151, type:

```
CRTBNDRPG PGM(MYLIB/RPGPGM) DFTACTGRP(*NO)
         DBGVIEW(*SOURCE) ACTGRP(*NEW)
         TEXT('RPG program for Sample Debug Program')
```

We could have created RPGPGM to run in the OPM default activation group. However, we decided to have it run in the same activation group as DEBUGEX, and since DEBUGEX needs only a temporary activation group, *NEW was chosen for both programs.

```
 *=================================================================*
 *  DEBUGEX - Program designed to illustrate use of ILE source     *
 *           debugger with ILE RPG/400 source.  Provides a         *
 *           sample of different data types and data structures.  *
 *                                                                 *
 *           Can also be used to produce sample formatted dumps.  *
 *=================================================================*

 *-----------------------------------------------------------------*
 * The DEBUG keyword enables the formatted dump facility.          *
 *-----------------------------------------------------------------*
H DEBUG

 *-----------------------------------------------------------------*
 * Define standalone fields for different ILE RPG/400 data types.  *
 *-----------------------------------------------------------------*
D String          S              6A   INZ('ABCDEF')
D Packed1D0       S              5P 2 INZ(-93.4)
D ZonedD3D2       S              3S 2 INZ(-3.21)
D Bin4D3          S              4B 3 INZ(-4.321)
D Bin9D7          S              9B 7 INZ(98.7654321)
D DBCSString      S              3G   INZ(G' BBCCDD ')

 *   Pointers
D NullPtr         S               *   INZ(*NULL)
D BasePtr         S               *   INZ(%ADDR(String))
D ProcPtr         S               *   ProcPtr INZ(%PADDR('c_proc'))
D BaseString      S              6A   BASED(BasePtr)
D BaseOnNull      S             10A   BASED(NullPtr)
 *
D Spcptr          S               *
D SpcSiz          S              9B 0 INZ(8)
```

```
 *   Date, Time, Timestamp
D BigDate         S               D   INZ(D'9999-12-31')
D BigTime         S               T   INZ(T'12.00.00')
D BigTstamp       S               Z   INZ(Z'9999-12-31-12.00.00.000000')

 *   Array
D Arry            S             3S 2 DIM(2) INZ(1.23)

 *   Table
D TableA          S             3    DIM(3) CTDATA

 *---------------------------------------------------------------*
 * Define different types of data structures.                    *
 *---------------------------------------------------------------*
D DS1             DS                  OCCURS(3)
D  Fld1                          5A   INZ('ABCDE')
D  Fld1a                         1A   DIM(5) OVERLAY(Fld1)
D  Fld2                          5B 2 INZ(123.45)
 *
D DS2             DS            10    OCCURS(2)
 *
D DS3             DS
D  Title                         5A   INZ('Mr.  ')
D  LastName                     10A   INZ('Jones     ')
D  FirstName                    10A   INZ('Fred      ')

 *---------------------------------------------------------------*
 * Define parameters for CALL and CALLB:                         *
 * 1.  Parm1 is used when calling RPGPROG program.               *
 * 2.  ExportFld is defined for export for use with ILE C/400    *
 *     procedure, c_proc.                                        *
 *---------------------------------------------------------------*
DParm1            S             4P 3 INZ(6.666)
DExportFld        S             6A   INZ('export') EXPORT

 *===============================================================*
 * Now the operations to modify values or call other objects.    *
 *===============================================================*
 *---------------------------------------------------------------*
 * Move 'a's to the data structure DS2.  After the move, the     *
 * first occurrence of DS2 contains 10 character 'a's.           *
 *---------------------------------------------------------------*
C                   MOVE      *ALL'a'       DS2

 *---------------------------------------------------------------*
 * Change the occurrence of DS2 to 2 and move 'b's to DS2,       *
 * making the first 10 bytes 'a's and the second 10 bytes 'b's  .*
 *---------------------------------------------------------------*
C     2             OCCUR     DS2
C                   MOVE      *ALL'b'       DS2
```

```
      *-----------------------------------------------------------------*
      * Fld1a is an overlay field of Fld1.  Since Fld1 is initialized   *
      * to 'ABCDE', the value of Fld1a(1) is 'A'.  After the            *
      * following MOVE operation, the value of Fld1a(1) is '1'.         *
      *-----------------------------------------------------------------*
     C                   MOVE      '1'           Fld1a(1)

      *-----------------------------------------------------------------*
      * Call the program RPGPGM, which is a separate program object.    *
      *-----------------------------------------------------------------*
     C     Plist1        PLIST
     C                   PARM                    Parm1
     C                   CALL      'RPGPGM'      Plist1

      *-----------------------------------------------------------------*
      * Call c_proc, which imports ExportFld from this program.         *
      *-----------------------------------------------------------------*
     C                   CALLB     'c_proc'
     C                   PARM                    SpcSiz
     C                   PARM                    SpcPtr

      *-----------------------------------------------------------------*
      * After the following SETON operation, *IN02 = '1'.              *
      *-----------------------------------------------------------------*
     C                   SETON                                        020406
     C                   IF        *IN02
     C                   MOVE      '1994-09-30'  BigDate

      *-----------------------------------------------------------------*
      *  Now start a formatted dump and return, by setting on LR.       *
      *-----------------------------------------------------------------*
     C                   DUMP
     C                   SETON                                        LR

      *=================================================================*
      * Compile-time data section for Table.                            *
      *=================================================================*
**
aaa
bbb
ccc
```

Figure 65 (Part 3 of 3). Source for Module DBGEX. DBGEX is the main module of the program DEBUGEX.

```
*===================================================================*
*  RPGPGM - Program called by DEBUGEX to illustrate the STEP        *
*           functions of the ILE source debugger.                   *
*                                                                   *
*  This program receives a parameter InputParm from DEBUGEX,        *
*  displays it, then returns.                                       *
*===================================================================*

D InputParm       S              4P 3

C     *ENTRY        PLIST
C                   PARM                    InputParm
C     InputParm     DSPLY
C                   SETON                                        LR
```

Figure 66. Source for Program RPGPGM. RPGPGM is called by DBGEX.

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
extern char EXPORTFLD[6];
void c_proc(int *size, char **ptr)
{
   *ptr = malloc(*size);
   memset(*ptr, 'P',*size );
   printf("import string: %6s.\n",EXPORTFLD);
}
```

Figure 67. Source for C Procedure cproc. cproc is called by DBGEX.

Chapter 11. Handling Exceptions

This chapter explains how ILE RPG/400 exception handling works, and how to use:

- Exception handlers
- ILE RPG/400-specific handlers
- ILE condition handlers.
- Cancel handlers.

ILE RPG/400 supports the following types of exception handlers:

- RPG-specific handlers, for example, the use of an error indicator or a *PSSR or INFSR error subroutine.
- ILE condition handlers, user-written exception handlers that you register at run time using the ILE condition handler bindable API CEEHDLR.
- ILE cancel handler which can be used when a procedure ends abnormally.

Most programs benefit from some sort of planned exception handling because it can minimize the number of unnecessary abnormal ends (namely, those associated with function checks). ILE condition handlers also allow you to handle exceptions in mixed-language applications in a consistent manner.

You can use the RPG exception handlers to handle most situations that might arise in a RPG application. The minimum level of exception handling which RPG provides is the use of error indicators on certain operations. To learn how to use them, read the following sections in this chapter:

- "ILE RPG/400 Exception Handling" on page 155
- "Specifying Error Indicators" on page 161

Additionally, to learn how ILE exception handling works, read:

- "Exception Handling Overview" (for general concepts)
- "Using RPG-Specific Handlers" on page 161
- The sections on error handling in *ILE Concepts*.

For information on exception handling and the RPG cycle, see *ILE RPG/400 Reference*.

Note: In this book the term 'exception handling' is used to refer to both exception handling and error handling. However, for consistency with other RPG terms, the term 'error' is used in the context of 'error indicator' and 'error subroutine'.

Exception Handling Overview

Exception handling is the process of:

- Examining an exception message which has been issued as a result of a run-time error
- Optionally modifying the exception to show that it has been received (i.e., handled)
- Optionally recovering from the exception by passing the exception information to a piece of code to take any necessary actions.

When a run-time error occurs, an exception message is generated. An exception message has one of the following types depending on the error which occurred:

***ESCAPE**	Indicates that a severe error has been detected.
***STATUS**	Describes the status of work being done by a program.
***NOTIFY**	Describes a condition requiring corrective action or reply from the calling program.
Function Check	Indicates that one of the three previous exceptions occurred and was not handled.

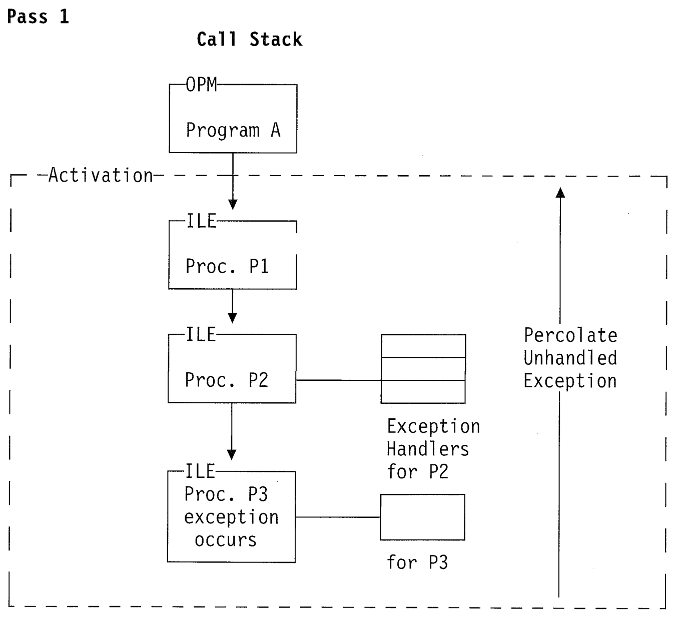
Exception messages are associated with call stack entries. Each call stack entry is in turn associated with a list of exception handlers defined for that entry. (See "The Call Stack" on page 92 for further discussion of a call stack.)

Figure 68 shows a call stack where an OPM program calls an ILE program consisting of several modules and therefore several procedures. Refer to this figure in the discussions which follow.

In general, when an exception occurs, the handlers associated with the call stack entry are given a chance to handle the exception. If the exception is not handled by any of the handlers on the list then it is considered to be unhandled, at which point the following default actions are taken for the unhandled exception:

1. If the exception is a function check, the call stack entry is removed from the stack.
2. The exception is moved (percolated) to the previous call stack entry.
3. The exception handling process is restarted for this call stack entry.

The action of allowing the previous call stack entry to handle an exception is referred to as **percolation**. Percolation continues until the exception is handled, or until the control boundary is reached. A **control boundary** is a call stack entry for which the immediately preceding call stack entry is in a different activation group **or** is an OPM program. In Figure 68 Procedure P1 is the control boundary.


Pass 1
Call Stack
OPM
Program A
Activation
ILE
Proc. P1
ILE
Proc. P2
ILE
Proc. P3 exception occurs
Exception Handlers for P2
for P3
Percolate Unhandled Exception

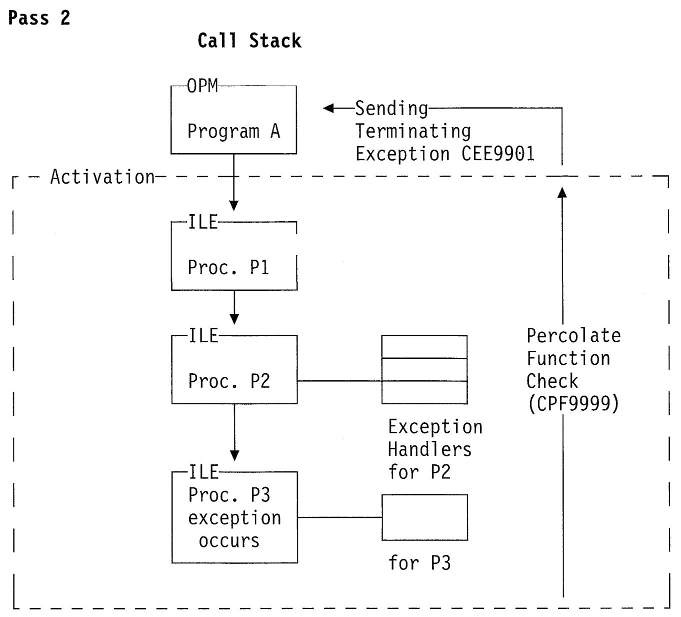

Pass 2
Call Stack
OPM
Program A
Sending Terminating Exception CEE9901
Activation
ILE
Proc. P1
ILE
Proc. P2
Percolate Function Check (CPF9999)
Exception Handlers for P2
ILE
Proc. P3 exception occurs
for P3

Figure 68 (Part 2 of 2). Call Stack and Exception Message Percolation

In OPM, the exception message is associated with the *program* which is active on the call stack. If the exception is not handled by the associated exception handlers, then a function check is sent to the same call stack entry which received the exception. If it remains unhandled, then the entry is removed and the function check is percolated. The process repeats until the exception is handled.

In ILE, an exception message is associated with the *procedure* which is active on the call stack. When the exception is percolated, it is *not* converted to a function check. Each call stack entry is given a chance to handle the original exception until the control boundary is reached. Only then is the exception converted to a function check, at which point the exception processing starts all over again beginning with the procedure which received the exception. This time each call stack entry is given a chance to handle the function check. If the control boundary is reached and the exception is still unhandled then a generic failure exception message CEE9901 is sent to the caller of the procedure at the control boundary. In addition, any call stack entry which did not to handle the message is removed.

ILE RPG/400 Exception Handling

ILE RPG/400 provides three types of exception handling mechanisms:

- An error indicator handler
- An error subroutine handler
- A default exception handler

RPG categorizes exceptions into two classes, program and file; this determines which type of error subroutine is called. Some examples of program exceptions are division by zero, out-of-bounds array index, or SQRT of a negative number. Some examples of file exceptions are undefined record type or a device error.

There are three ways for you to indicate that RPG should handle an exception. You can:

1. Specify an error indicator in positions 73 - 74 of the calculation specifications of the appropriate operation code.

2. Code a file error subroutine, which is defined by the INFSR keyword on a File Description specification, for file exceptions.
3. Code a program error subroutine, which is named *PSSR, for program exceptions.

Whenever an exception occurs, ILE RPG/400 does the following:

1. If an error indicator is present on the calculation specification and the exception is one that is expected for that operation:
   a. The indicator is set on
   b. The exception is handled
   c. Control resumes with the next ILE RPG/400 operation.
2. If no error indicator is present *and*
   - you have coded a *PSSR error subroutine and the exception is a program exception
     or
   - you have coded a INFSR error subroutine for the file and the exception is an I/O exception,

   then the exception will be handled and control will resume at the first statement of the error subroutine.
3. If no error indicator or error subroutine is coded, then the RPG default error handler is invoked.
   - If the exception is *not* a function check, then the exception will be percolated.
   - If the exception is a function check, then an inquiry message will be displayed. If the 'G' or 'R' option is chosen, the function check will be handled and control will resume at the appropriate point (*GETIN for 'G' or the same calculation specification that received the exception for 'R') in the procedure. Otherwise,the function check will be percolated and the procedure will be abnormally terminated.

See “Unhandled Exceptions” on page 158 for a full description of the RPG default handler.

Differences between OPM and ILE RPG/400 Exception Handling

For the most part, exception handling behaves the same in OPM RPG/400 and ILE RPG/400. The key difference lies in the area of unhandled exceptions.

In OPM, if an exception occurs and there is no RPG-specific handler enabled, then an inquiry message is issued. In ILE, this will only occur if the exception is a function check. If it is not, then the exception will be passed to the caller of the procedure or program, and any eligible higher call stack entries are given a chance to handle the exception. For example, consider the following example:

- PGM A calls PGM B, which in turn calls PGM C.
- PGM B has an error indicator coded for the call.
- PGM C has no error indicator or *PSSR error subroutine coded.
- PGM C gets an exception.

In OPM, an inquiry message would be issued for PGM C. In ILE, the exception is percolated to PGM B, since it is unhandled by PGM C. The error indicator in PGM

B is turned on allowing PGM B to handle the error, and in the process PGM C ends abnormally. There is no inquiry message.

If PGM C has a *PSSR error subroutine coded, then in both OPM and ILE, the exception is handled by PGM C and the error subroutine is run.

Note: Inquiry messages issued by ILE RPG/400 will start with the prefix 'RNQ', not 'RPG', as in OPM RPG/400.

Certain behavioral differences exist for some specific errors. See Appendix A, "Behavioral Differences Between OPM RPG/400 and ILE RPG/400" on page 301 for further information.

Using Exception Handlers

Planning the exception handling capability of your application means making the following decisions:

1. Decide if you will use the RPG-specific means of handling errors (e.g., error indicator or subroutine) or whether you will write a separate exception handling routine which you will register using the ILE API CEEHDLR. You might also choose to use both.
2. Decide on the recovery action, that is, where the program will resume processing if you use a separate exception handling routine.

In addition, keep in mind the following when planning your exception handlers:

- Priority of handlers
- Nested exceptions
- Default actions for unhandled exceptions
- Effect of optimization level

Exception Handler Priority

Exception handler priority becomes important if you use both language-specific error handling and ILE condition handlers. For an ILE RPG/400 procedure, exception handlers have the following priority:

1. Error indicator handler
2. I/O error subroutine handler
3. ILE condition handler
4. Program error subroutine handler
5. RPG default handler (for unhandled exceptions).

Nested Exceptions

Exceptions can be nested. A nested exception is an exception that occurs while another exception is being handled. When this happens, the processing of the first exception is temporarily suspended. Exception handling begins again with the most recently generated exception.

Unhandled Exceptions

An unhandled exception is one which has not been handled by an exception handler associated with the call stack entry which first received the exception. When an exception is unhandled, the ILE RPG/400 default handler takes one of the following actions:

If the message type is a function check (CPF9999) then the RPG default handler will issue an inquiry message describing the originating condition.

- If you pick the D(ump) or C(ancel) option then the procedure which first received the exception terminates and the function check is percolated to the caller.
- If you pick the R(etry) or G(et Input) option then the function check is handled, exception processing ends, and the procedure resumes processing at *GETIN (when G is chosen) or at the I/O operation in which the exception occurred (when R is chosen). For example, any read operation will be retried if the read failed because of record locking.

For other types of messages the exception is percolated up the call stack to the caller of the procedure. That procedure is presented with the exception and given a chance to handle it. If it does not, then the exception is percolated up the call stack until it reaches the control boundary, at which point the exception is converted to a function check, and exception handling starts over as described above.

Example of Unhandled Escape Message

The following scenario describes the events which occur when an escape message is issued and cannot be handled by the procedure in which it occurred. This scenario has the following assumptions:

1. There are two programs, PGM1 and PGM2 which run in the same activation group. Each contains a procedure, PRC1 and PRC2 respectively.
2. PRC1 calls PGM2 dynamically and PRC2 receives control.
3. The CALL operation code in PRC1 has an error indicator for the call.
4. No RPG exception handlers have been coded in PRC2. That is, there is no error indicator coded for the SUBST operation and there is no *PSSR error subroutine.
5. PRC2 has a SUBST operation where the Factor 1 entry is a negative number.

When PGM1 calls PGM2, and the SUBST operation is attempted, an exception message, RNX0100, is generated. Figure 69 depicts this scenario and the events which occur.

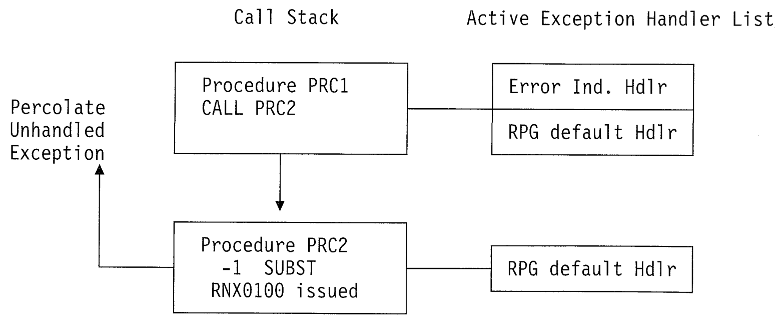


Figure 69. Scenario for Unhandled Escape Message

The following then occurs:

1. Since there is no error indicator or *PSSR error subroutine coded on the SUBST operation in PRC2, PRC2 cannot handle the program error, and so it is unhandled.
2. Since it is not a function check, it is percolated (passed up the call stack) to PRC1.
3. PRC1 receives (handles) the same exception message, and sets on the error indicator on the CALL operation with the side effect that PRC2 is terminated.
4. Processing then continues in PRC1 with the statement following the CALL operation.

Note: The same exception handling events described would apply to a procedure call (CALLB operation) as well.

Example of Unhandled Function Check

The following scenario describes the events which occur when a function check occurs and is not handled. This scenario has the following assumptions:

1. There are two programs, PGM1 and PGM2, each containing a procedure, PRC1 and PRC2 respectively.
2. PRC1 calls PGM2 dynamically and PRC2 receives control.
3. The CALL operation code in PRC1 does not have an error indicator coded.
4. No RPG exception handlers have been coded in PRC2. That is, there is no error indicator coded and there is no *PSSR error subroutine.
5. PRC2 has a pointer address error.

When PGM1 calls PGM2, a pointer error occurs because the basing pointer is defined as null. Consequently, MCH1306 is generated. A function check occurs when PRC2 tries to percolate the exception past the control boundary. Figure 70 on page 160 depicts this scenario and the events which occur.

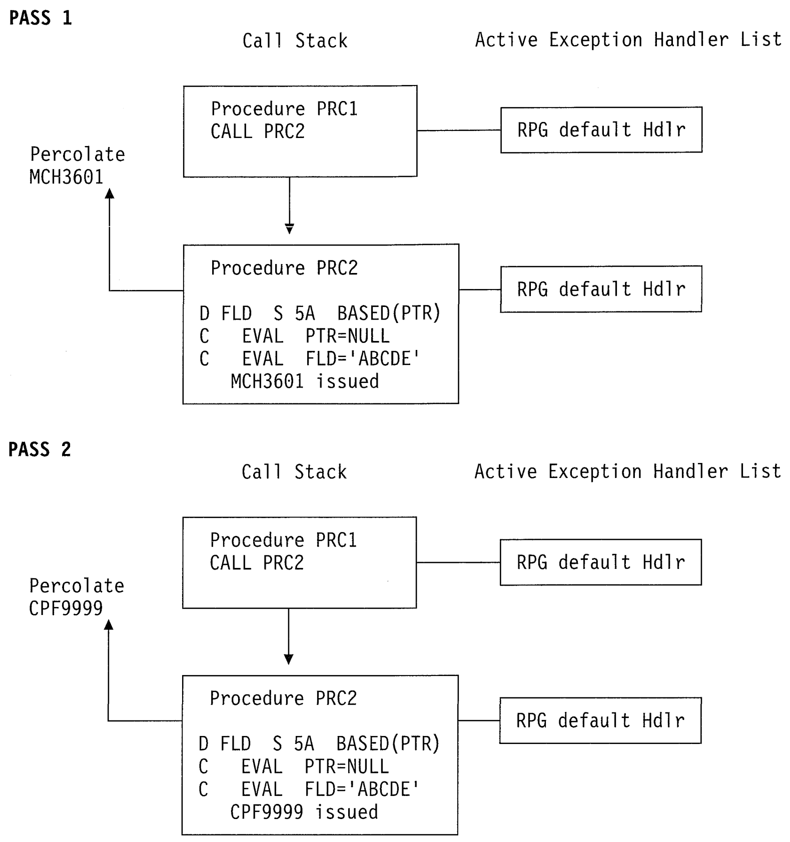


Figure 70. Scenario for Unhandled Function Check

The following then occurs:

1. Since there are no error indicators coded in PRC2, PRC2 cannot handle the function check, and so it is unhandled.
2. Since it is a function check, an inquiry message is issued describing the originating condition.
3. Depending on the response to the inquiry message, PRC2 may be terminated and the exception percolated to PRC1 (response is 'C') or processing may continue in PRC2 (response is 'G').

Optimization Considerations

While running a *FULL optimized program, the optimizer may keep frequently used values in machine registers and restore them to storage only at predefined points during normal program processing. Exception handling may break this normal processing and consequently program variables contained in registers may not be returned to their assigned storage locations.

Specifically, variables may not contain their current values if an exception occurs and you recover from it using one of:

- *PSSR error subroutine
- INFSR error subroutine
- User-defined exception handler
- The Go ('G') option from an inquiry message.
- The Retry ('R') option from an inquiry message.

ILE RPG/400 automatically defines indicators such that they contain their current values even with full optimization. To ensure that the content of fields or data structures contain their correct (current) values, specify the NOOPT keyword on the appropriate Definition specification.

For more information on the NOOPT keyword, see *ILE RPG/400 Reference*. For more information on optimization, see "Changing the Optimization Level" on page 59.

Using RPG-Specific Handlers

ILE RPG/400 provides three ways for you to enable HLL-specific handlers and to recover from the exception:

1. error indicators
2. INFSR error subroutine
3. *PSSR error subroutine

You can obtain more information about the error which occurred by coding the appropriate data structures and querying the relevant data structure fields.

This section provides some examples of how to use each of these RPG constructs. The *ILE RPG/400 Reference* provides more information on the *PSSR and INFSR error subroutines, on the EXSR operation code, and on the INFDS and PSDS data structures.

Specifying Error Indicators

To enable the RPG error indicator handler, you specify an error indicator in positions 73 and 74 in any of the following operation codes:

Table 10. Operation Codes Allowing an Error Indicator in Positions 73-74

ACQ	DSPLY	POST	SETGT
ADDDUR	EXFMT	READ (n)	SETLL
CALL	EXTRCT	READC	SUBDUR
CALLB	NEXT	READE (n)	SUBST (p)
CHAIN (n)	FEOD	READP (n)	TEST (x)
CHECK	IN	READPE (n)	UNLOCK
CHECKR	OCCUR	REL	UPDATE
CLOSE	OPEN	ROLBK	WRITE
COMMIT	ORxx	SCAN	XLATE (p)
DELETE	OUT		

If an exception occurs on the operation, the indicator is set on, the appropriate data structure (PSDS or INFDS) is updated, and control returns to the next sequential instruction. You can then test the indicator to determine what action to take.

When you specify an error indicator on an operation code, you can explicitly call a file error subroutine (INFSR) or a program error subroutine (*PSSR) with the EXSR operation. If either INFSR or *PSSR is explicitly called by the EXSR operation and Factor 2 of the ENDSR operation is blank or the field specified has a value of blank, control returns to the next sequential instruction following the EXSR operation.

Note: If an error indicator is coded on an operation, but the error which occurs is not related to the operation (for example, an array-index error on a CHAIN operation), any error indicator would be ignored. The error would be treated like any other program error.

Using an Error Subroutine

When you write a error subroutine you are doing two things:

1. Enabling the RPG subroutine error handler

   The subroutine error handler will handle the exception and pass control to your subroutine.

2. Optionally specifying a recovery action.

   You can use the error subroutine to take specific actions based on the error which occurred or you can have a generic action (for example, issuing an inquiry message for all errors).

The following considerations apply to error subroutines:

- You can explicitly call an error subroutine by specifying the name of the subroutine in Factor 2 of the EXSR operation.
- You can control the point where processing resumes by specifying a value in Factor 2 of the ENDSR operation of the subroutine.
- If an error subroutine is called the RPG error subroutine handler has already handled the exception. Thus, the call to the error subroutine reflects a return to program processing. If an exception occurs while the subroutine is running, the subroutine is called again. The procedure will loop unless you code the subroutine to avoid this problem.

  To see how to code an error subroutine to avoid such a loop, see “Avoiding a Loop in an Error Subroutine” on page 168.

Using a File Error (INFSR) Subroutine

To handle a file error or exception you can write a file error (INFSR) subroutine. When a file exception occurs:

1. The INFDS is updated.
2. A file error subroutine (INFSR) receives control if the exception occurs:
   - On an implicit (primary or secondary) file operation
   - On an explicit file operation that does not have an indicator specified in positions 73 - 74.

Note that a file error subroutine can handle errors in more than one file.

The following restrictions apply:

- If a file exception occurs during the start or end of a program, (for example, on an implicit open at the start of the cycle) control passes to the ILE RPG/400 default exception handler, and not to the error subroutine handler. Consequently, the file error subroutine will not be processed.
- If an error occurs that is not related to the operation (for example, an array-index error on a CHAIN operation), then any INFSR error subroutine would be ignored. The error would be treated like any other program error.

To add a file error subroutine to your program, you do the following steps:

1. Enter the name of the subroutine after the keyword INFSR on a File Description specification. The subroutine name can be *PSSR, which indicates that the program error subroutine is given control for the exception on this file.
2. Optionally identify the file information data structure on a File Description specification using the keyword INFDS.
3. Enter a BEGSR operation where the Factor 1 entry contains the same subroutine name that is specified for the keyword INFSR.
4. Identify a return point, if any, and code it on the ENDSR operation in the subroutine. For a discussion of the valid entries for Factor 2, see "Specifying a Return Point in the ENDSR Operation" on page 170.
5. Code the rest of the file error subroutine. While any of the ILE RPG/400 compiler operations can be used in the file error subroutine, it is not recommended that you use I/O operations to the same file that got the error. The ENDSR operation must be the last specification for the file error subroutine.

Figure 71 on page 164 shows an example of exception handling using an INFSR error subroutine. The program TRNSUPDT is a simple inventory update program. It uses a transaction file TRANSACT to update a master inventory file PRDMAS. If an I/O error occurs, then the INFSR error subroutine is called. If it is a record lock error, then the record is written to a backlog file. Otherwise, an inquiry message is issued.

Note that the File specification for PRDMAS identifies both the INFDS and identifies the INFSR to be associated with it.

The following is done for each record in the TRANSACT file:

1. The appropriate record in the product master file is located using the transaction product number.
2. If the record is found, then the quantity of the inventory is updated.
3. If an error occurs on the UPDATE operation, then control is passed to the INFSR error subroutine.
4. If the record is not found, then the product number is written to an error report.

```
  *=================================================================*
  * TRNSUPDT: This program is a simple inventory update program.    *
  * The transaction file (TRANSACT) is processed consecutively.     *
  * The product number in the transaction is used as key to access  *
  * the master file (PRDMAS) randomly.                              *
  * 1. If the record is found, the quantity of the inventory will   *
  *    be updated.                                                  *
  * 2. If the record is not found, an error will be printed on a    *
  *    report.                                                      *
  * 3. If the record is currently locked, the transaction will be   *
  *    written to a transaction back log file which will be         *
  *    processed later.                                             *
  * 4. Any other unexpected error will cause a runtime error        *
  *    message.                                                     *
  *=================================================================*

  *-----------------------------------------------------------------*
  * Define the files:                                               *
  *   1) PRDMAS      - Product master file                          *
  *   2) TRANSACT    - Transaction file                             *
  *   3) TRNBACKLG   - Transaction backlog file                      *
  *   2) PRINT       - Error report.                                *
  *-----------------------------------------------------------------*
FPRDMAS    UF   E           K DISK
F                                     INFSR(PrdInfsr)
F                                     INFDS(PrdInfds)
FTRANSACT  IP   E             DISK
FTRNBACKLG O    E             DISK
FPRINT     O    F   80        PRINTER

  *-----------------------------------------------------------------*
  * Define the file information data structure for file PRDMAS.     *
  * The *STATUS field is used to determine what action to take.     *
  *-----------------------------------------------------------------*
D PrdInfds        DS
D  PrdStatus        *STATUS

  *-----------------------------------------------------------------*
  * List of expected exceptions.                                    *
  *-----------------------------------------------------------------*
D ErrRecLock      C                   CONST(1218)
```

Figure 71 (Part 1 of 2). Example of File Exception Handling

```
      *-----------------------------------------------------------------*
      * Access the product master file using the transaction product    *
      * number.                                                         *
      *-----------------------------------------------------------------*
     C     TRNPRDNO      CHAIN     PRDREC                             10
      *-----------------------------------------------------------------*
      * If the record is found, update the quantity in the master file. *
      *-----------------------------------------------------------------*
     C                   IF        NOT *IN10
     C                   SUB       TRNQTY        PRDQTY
     C                   UPDATE    PRDREC
      *-----------------------------------------------------------------*
      * If the record is not found, write to the error report           *
      *-----------------------------------------------------------------*
     C                   ELSE
     C                   EXCEPT    NOTFOUND
     C                   ENDIF
     C                   SETON                                        LR
      *-----------------------------------------------------------------*
      * Error handling routine.                                         *
      *-----------------------------------------------------------------*
     C     PrdInfsr      BEGSR
      *-----------------------------------------------------------------*
      * If the master record is currently locked, write the transaction *
      * record to the back log file and skip to next transaction.       *
      *-----------------------------------------------------------------*
     C     PrdStatus     DSPLY
     C                   IF        (PrdStatus = ErrRecLock)
     C                   WRITE     TRNBREC
     C                   MOVE      '*GETIN'      ReturnPt          6
      *-----------------------------------------------------------------*
      * If unexpected error occurs, cause inquiry message to be issued. *
      *-----------------------------------------------------------------*
     C                   ELSE
     C                   MOVE      *BLANK        ReturnPt
     C                   ENDIF
     C                   ENDSR     ReturnPt

      *-----------------------------------------------------------------*
      * Error report format.                                            *
      *-----------------------------------------------------------------*
     OPRINT     E            NOTFOUND
     O                       TRNPRDNO
     O                                           29 'NOT IN PRDMAS FILE'
```

Figure 71 (Part 2 of 2). Example of File Exception Handling

When control is passed to the error subroutine, the following occurs:

- If the error is due to a record lock, then the record is written to a backlog file and control returns to the main part with the next transaction (via *GETIN as the return point).
- If the error is due to some other reason, then blanks are moved to ReturnPt. This will result in the RPG default handler receiving control. The recovery action at that point will depend on the nature of the error.

Note that the check for a record lock error is done by matching the *STATUS subfield of the INFDS for PRDMAS against the field ErrRecLock which is defined with the value of the record lock status code. The INFSR could be extended to handle other types of I/O errors by defining other errors, checking for them, and then taking an appropriate action.

Using a Program Error Subroutine

To handle a program error or exception you can write a program error subroutine (*PSSR). When a program error occurs:

1. The program status data structure is updated.
2. If an indicator is *not* specified in positions 73 and 74 for the operation code, the error is handled and control is transferred to the *PSSR.

   You can explicitly transfer control to a program error subroutine after a file error by specifying *PSSR after the keyword INFSR on the File Description specifications.

To add a *PSSR error subroutine to your program, you do the following steps:

1. Optionally identify the program status data structure (PSDS) by specifying an S in position 23 of the Definition specification.
2. Enter a BEGSR operation with a Factor 1 entry of *PSSR.
3. Identify a return point, if any, and code it on the ENDSR operation in the subroutine. For a discussion of the valid entries for Factor 2, see "Specifying a Return Point in the ENDSR Operation" on page 170.
4. Code the rest of the program error subroutine. Any of the ILE RPG/400 compiler operations can be used in the program error subroutine. The ENDSR operation must be the last specification for the program error subroutine.

Figure 72 on page 167 shows an example of a program error subroutine.

```
      *-----------------------------------------------------------------*
      *  Define relevant parts of program status data structure         *
      *-----------------------------------------------------------------*
     D Psds            SDS
     D  Loc              *ROUTINE
     D  Err              *STATUS
     D  Parms            *PARMS
     D  Name             *PROC

      *-----------------------------------------------------------------*
      *  BODY OF CODE GOES HERE                                         *
      *  An error occurs when division by zero takes place.             *
      *  Control is passed to the *PSSR subroutine.                     *
      *-----------------------------------------------------------------*

      *=================================================================*
      * *PSSR: Error Subroutine for the procedure.  We check for a      *
      *        division by zero error, by checking if the status is     *
      *        102.  If it is, we add 1 to the divisor and continue     *
      *        by moving *GETIN to ReturnPt.                            *
      *=================================================================*
     C     *PSSR         BEGSR

     C                   IF        Err = 102
     C                   ADD       1             Divisor
     C                   MOVE      '*GETIN'      ReturnPt          6

      *-----------------------------------------------------------------*
      *        An unexpected error has occurred, and so we move         *
      *        *CANCL to ReturnPt to end the procedure.                 *
      *-----------------------------------------------------------------*
     C                   ELSE
     C                   MOVE      '*CANCL'      ReturnPt
     C                   ENDIF

     C                   ENDSR     ReturnPt
```

*Figure 72. Example of *PSSR Subroutine*

The program-status data structure is defined on the Definition specifications. The predefined subfields *STATUS, *ROUTINE, *PARMS, and *PROGRAM are specified, and names are assigned to the subfields.

The *PSSR error subroutine is coded on the calculation specifications. If a program error occurs, ILE RPG/400 passes control to the *PSSR error subroutine. The subroutine checks to determine if the exception was caused by a divide operation in which the divisor is zero. If it was, 1 is added to the divisor (Divisor), and the literal '*DETC' is moved to the field ReturnPt, to indicate that the program should resume processing at the beginning of the detail calculations routine

If the exception was not a divide by zero, the literal '*CANCL' is moved into the ReturnPt field, and the procedure ends.

Avoiding a Loop in an Error Subroutine

In the previous example, it is unlikely that an error would occur in the *PSSR and thereby cause a loop. However, depending on how the *PSSR is written, loops may occur if an exception occurs while processing the *PSSR.

One way to avoid such a loop is to set a first-time switch in the subroutine. If it is not the first time through the subroutine, you can specify an appropriate return point, such as *CANCL, for the Factor 2 entry of the ENDSR operation.

Figure 73 shows a program NOLOOP which is designed to generate exceptions in order to show how to avoid looping within a *PSSR subroutine. The program generates an exception twice:

1. In the main body of the code, to pass control to the *PSSR
2. Inside the *PSSR to potentially cause a loop.

```
 *=================================================================*
 * NOLOOP:  Show how to avoid recursion in a *PSSR subroutine.     *
 *=================================================================*
FQSYSPRT   O    F  132        PRINTER

 *-----------------------------------------------------------------*
 * Array that will be used to cause an error                       *
 *-----------------------------------------------------------------*
D Arr1            S             10A   DIM(5)

 *-----------------------------------------------------------------*
 * Generate an array out of bounds error to pass control to *PSSR. *
 *-----------------------------------------------------------------*
C                   Z-ADD     -1            Neg1              5 0
C                   MOVE      Arr1(Neg1)    Arr1(Neg1)
C                   MOVE      *ON           *INLR

 *=================================================================*
 * *PSSR: Error Subroutine for the procedure.  We use the          *
 *        variable InPssr to detect recursion in the PSSR.         *
 *        If we detect recursion, then we *CANCL the procedure.    *
 *=================================================================*
C     *PSSR         BEGSR

C                   EXCEPT    In_Pssr

C                   IF        InPssr = 1
C                   MOVE      '*CANCL'      ReturnPt          6
C                   EXCEPT    Cancelling
C                   Z-ADD     0             InPssr            1 0

C                   ELSE
C                   Z-ADD     1             InPssr
```

Figure 73 (Part 1 of 2). Avoiding a Loop in an Error Subroutine

```
 *                                                                *
 *       We now generate another error in the PSSR to see         *
 *       how the subroutine cancels the procedure.                *
 *                                                                *
C                  MOVE      Arr1(Neg1)    Arr1(Neg1)
 *                                                                *
 *       Note that the next two operations will not be            *
 *       processed if Neg1 is still negative.                     *
 *                                                                *
C                  MOVE      '*GETIN'      ReturnPt
C                  Z-ADD     0             InPssr
C                  ENDIF

C                  ENDSR     ReturnPt

 *================================================================*
 * Procedure Output - for purposes of example                     *
 *================================================================*
OQSYSPRT   E           In_Pssr
O                                            'In PSSR'
OQSYSPRT   E           Cancelling
O                                            'Cancelling...'
```

Figure 73 (Part 2 of 2). Avoiding a Loop in an Error Subroutine

To create the program, using the source in Figure 73 on page 168, type:

```
CRTBNDRPG PGM(MYLIB/NOLOOP)
```

When you call the program, the following occurs:

1. An exception occurs when the program tries to do a MOVE operation on an array using a negative index. Control is passed to the *PSSR.
2. Since this is the first time through the *PSSR, the variable In_Pssr is not already set on. To prevent a future loop, the variable In_Pssr is set on.
3. Processing continues within the *PSSR with the MOVE after the ELSE. Again, an exception occurs and so processing of the *PSSR begins anew.
4. This time through, the variable In_Pssr is already set to 1. Since this indicates that the subroutine is in a loop, the procedure is canceled by setting the ReturnPt field to *CANCL.
5. The ENDSR operation receives control, and the procedure is canceled.

The output which the program produces (again, for the purposes of illustration) is shown below:

```
In PSSR
In PSSR
Cancelling...
```

The approach used here to avoid looping can also be used within an INFSR error subroutine.

Specifying a Return Point in the ENDSR Operation

When using an INFSR or *PSSR error subroutine, you can indicate the return point at which the program will resume processing, by entering one of the following as the Factor 2 entry of the ENDSR statement. The entry must be a six-position character field, literal, named constant, array element, or table name whose value specifies one of the following return points.

Note: If the return points are specified as literals, they must be enclosed in apostrophes and entered in uppercase (for example, *DETL, not *detl). If they are specified in fields or array elements, the value must be left-adjusted in the field or array element.

***DETL**	Continue at the beginning of detail lines.
***GETIN**	Continue at the get input record routine.
***TOTC**	Continue at the beginning of total calculations.
***TOTL**	Continue at the beginning of total lines.
***OFL**	Continue at the beginning of overflow lines.
***DETC**	Continue at the beginning of detail calculations.
***CANCL**	Cancel the processing of the program.
Blanks	Return control to the ILE RPG/400 default exception handler. This will occur when Factor 2 is a value of blanks *and* when Factor 2 is not specified. If the subroutine was called by the EXSR operation and Factor 2 is blank, control returns to the next sequential instruction.

After the ENDSR operation of the INFSR or the *PSSR subroutine is run, the ILE RPG/400 compiler resets the field or array element specified in Factor 2 to blanks. Because Factor 2 is set to blanks, you can specify the return point within the subroutine that is best suited for the exception that occurred.

If this field contains blanks at the end of the subroutine, the ILE RPG/400 default exception handler receives control following the running of the subroutine, unless the INFSR or the *PSSR subroutine was called by the EXSR operation. If the subroutine was called by the EXSR operation and Factor 2 of the ENDSR operation is blank, control returns to the next sequential instruction following the EXSR operation.

ILE Condition Handlers

ILE condition handlers are exception handlers that are registered at run time using the Register ILE Condition Handler (CEEHDLR) bindable API. They are used to handle, percolate or promote exceptions. The exceptions are presented to the condition handlers in the form of an ILE condition. You can register more than one ILE condition handler. ILE condition handlers may be unregistered by calling the Unregister ILE Condition Handler (CEEHDLU) bindable API.

When you use an ILE condition handler, you are responsible for modifying the exception message to indicate that it has been handled. If you do not, then the system will consider the message to be unhandled. To modify an message, refer to the Change Exception Message (QMHCHGEM) API in the System API Reference.

There are several reasons why you might want to use an ILE condition handler:

- You can bypass language-specific handling by handling the exception in your own handler.

  This enables you to provide the same exception handling mechanism in an application with modules in different ILE HLLs.

- You can use this API to scope exception handling to a call stack entry.

  In ILE RPG/400, the ILE bindable API CEEHDLR is scoped to the procedure that contains it. It remains in effect until you unregister it, or until the procedure returns.

  Note: Any call to the CEEHDLR API from any detail, total or subroutine calculation will make the condition handler active for the entire procedure, including all input, calculation, and output operations.

For information on how to use ILE condition handlers, refer to *ILE Concepts*.

Using a Condition Handler

The following example shows you how to:

1. Code a condition handler to handle the RPG 'out-of-bounds' error
2. Register a condition handler
3. Deregister a condition handler
4. Code a *PSSR error subroutine.

The example consists of two procedures:

- RPGHDLR, which consists of a user-written condition handler for out-of-bound substring errors
- SHOWERR, which tests the RPGHDLR procedure.

While SHOWERR is designed primarily to show how RPGHDLR works, the two procedures combined are also useful for determining 'how' ILE exception handling works. Both procedures write to QSYSPRT the 'actions' which occur as they are processed. You might want to modify these procedures in order to simulate other aspects of ILE exception handling which you would like to explore.

Figure 74 on page 172 shows the source for the procedure RPGHDLR. The procedure defines three procedure parameters: an ILE condition token structure, a field to contain the name of the procedure which contains the error, and a field to contain the possible actions, resume or percolate. (RPGHDLR does not promote any exceptions).

The basic logic of RPGHDLR is the following:

1. Test to see if it is an out-of-bounds error by testing the message ID
   - If it is, it writes 'Handling...' to QSYSPRT and then sets the action to 'Resume'.
   - If it is not, then it writes out 'Percolating' to QSYSPRT, and then sets the action to 'Percolate'.
2. Return.

```
 *=================================================================*
 * RPGHDLR: RPG exception handling procedure.                      *
 *          This procedure does the following:                     *
 *          Handles the exception if it is the RPG                 *
 *             out of bounds error (RNX0100)                       *
 *          otherwise                                              *
 *             percolates the exception                            *
 *          It also prints out what it has done.                   *
 *                                                                 *
 * Note:    This is the exception handling procedure for the       *
 *          SHOWERR procedure.                                     *
 *=================================================================*
FQSYSPRT   O    F  132        PRINTER

 *-----------------------------------------------------------------*
 * Procedure parameters                                            *
 * 1. Input: Condition token structure                             *
 * 2. Input: Procedure name in which error occurred                *
 * 3. Output: Code identifying actions to be performed on the      *
 *            exception                                            *
 * 4. Output: New condition if we decide to promote the            *
 *            condition.  Since this handler only resumes and      *
 *            percolates, we will ignore this parameter.           *
 *-----------------------------------------------------------------*
D CondTok         DS
D  MsgSev                        4B 0
D  MsgNo                         2A
D                                1A
D  MsgPrefix                     3A
D                                4A

D ProcName        S             10A

D Action          S              9B 0
 *
 * Action codes are:
 *
D Resume          C                   10
D Percolate       C                   20

 *-----------------------------------------------------------------*
 * Parameters                                                      *
 *-----------------------------------------------------------------*
C     *ENTRY        PLIST
C                   PARM                    CondTok
C                   PARM                    ProcName
C                   PARM                    Action
```

Figure 74 (Part 1 of 2). Source for Condition Handler for Out-of-Bounds Substring Error

```
C*---------------------------------------------------------------*
C*      If substring error, then handle else percolate.          *
C*      Note that the message number value (MsgNo) is in hex.    *
C*---------------------------------------------------------------*
C                   EXCEPT
C                   IF        MsgPrefix = 'RNX' AND
C                             MsgNo     = X'0100'
C                   EXCEPT    Handling
C                   EVAL      Action    = Resume
C                   ELSE
C                   EXCEPT    Perclating
C                   EVAL      Action    = Percolate
C                   ENDIF
C                   RETURN

 *===============================================================*
 * Procedure Output                                              *
 *===============================================================*
OQSYSPRT   E
O                                       'In Handler for     '
O                       ProcName
OQSYSPRT   E            Handling
O                                       ' Handling...'
OQSYSPRT   E            Perclating
O                                       ' Percolating..    .'
```

Figure 74 (Part 2 of 2). Source for Condition Handler for Out-of-Bounds Substring Error

Figure 75 on page 174 shows the source for the procedure SHOWERR, in which the condition handler RPGHDLR is registered.

The procedure parameters include a procedure pointer to RPGHDLR and a pointer to the procedure's program status data structure. In addition, it requires a definition for the error-prone array ARR1, and identification of the parameter lists used by the ILE bindable APIs CEEHDLR and CEEHDLU.

The basic logic of the program is as follows:

1. Register the handler RPGHDLR using the subroutine RegHndlr. This subroutine calls the CEEHDLR API, passing it the procedure pointer to RPGHDLR.
2. Generate an out-of-bounds substring error.

   The handler RPGHDLR is automatically called. It handles the exception, and indicates that processing should resumes in the next *machine* instruction following the error. Note that the next machine instruction may not be at the beginning of the next RPG operation.
3. Generate an out-of-bounds array error.

   Again, RPGHDLR is automatically called. However, this time it cannot handle the exception, and so it percolates it to the next exception handler associated with the procedure, namely, the *PSSR error subroutine.

   The *PSSR cancels the procedure.
4. Unregister the condition handler RPGHDLR via a call to CEEHDLU.
5. Return

As with the RPGHDLR procedure, SHOWERR writes to QSYSPRT to show what is occurring as it is processed.

```
 *=================================================================*
 * SHOWERR:       Show exception handling using a user-defined     *
 *                exception handler.                               *
 *=================================================================*
FQSYSPRT   O    F  132        PRINTER

 *----------------------------------------------------------------*
 * The following are the parameter definitions for the CEEHDLR    *
 * API.  The first is the procedure pointer to the                *
 * procedure which will handle the exception.  The second         *
 * is a pointer to a communication area which will be passed      *
 * to the exception handling procedure.  In this example,         *
 * we will pass a pointer to the name of this procedure           *
 *----------------------------------------------------------------*
D pConHdlr        S               *   PROCPTR
D                                     INZ(%paddr('RPGHDLR'))

 *----------------------------------------------------------------*
 * PSDS                                                           *
 *----------------------------------------------------------------*
D DSPsds         SDS                  NOOPT
D   ProcName        *PROC

 *----------------------------------------------------------------*
 * Array that will be used to cause an error                      *
 *----------------------------------------------------------------*
D Arr1            S             10A   DIM(5)

 *----------------------------------------------------------------*
 * CEEHDLR Interface                                              *
 *----------------------------------------------------------------*
C     CEEHDLR_PL    PLIST
C                   PARM                    pConHdlr
C                   PARM                    ProcName
C                   PARM                    *OMIT

 *----------------------------------------------------------------*
 * CEEHDLU Interface                                              *
 *----------------------------------------------------------------*
C     CEEHDLU_PL    PLIST
C                   PARM                    pConHdlr
C                   PARM                    *OMIT

 *----------------------------------------------------------------*
 * Register the handler and generate errors                       *
 *----------------------------------------------------------------*
C                   EXSR      RegHndlr

C*----------------------------------------------------------------*
C*      Generate a substring error                                *
C*----------------------------------------------------------------*
C                   Z-ADD     -1            Neg1              5 0
C     Neg1          SUBST     'Hello'       Examp            10
```

Figure 75 (Part 1 of 3). Source for Registering a Condition Handler

```
 *-----------------------------------------------------------------*
 *      The exception was handled by the handler and control      *
 *      resumes here.                                              *
 *-----------------------------------------------------------------*
C                   EXCEPT    ImBack

 *-----------------------------------------------------------------*
 *      Generate an array out of bounds error                      *
 *-----------------------------------------------------------------*
C                   MOVE      Arr1(Neg1)    Arr1(Neg1)

C*-----------------------------------------------------------------*
C*      The exception was not handled by the handler, so,          *
C*      control does not return here.  The exception is            *
C*      percolated and control resumes in the *PSSR.               *
C*-----------------------------------------------------------------*

 *-----------------------------------------------------------------*
 *      Deregister the handler                                     *
 *      Note: If an exception occurs before the handler is         *
 *      deregistered, it will be automatically deregistered        *
 *      when the procedure is cancelled.                           *
 *-----------------------------------------------------------------*
C                   EXSR      DeRegHndlr
C
C                   SETON                                        LR

 *=================================================================*
 * RegHdlr - Call the API to register the Handler                  *
 *=================================================================*
C     RegHndlr      BEGSR
C                   CALLB     'CEEHDLR'     CEEHDLR_PL

C                   ENDSR

 *=================================================================*
 * DeRegHndlr - Call the API to unregister the Handler             *
 *=================================================================*
C     DeRegHndlr    BEGSR

C                   CALLB     'CEEHDLU'     CEEHDLU_PL

C                   ENDSR

 *=================================================================*
 * *PSSR: Error Subroutine for the procedure                       *
 *=================================================================*
C     *PSSR         BEGSR

C                   EXCEPT    InPssr
C                   EXCEPT    Cancelling

C                   ENDSR     '*CANCL'
```

Figure 75 (Part 2 of 3). Source for Registering a Condition Handler

```
      *=================================================================*
      * Procedure Output                                                *
      *=================================================================*
     OQSYSPRT   E           ImBack
     O                                            'I''m Back'
     OQSYSPRT   E           InPssr
     O                                            'In PSSR'
     OQSYSPRT   E           Cancelling
     O                                            'Cancelling...'
```

Figure 75 (Part 3 of 3). Source for Registering a Condition Handler

If you want to try these procedures, follow these steps:

1. To create the procedure RPGHDLR, using the source shown in Figure 74 on page 172, type:

   ```
   CRTRPGMOD MODULE(MYLIB/RPGHDLR)
   ```

2. To create the procedure SHOWERR, using the source shown in Figure 75 on page 174, type:

   ```
   CRTRPGMOD MODULE(MYLIB/SHOWERR)
   ```

3. To create the program, ERRORTEST, type

   ```
   CRTPGM PGM(MYLIB/ERRORTEST) MODULE(SHOWERR RPGHDLR)
   ```

4. To run the program ERRORTEST, type:

   ```
   OVRPRTF FILE(QSYSPRT) SHARE(*YES)
   CALL PGM(MYLIB/ERRORTEST)
   ```

   The output is shown below:

   ```
    In Handler for SHOWERR
     Handling...
    I'm Back
    In Handler for SHOWERR
     Percolating...
    In PSSR
    Cancelling...
   ```

Using Cancel Handlers

Cancel handlers provide an important function by allowing you to get control for clean-up and recovery actions when call stack entries are terminated by something other than a normal return. For example, you might want one to get control when a procedure ends via a system request '2', or because an inquiry message was answered with 'C' (Cancel).

A cancel handler may be enabled around a body of code inside a procedure. When a cancel handler is enabled around a certain body of code inside a procedure it only gets control if the suspend point of the call stack entry is inside that code, and the call stack entry is cancelled.

The Register Call Stack Entry Termination User Exit Procedure (CEERTX) and the Unregister Call Stack Entry Termination User Exit Procedure (CEETUTX) ILE bindable APIs provide a way of dynamically registering a user-defined routine to be run when the call stack entry for which it is registered is cancelled. The System API Reference contains information on these ILE bindable APIs.

Chapter 12. Obtaining a Dump

This chapter describes how to obtain an ILE RPG/400 formatted dump and provides a sample formatted dump.

Obtaining an ILE RPG/400 Formatted Dump

To obtain an ILE RPG/400 formatted dump (printout of storage) for a procedure while it is running, you can:

- Code one or more DUMP operation codes in your Calculation specifications
- Respond to a run-time message with a D or F option. It is also possible to automatically reply to make a dump available. Refer to the "System Reply List" discussion in the *CL Programming* book.

The formatted dump includes field contents, data structure contents, array and table contents, the file information data structures, and the program status data structure of the procedure. The dump is written to the file called QPPGMDMP. (A system abnormal dump is written to the file QPSRVDMP.)

If you respond to an ILE RPG/400 run-time message with an F option, the dump also includes the hexadecimal representation of the open data path (ODP, a data management control block).

If a program consists of more than one procedure, the information in the formatted dump reflects information about the *procedure* which was active at the time of the dump request.

Note: To obtain a dump of variable data, the program object must have debug data. That is, it must be created any debug view except *NONE.

Using the DUMP Operation Code

You can code one or more DUMP operation codes in your source to obtain a ILE RPG/400 formatted dump. The DUMP operations can be coded at any point or at several points in the calculation specifications. The output records are written whenever the DUMP operation occurs.

You should know the following characteristics of the DUMP operation code before using it:

- The DUMP operation runs (is active) only if keyword DEBUG(*YES) is specified on the control specification. If the keyword is not specified, or if DEBUG(*NO) is specified, the DUMP operation is checked for errors and the statement is printed on the listing, but the DUMP is not processed
- If the DUMP operation is conditioned, it occurs only if the condition is met.
- If a DUMP operation is bypassed by a GOTO operation, the DUMP operation does not occur.

See *ILE RPG/400 Reference* for a detailed description of the DUMP operation.

Example of a Formatted Dump

The following figures show an example of a formatted dump of a procedure similar to DBGEX. (DBGEX is one of the two procedures in the program DEBUGEX which is discussed in Chapter 10, "Debugging Programs" on page 113.) In order to show how data buffers are handled in a formatted dump we added the output file QSYSPRT.

The dump for this example is a full-formatted dump; that is, it was created when an inquiry message was answered with an 'F'.

Program Status Information:

```
Program Status Area:
Procedure Name . . . . . . . . . . . . :   DBGEX2      
Program Name . . . . . . . . . . . . . :   TEST             A
  Library . . . . . . . . . . . . . . . :   MYLIB
Module Name. . . . . . . . . . . . . . :   DBGEX2
Program Status . . . . . . . . . . . . :   00202       B
Previous Status  . . . . . . . . . . . :   00000       C
Statement in Error . . . . . . . . . . :   00000088    D
RPG Routine  . . . . . . . . . . . . . :   RPGPGM      E
Number of Parameters . . . . . . . . . :
Message Type . . . . . . . . . . . . . :   MCH         F
Additional Message Info  . . . . . . . :   4431
Message Data . . . . . . . . . . . . . :
              Program signature violation.
Status that caused RNX9001 . . . . . . :   0000
Last File Used . . . . . . . . . . . . :
Last File Status . . . . . . . . . . . :                    G
Last File Operation  . . . . . . . . . :
Last File Routine  . . . . . . . . . . :
Last File Statement  . . . . . . . . . :
Last File Record Name  . . . . . . . . :
Job Name . . . . . . . . . . . . . . . :   QPADEV0010
User Name  . . . . . . . . . . . . . . :   MYUSERID
Job Number . . . . . . . . . . . . . . :   002273
Date Entered System  . . . . . . . . . :   09/30/1994
Date Started . . . . . . . . . . . . . :   *N/A*            H
Time Started . . . . . . . . . . . . . :   *N/A*
Compile Date . . . . . . . . . . . . . :   093094
Compile Time . . . . . . . . . . . . . :   153438
Compiler Level . . . . . . . . . . . . :   0001
Source File  . . . . . . . . . . . . . :   QRPGLESRC
  Library . . . . . . . . . . . . . . . :   MYLIB
Member . . . . . . . . . . . . . . . . :   DBGEX2
```

Figure 76. Program Status Information section of Formatted Dump

A Procedure Identification: the procedure name, the program and library name, and the module name.

B Current status code.

C Previous status code.

D ILE RPG/400 source statement in error.

E ILE RPG/400 routine in which the exception or error occurred.

F CPF or MCH for a machine exception.

G Information about the last file used in the program before an exception or error occurred. In this case, no files were used.

H Program information. '*N/A*' indicates fields for which information is not available in the program. These fields are only updated if they are included in the PSDS.

Feedback Areas:

```
INFDS FILE FEEDBACK  I
File . . . . . . . . . . . . . . . . . :   QSYSPRT
File Open  . . . . . . . . . . . . . . :   YES
File at EOF  . . . . . . . . . . . . . :   NO
File Status  . . . . . . . . . . . . . :   00000
File Operation . . . . . . . . . . . . :   OPEN I
File Routine . . . . . . . . . . . . . :   *INIT
Statement Number . . . . . . . . . . . :   *INIT
Record Name  . . . . . . . . . . . . . :
Message Identifier . . . . . . . . . . :

OPEN FEEDBACK  J
ODP type . . . . . . . . . . . . . . . :   SP
File Name  . . . . . . . . . . . . . . :   QSYSPRT
   Library . . . . . . . . . . . . . . :   QSYS
Member . . . . . . . . . . . . . . . . :   Q501383525
Spool File . . . . . . . . . . . . . . :   Q04079N002
   Library . . . . . . . . . . . . . . :   QSPL
Spool File Number  . . . . . . . . . . :   7
Primary Record Length  . . . . . . . . :   80
Input Block Length . . . . . . . . . . :   0
Output Block Length  . . . . . . . . . :   80
Device Class . . . . . . . . . . . . . :   PRINTER
Lines per Page . . . . . . . . . . . . :   66
Columns per Line . . . . . . . . . . . :   132
Allow Duplicate Keys . . . . . . . . . :   *N/A*  K
Records to Transfer  . . . . . . . . . :   1
Overflow Line  . . . . . . . . . . . . :   60
Block Record Increment . . . . . . . . :   0
File Sharing Allowed . . . . . . . . . :   NO
Device File Created with DDS . . . . . :   NO
IGC or graphic capable file. . . . . . :   NO
File Open Count. . . . . . . . . . . . :   1
Separate Indicator Area. . . . . . . . :   NO
User Buffers . . . . . . . . . . . . . :   NO
Open Identifier. . . . . . . . . . . . :   Q04079N002
Maximum Record Length. . . . . . . . . :   0
ODP Scoped to Job. . . . . . . . . . . :   NO
Maximum Program Devices. . . . . . . . :   1
Current Program Device Defined . . . . :   1
Device Name  . . . . . . . . . . . . . :   *N
Device Description Name. . . . . . . . :   *N
Device Class . . . . . . . . . . . . . :   '02'X
Device Type. . . . . . . . . . . . . . :   '08'X

COMMON I/O FEEDBACK  L
Number of Puts . . . . . . . . . . . . :   0
Number of Gets . . . . . . . . . . . . :   0
Number of Put/Gets . . . . . . . . . . :   0
Number of other I/O  . . . . . . . . . :   0
Current Operation  . . . . . . . . . . :   '00'X
Record Format  . . . . . . . . . . . . :
Device Class and Type. . . . . . . . . :   '0208'X
Device Name  . . . . . . . . . . . . . :   *N
Length of Last Record  . . . . . . . . :   80
Number of Records Retrieved. . . . . . :   80
Last I/O Record Length . . . . . . . . :   0
Current Block Count. . . . . . . . . . :   0

PRINTER FEEDBACK:
Current Line Number. . . . . . . . . . :   1
Current Page . . . . . . . . . . . . . :   1
Major Return Code. . . . . . . . . . . :   00
Minor Return Code. . . . . . . . . . . :   00

Output Buffer:
   0000   00000000  00000000  00000000  00000000  00000000  00000000  00000000  00000000     *                                *
   0020   00000000  00000000  00000000  00000000  00000000  00000000  00000000  00000000     *                                *
   0040   00000000  00000000  00000000  00000000                                             *                                *
```

Figure 77. Feedback Areas section of Formatted Dump

I This is the file feedback section of the INFDS. Only fields applicable to the file type are printed. The rest of the INFDS feedback sections are not dumped, since they are only updated if they have been declared in the program.

K This is the file open feedback information for the file. See *Data Management* for a description of the fields.

L This is the common I/O feedback information for the file. See *Data Management* for a description of the fields.

Information with Full-Formatted Dump:

```
Open Data Path:
  0000   64800000 00001AF0 00001B00 000000B0 00000140 000001C6 00000280 000002C0    *       0               F        *
  0020   00000530 00000000 00000000 00000380 00000000 06000000 00000000 00000000    *                                *
  0040   00008000 00000000 003AC02B A00119FF 000006C0 00003033 00000000 00000000    *                                *
  0060   80000000 00000000 003AC005 CF001CB0 00000000 00000000 00000000 00000000    *                                *
  0080   80000000 00000000 003AA024 D0060120 01900000 00010000 00000050 00000000    *                           &    *
  00A0   1F000000 00000000 00000000 00000000 E2D7D8E2 E8E2D7D9 E3404040 D8E2E8E2    *                SPQSYSPRT   QSYS*
  00C0   40404040 4040D8F0 F4F0F7F9 D5F0F0F2                                        *      Q04079N002QSPL         & *
Open Feedback:
  0000   E2D7D8E2 E8E2D7D9 E3404040 D8E2E8E2 40404040 4040D8F0 F4F0F7F9 D5F0F0F2    *SPQSYSPRT   QSYS      Q04079N002*
  0020   D8E2D7D3 40404040 40400007 00500000 D8F5F0F1 F3F8F3F5 F2F50000 00000000    *QSPL         &  Q501383525      *
  0040   00500002 00000000 42008400 00000000 0000D5A4 00100000 00000008 00000000    * &        d       Nu            *
  0060   00000000 00000000 00000100 3C000000 0005E000 5CD54040 40404040 40400001    *                    *N          *
  0080   00000000 00001300 00000000 00000000 00010001 5CD54040 40404040 40400000    *                    *N          *
  00A0   07100000 00000000 00450045 00450045 07A10045 00450045 00700045 00450045    *                                *
  00C0   00450045 00450045 002F0030 00040005 5CD54040 40404040 40400208 00000000    *                *N              *
  00E0   20000000 00000000 00000000 00000000 00000000 00000001 C2200000 00059A00    *                        B       *
  0100   00000000 00000000 00000000 00000000 00000000 4040                          *                                *
Common I/O Feedback:
  0000   00900000 00000000 00000000 00000000 00000000 00000000 00000000 00000208    *                                *
  0020   5CD54040 40404040 40400000 00500000 00000000 00000000 00000000 00000000    **N           &                  *
  0040   00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000    *                                *
  0060   00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000    *                                *
  0080   00000000 00000000 00000000 00000000                                        *                                *
I/O Feedback for Device:
  0000   00010000 00010000 00000000 00000000 00000000 00000000 00000000 00000000    *                                *
  0020   0000F0F0 0001                                                              *  0000                          *
```

Figure 78. Information Provided for Full-Formatted Dump

The common open data path and the feedback areas associated with the file are included in the dump if you respond to an ILE RPG/400 inquiry message with an F option.

Data Information:

```
ILE RPG/400 FORMATTED DUMP
Module Name. . . . . . . . . . . . . . :   DBGEX
Optimization Level . . . . . . . . . . :   *NONE        M

N
Halt Indicators:
H1 '0'   H2 '0'   H3 '0'   H4 '0'   H5 '0'   H6 '0'   H7 '0'   H8 '0'   H9 '0'
Command/Function Key Indicators:
KA '0'   KB '0'   KC '0'   KD '0'   KE '0'   KF '0'   KG '0'   KH '0'   KI '0'   KJ '0'
KK '0'   KL '0'   KM '0'   KN '0'   KP '0'   KQ '0'   KR '0'   KS '0'   KT '0'   KU '0'
KV '0'   KW '0'   KX '0'   KY '0'
Control Level Indicators:
L1 '0'   L2 '0'   L3 '0'   L4 '0'   L5 '0'   L6 '0'   L7 '0'   L8 '0'   L9 '0'
Overflow Indicators:
OA '0'   OB '0'   OC '0'   OD '0'   OE '0'   OF '0'   OG '0'   OV '0'
External Indicators:
U1 '0'   U2 '0'   U3 '0'   U4 '0'   U5 '0'   U6 '0'   U7 '0'   U8 '0'
General Indicators:
01 '0'   02 '1'   03 '0'   04 '1'   05 '0'   06 '1'   07 '0'   08 '0'   09 '0'   10 '0'
11 '0'   12 '0'   13 '0'   14 '0'   15 '0'   16 '0'   17 '0'   18 '0'   19 '0'   20 '0'
21 '0'   22 '0'   23 '0'   24 '0'   25 '0'   26 '0'   27 '0'   28 '0'   29 '0'   30 '0'
31 '0'   32 '0'   33 '0'   34 '0'   35 '0'   36 '0'   37 '0'   38 '0'   39 '0'   40 '0'
41 '0'   42 '0'   43 '0'   44 '0'   45 '0'   46 '0'   47 '0'   48 '0'   49 '0'   50 '0'
51 '0'   52 '0'   53 '0'   54 '0'   55 '0'   56 '0'   57 '0'   58 '0'   59 '0'   60 '0'
61 '0'   62 '0'   63 '0'   64 '0'   65 '0'   66 '0'   67 '0'   68 '0'   69 '0'   70 '0'
71 '0'   72 '0'   73 '0'   74 '0'   75 '0'   76 '0'   77 '0'   78 '0'   79 '0'   80 '0'
81 '0'   82 '0'   83 '0'   84 '0'   85 '0'   86 '0'   87 '0'   88 '0'   89 '0'   90 '0'
91 '0'   92 '0'   93 '0'   94 '0'   95 '0'   96 '0'   97 '0'   98 '0'   99 '0'
Internal Indicators:
LR '0'   MR '0'   RT '0'   1P '0'

O
NAME                     ATTRIBUTES             VALUE
_QRNU_DSI_DS1            BIN(9,0)               1            P
_QRNU_DSI_DS2            BIN(9,0)               2
_QRNU_TABI_TABLEA        BIN(9,0)               1
ARRY                     ZONED(3,2)             DIM(2)
                           (1-2)                1.23
BASEONNULL               CHAR(10)               NOT ADDRESSABLE
BASEPTR                  POINTER                SPP:C01947001218
BASESTRING               CHAR(6)                'ABCDEF'
BIGDATE                  DATE(10)               '1994-09-30'
BIGTIME                  TIME(8)                '12.00.00'
BIGTSTAMP                TIMESTAMP(26)          '9999-12-31-12.00.00.000000'
BIN4D3                   BIN(4,3)               -4.321
BIN9D7                   BIN(9,7)               98.7654321
DBCSSTRING               GRAPHIC(3)             ' BBCCDD '
```

Figure 79 (Part 1 of 2). Data section of Formatted Dump

```
 DS1                 DS                  OCCURS(3)  Q
   OCCURRENCE(1)
     FLD1            CHAR(5)             '1BCDE'
     FLD1A           CHAR(1)             DIM(5)
                       (1)               '1'
                       (2)               'B'
                       (3)               'C'
                       (4)               'D'
                       (5)               'E'
     FLD2            BIN(5,2)            123.45
   OCCURRENCE(2)
     FLD1            CHAR(5)             '1BCDE'
     FLD1A           CHAR(1)             DIM(5)
                       (1)               '1'
                       (2)               'B'
                       (3)               'C'
                       (4)               'D'
                       (5)               'E'
     FLD2            BIN(5,2)            123.45
   OCCURRENCE(3)
     FLD1            CHAR(5)             '1BCDE'
     FLD1A           CHAR(1)             DIM(5)
                       (1)               '1'
                       (2)               'B'
                       (3)               'C'
                       (4)               'D'
                       (5)               'E'
     FLD2            BIN(5,2)            123.45
 DS2                 CHAR(10)            DIM(2)     R
                       (1)               'aaaaaaaaaa'
                       (2)               'bbbbbbbbbb'
 DS3                 DS                            S
   FIRSTNAME         CHAR(10)            'Fred      '
                     VALUE IN HEX        'C6998584404040404040'X
   LASTNAME          CHAR(10)            'Jones     '
                     VALUE IN HEX        'D1969585A24040404040'X
   TITLE             CHAR(5)             'Mr.  '
                     VALUE IN HEX        'D4994B4040'X
 EXPORTFLD           CHAR(6)             'export'
 NULLPTR             POINTER             SYP:*NULL
 PACKED1D0           PACKED(5,2)         -093.40
 PARM1               PACKED(4,3)         6.666
 PROCPTR             POINTER             PRP:A00CA02EC200  T
 SPCPTR              POINTER             SPP:A026FA0100C0
 SPCSIZ              BIN(9,0)            000000008.
 STRING              CHAR(6)             'ABCDEF'
 TABLEA              CHAR(3)             DIM(3)
                       (1)               'aaa'
                       (2)               'bbb'
                       (3)               'ccc'
 ZONEDD3D2           ZONED(3,2)          -3.21
```

Figure 79 (Part 2 of 2). Data section of Formatted Dump

M Optimization level

N General indicators 1-99 and their current status ('1' is on, '0' is off). Note that indicators *IN02, *IN04, and *IN06 were not yet set.

O Beginning of user variables, listed in alphabetical order.

P Internally-defined fields which contain index for tables and multiple-occurrence data structures.

Q Multiple-occurrence data structure.

R Data structures with no subfields are displayed as character strings.

S Data structure subfields are listed in *alphabetical order, not in the order in which they are defined.* Gaps in the subfield definitions are not shown.

T Attribute does not differentiate between basing and procedure pointer.

Working with Files and Devices

This section describes how to use files and devices in ILE RPG/400 programs. Specifically, it shows how to:

- Associate a file to a device
- Define a file (as program-described or externally-described)
- Process files
- Access database files
- Access externally-attached devices
- Write an interactive application

Note: The term 'RPG IV program' refers to an ILE program containing one or more procedures written in RPG IV.

Chapter 13. Defining Files

Files serve as the connecting link between a program and the device used for I/O. Each file on the system has an associated file description which describes the file characteristics and how the data associated with the file is organized into records and fields.

In order for a program to perform any I/O operations, it must identify the file description(s) the program is referencing, what type of I/O device is being used, and how the data is organized. This chapter provides general information on:

- associating file descriptions with input/output devices
- defining externally-described files
- defining program-described files
- data management operations

Information on how to use externally- and program-described files with different device types is found in subsequent chapters.

Associating Files with Input/Output Devices

The key element for all I/O operations on the AS/400 is the file. The system supports the following file types:

database files	allow storage of data permanently on system
device files	allow access to externally-attached devices. Include display files, printer files, tape files, diskette files, and ICF files.
save files	used to store saved data on disk
DDM files	allow access to data files stored on remote systems.

Each I/O device has a corresponding file description of one of the above types which the program uses to access that device.

The actual device association is made when the file is processed: the data is read from or written to the device when the file is used for processing.

To indicate to the operating system which file description(s) your program will use, you specify a *file name* in positions 7 through 16 of a file description specification for each file used. In positions 36 through 42 you specify an RPG *device name*. The device name defines which RPG operations can be used with the associated file. The device name can be one of: DISK, PRINTER, WORKSTN, SEQ, or SPECIAL. Figure 80 shows a file description specification for a display (WORKSTN) file FILEX.

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... *
FFilename++IPEASFRlen+LKlen+AIDevice+.Keywords+++++++++++++++++++++++++++++
FFILEX     CF   E             WORKSTN
```

Figure 80. Identifying a Display File in an RPG Program

Note that it is the file name, not the device name (specified in positions 36 through 42) which points to the OS/400 file description that contains the specifications for the actual device.

The RPG device types correspond to the above file types as follows:

Table 11. Correlation of RPG Device Types with AS/400 File Types

RPG Device Type	AS/400 File Type
DISK	database, save, DDM files
PRINTER	printer files
WORKSTN	display, ICF files
SEQ	tape, diskette, save, printer, database
SPECIAL	N/A

Figure 81 illustrates the association of the RPG file name FILEX, as coded in Figure 80 on page 185, with a system file description for a display file.

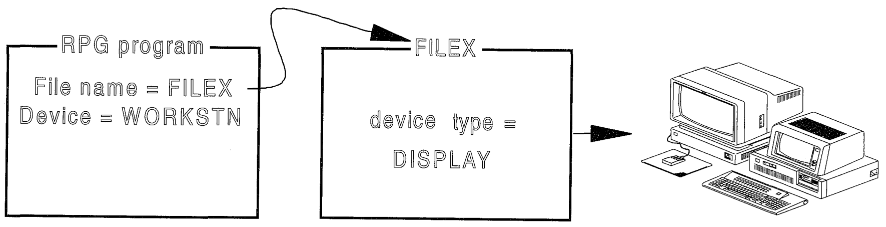


Figure 81. Associating a file name with a display file description

At compilation time, certain RPG operations are valid only for a specific RPG device name. In this respect, the RPG operation is device dependent. One example of device dependency is that the EXFMT operation code is valid only for a WORKSTN device.

Other operation codes are device independent, meaning that they can be used with any device type. For example, WRITE is a device-independent operation.

The SEQ Device The device SEQ is an independent device type. Figure 82 illustrates the association of the RPG file name FILEY with a system file description for a sequential device. When the program is run, the actual I/O device is specified in the description of FILEY. For example, the device might be PRINTER.

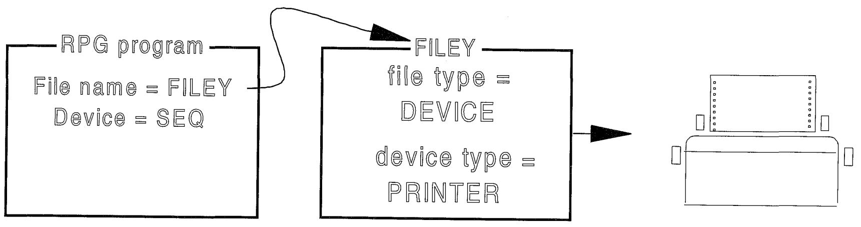


Figure 82. Associating a file name with a display file description

Although the file name and file type are coded in the RPG program, in many cases you can change the type of file or the device used in a program without changing the program. To find out how, see “Overriding and Redirecting File Input and Output” on page 197.

Naming Files

On the AS/400 system, files are made up of members. These files are organized into libraries. The convention for naming files is `library-name/file-name`.

In an ILE RPG/400 program, file names are identified in positions 7 through 16 in file description specifications. File names can be up to ten characters long and must be unique.

You do not qualify the file name with a library within a program. At run time, the system searches the library list associated with your job to find the file. If you wish to change the name, member, or specify a particular library, you can use a file override command. See "Overriding and Redirecting File Input and Output" on page 197 for more information on file overrides.

Types of File Descriptions

When identifying the file description your program will be using, you must indicate whether it is a program-described file or an externally-described file.

- For a **program-described file**, the description of the fields are coded within the RPG source member on input and/or output specifications.

  The description of the file to the operating system includes information about where the data comes from and the length of the records in the file.

- For an **externally-described file**, the compiler retrieves the description of the fields from an external file-description which was created using DDS, IDDU, or SQL/400 commands. Therefore, you do not have to code the field descriptions on input and/or output specifications within the RPG source member.

  The external description includes information about where the data comes from, such as the database or a specific device, and a description of each field and its attributes. The file must exist and be accessible from the library list before you compile your program.

Externally-described files offer the following advantages:

- Less coding in programs. If the same file is used by many programs, the fields can be defined once to the operating system and used by all the programs. This practice eliminates the need to code input and output specifications for RPG programs that use externally-described files.
- Less maintenance activity when the file's record format is changed. You can often update programs by changing the file's record format and then recompiling the programs that use the files without changing any coding in the program.
- Improved documentation because programs using the same files use consistent record-format and field names.
- Improved reliability. If level checking is specified, the RPG program will notify the user if there are changes in the external description. See "Level Checking" on page 195 for further information.

If an externally-described file (identified by an E in position 22 of the file description specification) is specified for the devices SEQ or SPECIAL, the RPG program uses the field descriptions for the file, but the interface to the operating system is as though the file were a program-described file. Externally-described files cannot specify device-dependent functions such as forms control for PRINTER files because this information is already defined in the external description.

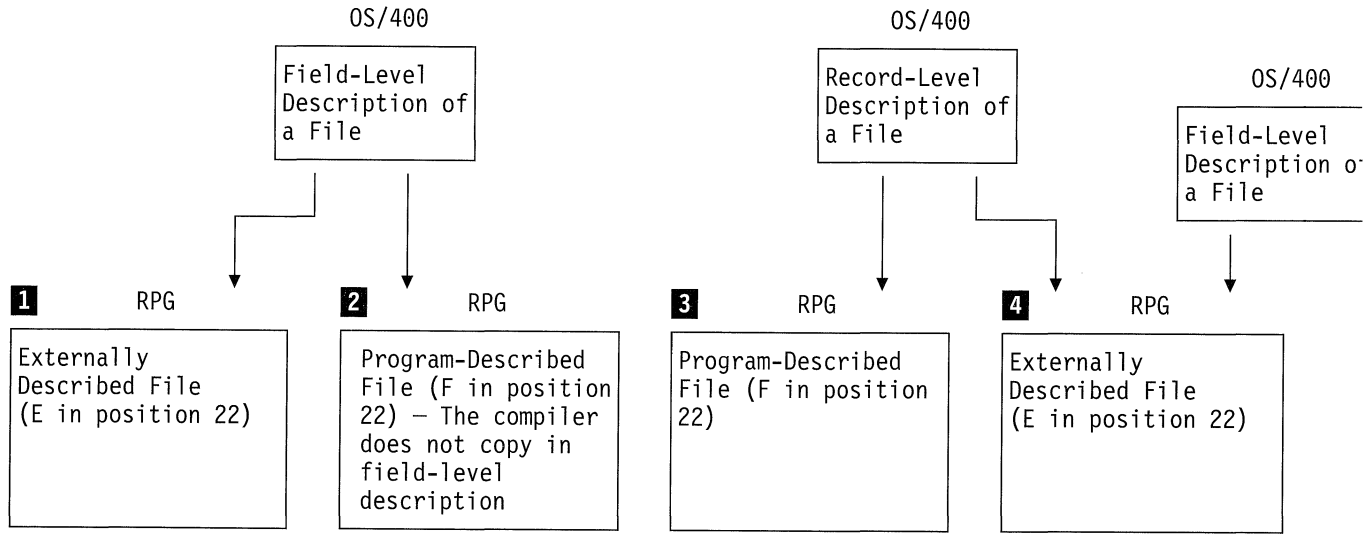
Using Files with External-Description as Program-Described

A file created from external descriptions can be used as a program-described file in the program. To use an externally-described file as a program-described file,

1. Specify the file as program-described (F in position 22) in the file description specification of the program.
2. Describe the fields in the records on the input or/and output specifications of the program.

At compile time, the compiler uses the field descriptions in the input or/and output specifications. It does not retrieve the external descriptions.

Example of Some Typical Relationships between Programs and Files



OS/400
Field-Level Description of a File
OS/400
Record-Level Description of a File
OS/400
Field-Level Description of a File
1 RPG
Externally Described File (E in position 22)
2 RPG
Program-Described File (F in position 22) – The compiler does not copy in field-level description
3 RPG
Program-Described File (F in position 22)
4 RPG
Externally Described File (E in position 22)

Figure 83. Typical Relationships between an RPG Program and Files on the AS/400 System

1 The program uses the field-level description of a file that is defined to the operating system. An externally-described file is identified by an E in position 22 of the file description specifications. At compilation time, the compiler copies in the external field-level description.

2 An externally-described file (that is, a file with field-level external description) is used as a program-described file in the program. A program-described file is identified by an F in position 22 of the file description specifications. This entry tells the compiler not to copy in the external field-level descriptions. This file does not have to exist at compilation time.

3 A file is described only at the record level to the operating system. The fields in the record are described within the program; therefore, position 22 of the file description specifications must contain an F. This file does not have to exist at compilation time.

4 A file name can be specified at compilation time (that is, coded in the RPG source member), and a different file name can be specified at run time. The E in position 22 of the file description specifications indicates that the external description of the file is to be copied in at compilation time. At run time, a file override command can be used so that a different file is accessed by the program. To override a file at run time, you must make sure that record names in both files are the same. The RPG program uses the record-format name on the input/output operations, such as a READ operation where it specifies what record type is expected. See "Overriding and Redirecting File Input and Output" on page 197 for more information.

Defining Externally-Described Files

You can use DDS to describe files to the OS/400 system. Each record type in the file is identified by a unique record-format name.

An E entry in position 22 of the file description specifications identifies an externally-described file. The E entry indicates to the compiler that it is to retrieve the external description of the file from the system when the program is compiled.

The information in this external description includes:

- File information, such as file type, and file attributes, such as access method (by key or relative record number)
- Record-format description, which includes the record format name and field descriptions (names, locations, and attributes).

The information the compiler retrieves from the external description is printed on the compiler listing as long as OPTION(*EXPDDS) is specified on either the CRTRPGMOD or CRTBNDRPG command when compiling the source member. (The default for both of these commands is OPTION(*EXPDDS).)

The following section describes how to use a file description specification to rename or ignore record formats and how to use input and output specifications to modify external descriptions. Remember that input and output specifications for externally described files are optional.

Renaming Record-Format Names

Many of the functions that you can specify for externally-described files (such as the CHAIN operation) operate on either a file name or a record-format name. Consequently, each file and record-format name in the program must be a unique symbolic name.

To rename a record-format name, use the RENAME keyword on the file description specifications for the externally-described file as shown in Figure 84. The format is RENAME(*old name*:*new name*).

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... *
FFilename++IPEASFRlen+LKlen+AIDevice+.Keywords+++++++++++++++++++++++++++++
FITMMSTL   IP   E           K DISK    RENAME(ITEMFORMAT:MSTITM)
 *
```

Figure 84. RENAME Keyword for Record Format Names in an Externally-Described File

The RENAME keyword is generally used if the program contains two files which have the same record-format names. In Figure 84, the record format ITEMFORMAT in the externally-described file ITMMSTL is renamed MSTITM for use in this program.

Ignoring Record Formats

If a record format in an externally-described file is not to be used in a program, you can use the IGNORE keyword to make the program run as if the record format did not exist in the file. For logical files, this means that all data associated with that format is inaccessible to the program. Use the IGNORE keyword on a file description specifications for the externally-described file as shown in Figure 85.

The file must have more than one record format, and not all of them can be ignored; at least one must remain. Except for that requirement, any number of record formats can be ignored for a file.

Once a record-format is ignored, it cannot be specified for any other keyword (SFILE, RENAME, or INCLUDE), or for another IGNORE.

Ignored record-format names appear on the cross-reference listing, but they are flagged as ignored.

To indicate that a record format from an externally-described file, is to be ignored, enter the keyword and parameter IGNORE(*record-format name*) on the file description specification in the Keyword field.

Alternatively, the INCLUDE keyword can be used to include only those record format names that are to be used in a program. All other record formats contained in the file will be excluded.

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... *
FFilename++IPEASFRlen+LKlen+AIDevice+.Keywords+++++++++++++++++++++++++++++
 *
 *  Assume the file ITMMSTL contains the following record formats:
 *  EMPLNO, NAME, ADDR, TEL, WAGE.  To make the program run as if only the
 *  EMPLNO and NAME records existed, either of the following two methods
 *  can be used:
 *
FITMMSTL   UF   E           K DISK    IGNORE(ADDR:TEL:WAGE)
 *
 *  OR:
 *
FITMMSTL   UF   E           K DISK    INCLUDE(EMPLNO:NAME)
 *
```

Figure 85. IGNORE Keyword for Record Formats in an Externally-Described File

Using Input Specifications to Modify an External Description

You can use the input specifications to override certain information in the external description of an input file or to add RPG functions to the external description. On the input specifications, you can:

- Assign record-identifying indicators to record formats as shown in Figure 86 on page 191.

- Rename a field as shown in Figure 86 on page 191.
- Assign control-level indicators to fields as shown in Figure 86.
- Assign match-field values to fields for matching record processing as shown in Figure 87 on page 192.
- Assign field indicators as shown in Figure 87 on page 192.

You cannot use the input specifications to override field locations in an externally-described file. The fields in an externally-described file are placed in the records in the order in which they are listed in the data description specifications. Also, device-dependent functions such as forms control, are not valid in an RPG program for externally-described files.

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... *
IRcdname+++....In.........................................................*
IMSTRITEM      01 1
I..............Ext-field+..................Field+++++++++L1M1..PlMnZr......
I              ITEMNUMB 2                  ITEM          L1 3
 *
IMSTRWHSE      02
I              ITEMNUMB                    ITEM          L1
 *
```

Figure 86. Overriding and Adding RPG Functions to an External Description

1 To assign a record-identifying indicator to a record in an externally-described file, specify the record-format name in positions 7 through 16 of the input specifications and assign a valid record-identifying indicator in positions 21 and 22. A typical use of input specifications with externally-described files is to assign record-identifying indicators.

In this example, record-identifying indicator 01 is assigned to the record MSTRITEM and indicator 02 to the record MSTRWHSE.

2 To rename a field in an externally-described record, specify the external name of the field, left-adjusted, in positions 21 through 30 of the field-description line. In positions 49 through 62, specify the name that is to be used in the program.

In this example, the field ITEMNUMB in both records is renamed ITEM for this program.

3 To assign a control-level indicator to a field in an externally-described record, specify the name of the field in positions 49 through 62 and specify a control-level indicator in positions 63 and 64.

In this example, the ITEM field in both records MSTRITEM and MSTRWHSE is specified to be the L1 control field.

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... *
IFilename++SqNORiPos1+NCCPos2+NCCPos3+NCC.................................
IMSTREC       01 1
I.............Ext-field+.................Field+++++++++L1M1..PlMnZr......
I                                        CUSTNO          M1 1
 *
IWKREC        02
I                                        CUSTNO          M1
I                                        BALDUE                  98 2
 *
```

Figure 87. Adding RPG Functions to an External Description

1 To assign a match value to a field in an externally-described record, specify the record-format name in positions 7 through 16 of the record-identification line. On the field-description line specify the name of the field in positions 49 through 62 and assign a match-level value in positions 65 and 66.

In this example, the CUSTNO field in both records MSTREC and WKREC is assigned the match-level value M1.

2 To assign a field indicator to a field in an externally described record, specify the record-format name in positions 7 through 16 of the record-identification line. On the field-description line, specify the field name in positions 49 through 62, and specify an indicator in positions 69 through 74.

In this example, the field BALDUE in the record WKREC is tested for zero when it is read into the program. If the field's value is zero, indicator 98 is set on.

Using Output Specifications

Output specifications are optional for an externally-described file. RPG supports file operation codes such as WRITE and UPDATE that use the external record-format description to describe the output record without requiring output specifications for the externally described file.

You can use output specification to control when the data is to be written, or to specify selective fields that are to be written. The valid entries for the field-description line for an externally-described file are output indicators (positions 21 - 29), field name (positions 30 - 43), and blank after (position 45). Edit words and edit codes for fields written to an externally-described file are specified in the DDS for the file. Device-dependent functions such as fetch overflow (position 18) or space/skip (positions 40 - 51) are not valid in an RPG program for externally-described files. The overflow indicator is not valid for externally described files either. For a description of how to specify editing in the DDS, see the *DDS Reference*.

If output specifications are used for an externally-described file, the record-format name is specified in positions 7 - 16 instead of the file name.

If all the fields in an externally-described file are to be placed in the output record, enter *ALL in positions 30 through 43 of the field-description line. If *ALL is specified, you cannot specify other field description lines for that record.

If you want to place only certain fields in the output record, enter the field name in positions 30 through 43. The field names you specify in these positions must be the field names defined in the external record description, unless the field was renamed on the input specifications. See Figure 88 on page 193.

You should know about these considerations for using the output specifications for an externally-described file:

- In the output of an update record, only those fields specified in the output field specifications and meeting the conditions specified by the output indicators are placed in the output record to be rewritten. Fields not specified in the output specifications are rewritten using the values that were read. This technique offers a good method of control as opposed to the UPDATE operation code that updates all fields.
- In the creation of a new record, the fields specified in the output field specifications are placed in the record. Fields not specified in the output field specifications or not meeting the conditions specified by the output indicators are written as default values, which depend on the data format specified in the external description (for example: a blank for character fields; zero for numeric fields).

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... *
OFilename++DF..N01N02N03Excnam++++B++A++Sb+Sa+...........................*
OITMREC    D    20
O..............N01N02N03Field+++++++++YB.End++PConstant/editword/DTformat++
O                        *ALL 1
 *
OSLSREC    D    30
O                        SLSNAM 2
O                        COMRAT
O                   15   BONUS
 *
```

Figure 88. Output Specifications for an Externally-Described File

1 For an update file, all fields in the record are written to the externally-described record ITMREC using the current values in the program for all fields in the record.

For the creation of a new record, all fields in the record are written to the externally-described record ITMREC using the current values in the program for the fields in the record.

2 To update a record, the fields SLSNAM and COMRAT are written to the externally-described record SLSREC when indicator 30 is on. The field BONUS is written to the SLSREC record when indicators 30 and 15 are on. All other fields in the record are written with the values that were read.

To create a new record, the fields SLSNAM and COMRAT are written to the externally-described record SLSREC when indicator 30 is on. The field BONUS is written when indicators 30 and 15 are on. All other fields in the record are written as default values, which depend on their data type (for example: a blank for character fields; zero for numeric fields).

Handling Floating-Point Fields

The ILE RPG/400 program does not support the use of floating-point fields. If you process an externally-described file with floating-point fields, the following happens:

- You cannot access the floating-point fields.
- When you create a new record, the floating-point fields in the record have the value zero.
- When you update existing records, the floating-point fields are unchanged.
- If you attempt to use a floating-point field as a key field, your program receives a compile-time error.

Handling Variable-Length Fields

The ILE RPG/400 program does not support the use of variable-length character or graphic fields. However, if CVTOPT(*VARCHAR) or CVTOPT(*VARGRAPHIC) is specified on the CRTRPGMOD or CRTBNDRPG command, the ILE RPG/400 compiler will recognize variable-length character or graphic fields in an external file description and bring them into the program as fixed-length character fields. The first two bytes of the field will contain the variable length, and the remaining bytes will contain the character data.

Note: When using these fields it is your responsibility to process the data according to how the compiler has placed the data in the fixed-length field. Furthermore, you must make sure that the first two bytes, which contain the variable length, are properly initialized when new records are created or updated.

If CVTOPT(*VARCHAR) or CVTOPT(*VARGRAPHIC) is not specified on one of the create commands, the compiler will ignore the field in the external description and the program will not be able to access the field.

When you create a new record, the first two bytes of the field will be set to 0 and the rest of the field will be set to blank.

When you update an existing record, the variable-length fields are unchanged.

If you attempt to use a variable-length field as a key field, your program receives a compile-time error.

Handling Null Values

The ILE RPG/400 program does not support the use of null values. However, if ALWNULL(*YES) is specified on the CRTRPGMOD or CRTBNDRPG command, the ILE RPG/400 compiler will accept records with null-value fields in input-only files. Null values will be read in as default values, which depend on the data format specified in the external description (for example: a blank for character fields; zero for numeric fields).

If an attempt is made to retrieve a record containing a null value, and ALWNULL(*YES) is not specified on one of the create commands, a data-mapping error occurs and no data in the record is accessible to the ILE RPG/400 program.

Level Checking

Because HLL programs are dependent on receiving, at run-time, an externally-described file whose format agrees with what was copied into the program at compilation time, the system provides a level-check function that ensures that the format is the same.

The RPG compiler always provides the information required by level checking when an externally-described DISK, WORKSTN, or PRINTER file is used. The level-check function can be requested on the create, change, and override file commands. The default on the create file command is to request level checking.

Level checking occurs on a record-format basis when the file is opened unless you specify LVLCHK(*NO) when you issue a file override command or create a file. If the level-check values do not match, the program is notified of the error. The RPG program then handles the OPEN error as described in Chapter 11, "Handling Exceptions" on page 153.

The RPG program does not provide level checking for program-described files or for files using the devices SEQ or SPECIAL.

For more information on how to specify level checking, see *Data Management*.

Defining Program-Described Files

Program-described files are files whose records and fields are described on input/output specifications in the program that uses the file. To use a program-described file in an RPG program you must:

1. Identify the file(s) in the file description specifications.
2. If it is an input file, describe the record and fields in the input specifications. The file name in positions 7 through 16 in the input specifications must be the same as the corresponding name entered in the file specifications.

   On the record-identification entries you indicate whether you want to perform sequence checking of records within the file.
3. Enter the same file name as in step 1 in the FACTOR 2 field of those calculation specifications which require it. For example, WRITE operations to a program-described file require a data structure name in the result field.
4. If it is an output file, describe the record and fields in the output specifications. In addition, you specify how the output is to be printed. The file name in positions 7 through 16 in the output specifications must be the same as the corresponding name entered in the file specifications.

A program-described file must exist on the system, and be in your library list, before the program can run. To create a file, use one of the Create File commands, which can be found in the *CL Reference*.

Data Management Operations and ILE RPG/400 I/O Operations

Data management is the part of the operating system that controls the storing and accessing of data by an application program. Table 12 shows the data management operations provided by the AS/400 and their corresponding ILE RPG/400 operation. It also shows which operations are allowed for which ILE RPG/400 device type.

Table 12. Data Management Operations and the Corresponding RPG I/O Operation

Data Management Operation	ILE RPG/400 I/O Operation
OPEN	OPEN
READ	
By relative record number	READ, CHAIN
By key	READ, READE, CHAIN, primary file
Sequential	READ
Previous	READP, READPE
Next	READ, READE
Invited Device	READ
WRITE-READ	EXFMT
WRITE	
By relative record number	WRITE
By key	WRITE, EXCEPT, primary file
Sequential	WRITE, EXCEPT
FEOD	FEOD
UPDATE	
By relative record number	UPDATE, primary file
By key	UPDATE, primary file
DELETE	
By relative record number	DELETE, primary file
By key	DELETE, primary file
ACQUIRE	ACQ
RELEASE	REL
COMMIT	COMMIT
ROLLBACK	ROLBK
CLOSE	CLOSE, LR RETURN

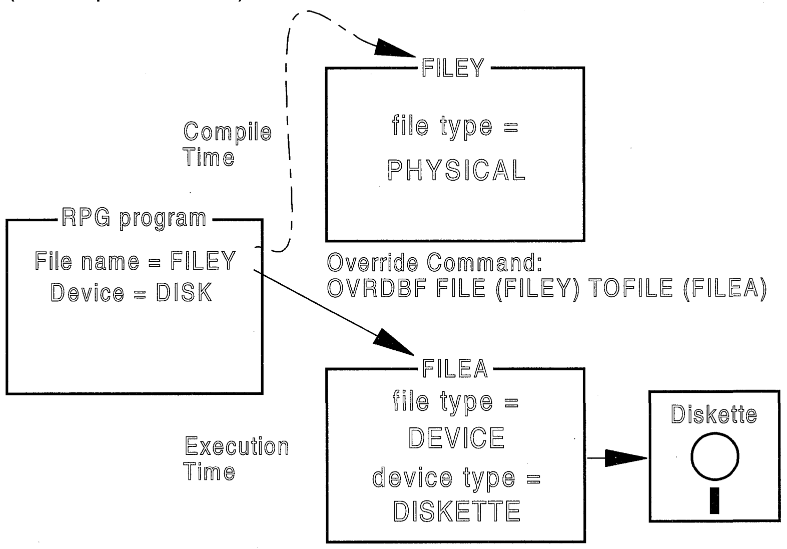
Chapter 14. General File Considerations

This chapter provides information on the following aspects of file processing on the AS/400 using RPG:

- overriding and redirecting file input and output
- file locking by an RPG program
- record locking by an RPG program
- sharing an open data path
- AS/400 spooling functions
- unblocking and blocking records to improve performance
- using SRTSEQ/ALTSEQ in an RPG program vs a DDS file

Overriding and Redirecting File Input and Output

OS/400 commands can be used to override a parameter in the specified file description or to redirect a file at compilation time or run time. File redirection allows you to specify a file at run time to replace the file specified in the program (at compilation time):



In the preceding example, the CL command OVRDBF (Override With Database File) allows the program to run with an entirely different device file than was specified at compilation time.

To override a file at run time, you must make sure that record names in both files are the same. The RPG program uses the record-format name on the input/output operations, such as a READ operation where it specifies what record type is expected.

Not all file redirections or overrides are valid. At run time, checking ensures that the specifications within the RPG program are valid for the file being processed. The OS/400 system allows some file redirections even if device specifics are contained in the program. For example, if the RPG device name is PRINTER, and the actual file the program connects to is not a printer, the OS/400 system ignores the RPG print spacing and skipping specifications. There are other file redirections that the OS/400 system does not allow and that cause the program to end. For example, if the RPG device name is WORKSTN and the EXFMT operation is specified in the program, the program is stopped if the actual file the program connects to is not a display or ICF file. In ILE, overrides are scoped to the activation group level, job level, or call level. Overrides that are scoped to the activation group level remain in effect until they are deleted, replaced, or until the activation group in which they are specified ends. Overrides that are scoped to the job level remain in effect until they are deleted, replaced, or until the job in which they are specified ends. This is true regardless of the activation group in which the overrides were specified. Overrides that are scoped to the call level remain in effect until they are deleted, replaced, or until the program or procedure in which they are specified ends.

The default scope for overrides is the activation group. For job-level scope, specify OVRSCOPE(*JOB) on the override command. For call-level scope, specify OVRSCOPE(*CALLLVL) on the override command.

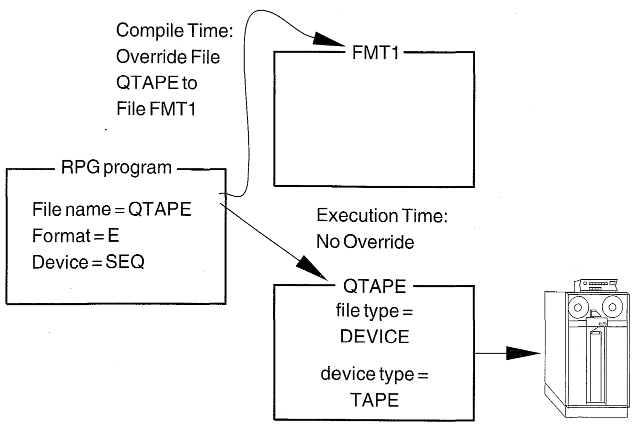
See *Data Management* for more detailed information on valid file redirections and file overrides. *ILE Concepts* also contains information about overrides and activation group vs. job level scope.

Example of Redirecting File Input and Output

The following example shows the use of a file override at compilation time. Assume that you want to use an externally described file for a TAPE device which does not have field-level description. You must:

1. Define a physical file named FMT1 with one record format that contains the description of each field in the record format. The record format is defined on the data description specifications (DDS). For a tape device, the externally described file should contain only one record format.
2. Create the file named FMT1 with a Create Physical File CL command.
3. Specify the file name of QTAPE (which is the IBM-supplied device file name for magnetic tape devices) in the RPG program. This identifies the file as externally described (indicated by an E in position 22 of the file description specifications), and specifies the device name SEQ in positions 36 through 42.
4. Use an override command–OVRDBF FILE(QTAPE) TOFILE(FMT1)–at compilation time to override the QTAPE file name and use the FMT1 file name. This command causes the compiler to copy in the external description of the FMT1 file, which describes the record format to the RPG compiler.
5. Create the RPG program using the CRTBNDRPG command or the CRTPGM command.
6. Call the program at run time. The override to file FMT1 should not be in effect while the program is running. If the override is in effect, use the CL command DLTOVR (Delete Override) before calling the program.

   Note: You may need to use the CL command OVRTAPF before you call the program to provide information necessary for opening the tape file.



File Locking

The OS/400 system allows a lock state (exclusive, exclusive allow read, shared for update, shared no update, or shared for read) to be placed on a file used during the execution of a job. Programs within a job are not affected by file lock states. A file lock state applies only when a program in another job tries to use the file concurrently. The file lock state can be allocated with the CL command ALCOBJ (Allocate Object). For more information on allocating resources and lock states, see *Data Management*.

The OS/400 system places the following lock states on database files when it opens the files:

File Type	Lock State
Input	Shared for read
Update	Shared for update
Add	Shared for update
Output	Shared for update

The shared-for-read lock state allows another user to open the file with a lock state of shared for read, shared for update, shared no update, or exclusive allow read, but the user cannot specify the exclusive use of the file. The shared-for-update lock state allows another user to open the file with shared-for-read or shared-for-update lock state.

The RPG program places an exclusive-allow-read lock state on device files. Another user can open the file with a shared-for-read lock state.

The lock state placed on the file by the RPG program can be changed if you use the Allocate Object command.

Record Locking

When a record is read by a program, it is read in one of two modes: input or update. If a program reads a record for update, a lock is placed on that record. Another program cannot read the same record for update until the first program releases that lock. If a program reads a record for input, no lock is placed on the record. A record that is locked by one program can be read for input by another program.

In RPG IV programs, you use an update file to read records for update. A record read from a file with a type other than update can be read for inquiry only. By default, any record that is read from an update file will be read for update. For update files, you can specify that a record be read for input by using one of the input operations CHAIN, READ, READE, READP, or READPE and specifying an operation code extender (N) in the operation code field following the operation code name.

When a record is locked by an RPG IV program, that lock remains until one of the following occurs:

- the record is updated.
- the record is deleted.
- another record is read from the file (either for inquiry or update).
- a SETLL or SETGT operation is performed against the file
- an UNLOCK operation is performed against the file.
- an output operation defined by an output specification with no field names included is performed against the file.

  Note: An output operation that adds a record to a file does not result in a record lock being released.

If your program reads a record for update and that record is locked through another program in your job or through another job, your read operation will wait until the record is unlocked (except in the case of shared files, see "Sharing an Open Data Path" on page 201). If the wait time exceeds that specified on the WAITRCD parameter of the file, an exception occurs. If your program does not handle this exception (RNX1218) then the default error handler is given control when a record lock timeout occurs, an RNQ1218 inquiry message will be issued. One of the options listed for this message is to retry the operation on which the timeout occurred. This will cause the operation on which the timeout occurred to be re-issued, allowing the program to continue as if the record lock timeout had not occurred. Note that if the file has an INFSR specified in which an I/O operation is performed on the file before the default error handler is given control, unexpected results can occur if the input operation that is retried is a sequential operation, since the file cursor may have been modified.

If no changes are required to a locked record, you can release it from its locked state, without modifying the file cursor, by using the UNLOCK operation or by processing output operations defined by output specifications with no field names included. These output operations can be processed by EXCEPT output, detail output, or total output.

(There are exceptions to these rules when operating under commitment control. See "Using Commitment Control" on page 231 for more information.)

Sharing an Open Data Path

An open data path is the path through which all input and output operations for a file are performed. Usually a separate open data path is defined each time a file is opened. If you specify SHARE(*YES) for the file creation or on an override, the first program's open data path for the file is shared by subsequent programs that open the file concurrently.

The position of the current record is kept in the open data path for all programs using the file. If you read a record in one program and then read a record in a called program, the record retrieved by the second read depends on whether the open data path is shared. If the open data path is shared, the position of the current record in the called program is determined by the current position in the calling program. If the open data path is not shared, each program has an independent position for the current record.

If your program holds a record lock in a shared file and then calls a second program that reads the shared file for update, you can release the first program's lock by :

- performing a READ operation on the update file by the second program, or
- using the UNLOCK or the read-no-lock operations.

In ILE, shared files are scoped to either the job level or the activation group level. Shared files that are scoped to the job level can be shared by any programs running in *any* activation group within the job. Shared files that are scoped to the activation group level can be shared *only* by the programs running in the same activation group.

The default scope for shared files is the activation group. For job-level scope, specify OVRSCOPE(*JOB) on the override command.

ILE RPG/400 offers several enhancements in the area of shared ODPs. If a program or procedure performs a read operation, another program or procedure can update the record as long as SHARE(*YES) is specified for the file in question. In addition, when using multiple-device files, if one program acquires a device, any other program sharing the ODP can also use the acquired device. It is up to the programmer to ensure that all data required to perform the update is available to the called program.

Sharing an open data path improves performance because the OS/400 system does not have to create a new open data path. However, sharing an open data path can cause problems. For example, an error is signaled in the following cases:

- If a program sharing an open data path attempts file operations other than those specified by the first open (for example, attempting input operations although the first open specified only output operations)
- If a program sharing an open data path for an externally described file tries to use a record format that the first program ignored
- If a program sharing an open data path for a program described file specifies a record length that exceeds the length established by the first open.

When several files in one program are overridden to one shared file at run time, the file opening order is important. In order to control the file opening order, you should use a programmer-controlled open or use a CL program to open the files before calling the program.

If a program shares the open data path for a primary or secondary file, the program must process the detail calculations for the record being processed before calling another program that shares that open data path. Otherwise, if lookahead is used or if the call is at total time, sharing the open data path for a primary or secondary file may cause the called program to read data from the wrong record in the file.

You must make sure that when the shared file is opened for the first time, all of the open options that are required for subsequent opens of the file are specified. If the open options specified for subsequent opens of a shared file are not included in those specified for the first open of a shared file, an error message is sent to the program.

Table 13 details the system open options allowed for each of the open options you can specify.

Table 13. System Open Options Allowed with User Open Options

RPG User Open Options	**System Open Options**
INPUT	INPUT
OUTPUT	OUTPUT (program created file)
UPDATE	INPUT, UPDATE, DELETE
ADD	OUTPUT (existing file)

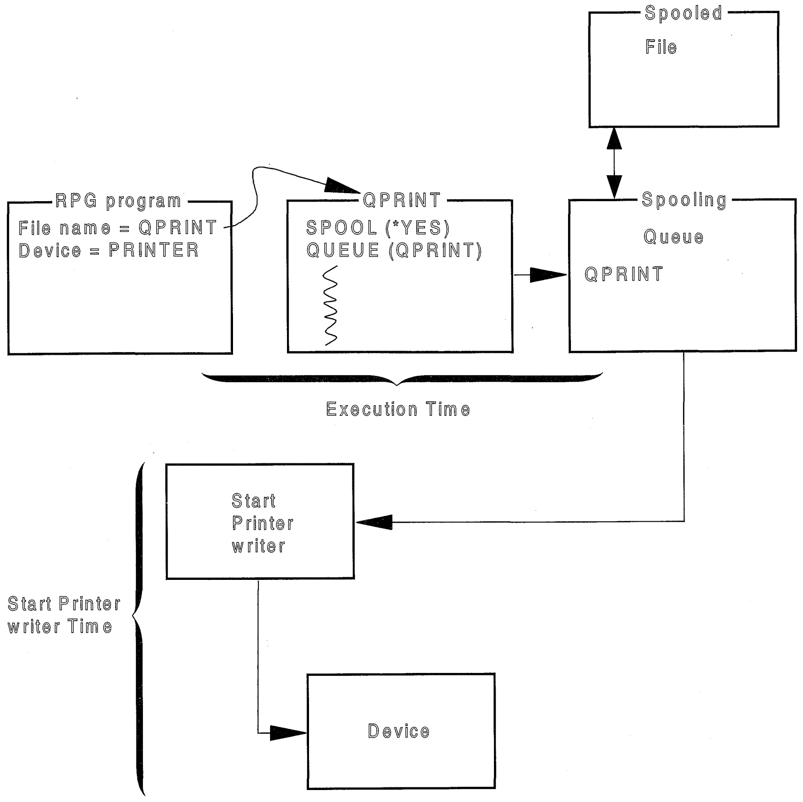
For additional information about sharing an open data path, see *DB2/400 Database Programming. ILE Concepts* also contains information about sharing open data paths and activation group vs. job level scope.

Spooling

Spooling is a system function that puts data into a storage area to wait for processing. The AS/400 system provides for the use of input and output spooling functions. Each AS/400 file description contains a spool attribute that determines whether spooling is used for the file at run time. The RPG program is not aware that spooling is being used. The actual physical device from which a file is read or to which a file is written is determined by the spool reader or the spool writer. For more detailed information on spooling, see *Data Management*.

Output Spooling

Output spooling is valid for batch or interactive jobs. The description of the file that is specified in the RPG program by the file name contains the specification for spooling as shown in the following diagram:



File override commands can be used at run time to override the spooling options specified in the file description, such as the number of copies to be printed. In addition, AS/400 spooling support allows you to redirect a file after the program has run. You can direct the same printed output to a different device such as a diskette.

Record Blocking and Unblocking

The RPG compiler unblocks input records and blocks output records to improve run-time performance in SEQ or DISK files if:

- The file is an output-only file (O is specified in position 17 of the file description specifications) and contains only one record format if the file is externally described.
- The file is a combined table file. (C is specified in position 17, and T in position 18 of the file description specifications.)
- The file is an input-only file. (I is specified in position 17 of the file description specifications.) It contains only one record format if the file is externally described, and uses only the OPEN, CLOSE, FEOD, and READ operation codes.
- The RECNO keyword is not specified on the file description specification.

The RPG compiler generates object program code to block and unblock records for all SEQ or DISK files that satisfy these conditions. Certain OS/400 system restrictions may prevent blocking and unblocking. In those cases, performance is not improved.

The input/output and device-specific feedback of the file information data structure are not updated after each read or write (except for the RRN and Key information on block reads) for files in which the records are blocked and unblocked by the RPG compiler. The feedback area is updated each time a block of records is transferred. (For further details on the file information data structure see the *ILE RPG/400 Reference*.)

You can obtain valid updated feedback information by using the CL command OVRDBF (Override with Database File) with SEQONLY(*NO) specified. If you use a file override command, the RPG IV language does not block or unblock the records in the file.

SRTSEQ/ALTSEQ in an RPG Program vs a DDS File

When a keyed file is created using SRTSEQ and LANGID, the SRTSEQ specified is used when comparing character keys in the file during CHAIN, SETLL, SETGT, READE and READPE operations. You do not have to specify the same, or any, SRTSEQ value when creating the RPG program or module.

When a value for SRTSEQ is specified on CRTBNDRPG or CRTRPGMOD, then all character comparison operations in the program will use this SRTSEQ. This value affects the comparison of *all* fields, including key fields, fields from other files and fields declared in the program.

You should decide whether to use SRTSEQ for your RPG program based on how you want operations such as IFxx, COMP, and SORTA, to work on your character data, not on what was specified when creating your files.

Chapter 15. Accessing Database Files

You can access a database file from your program by associating the file name with the device DISK in the appropriate file specification.

DISK files of an ILE RPG/400 program also associate with distributed data management (DDM) files, which allow you to access files on remote systems as database files.

Database Files

Database files are objects of type *FILE on the AS/400. They can be either physical or logical files and either externally-described or program-described. You access database files by associating the file name with the device DISK in positions 36 through 42 of the file description specifications.

Database files can be created by OS/400 Create File commands. For more information on describing and creating database files, refer to *DB2/400 Database Programming* and *DDS Reference*.

Physical Files and Logical Files

Physical files contain the actual data that is stored on the system, and a description of how data is to be presented to or received from a program. They contain only one record format, and one or more members. Records in database files can be externally- or program-described.

A physical file can have a keyed sequence access path. This means that data is presented to a program in a sequence based on one or more key fields in the file.

Logical files do not contain data. They contain a description of records found in one or more physical files. A logical file is a view or representation of one or more physical files. Logical files that contain more than one format are referred to as **multi-format** logical files.

If your program processes a logical file which contains more than one record format, you can use a read by record format to set the format you wish to use.

Data Files and Source Files

A **data file** contains actual data, or a view of the data. Records in data files are grouped into members. All the records in a file can be in one member or they can be grouped into different members. Most database commands and operations by default assume that database files which contain data have *only one* member. This means that when your program accesses database files containing data, you do not need to specify the member name for the file unless your file contains more than one member. If your file contains more than one member and a particular member is not specified, the first member is used.

Usually, database files that contain source programs are made up of more than one member. Organizing source programs into members within database files allows you to better manage your programs. The **source member** contains source statements that the system uses to create program objects.

Using Externally-Described Disk Files

Externally described DISK files are identified by an E in position 22 of the file description specifications. The E indicates that the compiler is to retrieve the external description of the file from the system when the program is compiled. Therefore, you must create the file before the program is compiled.

The external description for a DISK file includes:

- The record-format specifications that contain a description of the fields in a record
- Access path specifications that describe how the records are to be retrieved.

These specifications result from the DDS for the file and the OS/400 create file command that is used for the file.

Record Format Specifications

The record-format specifications allow you to describe the fields in a record and the location of the fields in a record. The fields are located in the record in the order specified in the DDS. The field description generally includes the field name, the field type, and the field length (including the number of decimal positions in a numeric field). Instead of specifying the field attributes in the record format for a physical or logical file, you can define them in a field-reference file.

In addition, the DDS keywords can be used to:

- Specify that duplicate key values are not allowed for the file (UNIQUE)
- Specify a text description for a record format or a field (TEXT).

For a complete list of the DDS keywords that are valid for a database file, see the *DB2/400 Database Programming*.

Figure 89 on page 207 shows an example of the DDS for a database file, and Figure 90 on page 207 for a field-reference file that defines the attributes for the fields used in the database file. See the *DDS Reference* for more information on a field-reference file.

Access Path

The description of an externally described file contains the access path that describes how records are to be retrieved from the file. Records can be retrieved based on an arrival sequence (non-keyed) access path or on a keyed-sequence access path.

The arrival sequence access path is based on the order in which the records are stored in the file. Records are added to the file one after another.

For the keyed-sequence access path, the sequence of records in the file is based on the contents of the key field that is defined in the DDS for the file. For example, in the DDS shown in Figure 89 on page 207, CUST is defined as the key field. The keyed-sequence access path is updated whenever records are added, deleted, or when the contents of a key field change.

For a complete description of the access paths for an externally-described database file, see *DB2/400 Database Programming*.

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ..*
A.........T.Name++++++.Len++TDpB......Functions++++++++++++++++++++*
A** LOGICAL  CUSMSTL    CUSTOMER MASTER FILE
A                                      UNIQUE
A          R CUSREC                    PFILE(CUSMSTP)
A                                      TEXT('Customer Master Record')
A            CUST
A            NAME
A            ADDR
A            CITY
A            STATE
A            ZIP
A            SRHCOD
A            CUSTYP
A            ARBAL
A            ORDBAL
A            LSTAMT
A            LSTDAT
A            CRDLMT
A            SLSYR
A            SLSLYR
A          K CUST
```

Figure 89. Example of the Data Description Specifications for a Database File

The sample DDS are for the customer master logical file CUSMSTL. The file contains one record format CUSREC (customer master record). The data for this file is contained in the physical file CUSMSTP, which is identified by the keyword PFILE. The UNIQUE keyword is used to indicate that duplicate key values are not allowed for this file. The CUST field is identified by a K in position 17 of the last line as the key field for this record format.

The fields in this record format are listed in the order they are to appear in the record. The attributes for the fields are obtained from the physical file CUSMSTP. The physical file, in turn, refers to a field-reference file to obtain the attributes for the fields. The field-reference file is shown in Figure 90.

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ..*
A.........T.Name++++++RLen++TDpB......Functions++++++++++++++++++++*
A**FLDRED    DSTREF     DISTRIBUTION APPLICATION FIELD REFERENCE
A          R DSTREF                    TEXT('Distribution Field Ref')
A* COMMON FIELDS USED AS REFERENCE
A            BASDAT         6  0       EDTCDE(Y) 1
A                                      TEXT('Base Date Field')
A* FIELDS USED BY CUSTOMER MASTER FILE
A            CUST           5          CHECK(MF) 2
A                                      COLHDG('Customer' 'Number')
A            NAME          20          COLHDG('Customer Name')
A            ADDR      R               REFFLD(NAME) 3
A                                      COLHDG('Customer Address')
```

Figure 90 (Part 1 of 2). Example of a field Reference File

```
A          CITY      R               REFFLD(NAME) 3
A                                    COLHDG('Customer City')
A          STATE          2          CHECK(MF) 2
A                                    COLHDG('State')
A          SRHCOD         6          CHECK(MF) 2
A                                    COLHDG('Search' 'Code')
A                                    TEXT('Customer Number Search +
A                                    Code')
A          ZIP            5  0       CHECK(MF) 2
A                                    COLHDG('Zip' 'Code')
A          CUSTYP         1  0       RANGE(1 5) 4
A                                    COLHDG('Cust' 'Type')
A                                    TEXT('Customer Type 1=Gov 2=Sch+
A                                    3=Bus 4=Pvt 5=Oth')
A          ARBAL          8  2       COLHDG('Accts Rec' 'Balance') 5
A                                    EDTCDE(J) 6
A          ORDBAL    R               REFFLD(ARBAL)
A                                    COLHDG('A/R Amt in' 'Order +
A                                    File')
A          LSTAMT    R               REFFLD(ARBAL)
A                                    COLHDG('Last' 'Amount' 'Paid')
A                                    TEXT('Last Amount Paid in A/R')
A          LSTDAT    R               REFFLD(BASDAT)
A                                    COLHDG('Last' 'Date' 'Paid')
A                                    TEXT('Last Date Paid in A/R')
A          CRDLMT    R               REFFLD(ARBAL)
A                                    COLHDG('Credit' 'Limit')
A                                    TEXT('Customer Credit Limit')
A          SLSYR     R+   2          REFFLD(ARBAL)
A                                    COLHDG('Sales' 'This' 'Year')
A                                    TEXT('Customer Sales This Year')
A          SLSLYR    R+   2          REFFLD(ARBAL)
A                                    COLHDG('Sales' 'Last' 'Year')
A                                    TEXT('Customer Sales Last Year') 7
```

Figure 90 (Part 2 of 2). Example of a field Reference File

This example of a field-reference file shows the definitions of the fields that are used by the CUSMSTL (customer master logical) file as shown in Figure 89 on page 207. The field-reference file normally contains the definitions of fields that are used by other files. The following text describes some of the entries for this field-reference file.

1 The BASDAT field is edited by the Y edit code, as indicated by the keyword EDTCDE(Y). If this field is used in an externally-described output file for an ILE RPG/400 program, the edit code used is the one specified in this field-reference file; it cannot be overridden in the ILE RPG/400 program. If the field is used in a program-described output file for an ILE RPG/400 program, an edit code must be specified for the field in the output specifications.

2 The CHECK(MF) entry specifies that the field is a mandatory fill field when it is entered from a display work station. Mandatory fill means that all characters for the field must be entered from the display work station.

3 The ADDR and CITY fields share the same attributes that are specified for the NAME field, as indicated by the REFFLD keyword.

4 The RANGE keyword, which is specified for the CUSTYP field, ensures that the only valid numbers that can be entered into this field from a display work station are 1 through 5.

5 The COLHDG keyword provides a column head for the field if it is used by the Interactive Database Utilities (IDU).

6 The ARBAL field is edited by the J edit code, as indicated by the keyword EDTCDE(J).

7 A text description (TEXT keyword) is provided for some fields. The TEXT keyword is used for documentation purposes and appears in various listings.

Valid Keys for a Record or File

For a keyed-sequence access path, you can define one or more fields in the DDS to be used as the key fields for a record format. (However, floating-point fields, variable-length fields, and null capable fields cannot be used as key fields in an RPG program.) All record types in a file do not have to have the same key fields. For example, an order header record can have the ORDER field defined as the key field, and the order detail records can have the ORDER and LINE fields defined as the key fields.

The key for a file is determined by the valid keys for the record types in that file. The file's key is determined in the following manner:

- If all record types in a file have the same number of key fields defined in the DDS that are identical in attributes, the *key for the file* consists of all fields in the key for the record types. (The corresponding fields do not have to have the same name.) For example, if the file has three record types and the key for each record type consists of fields A, B, and C, the file's key consists of fields A, B, and C. That is, the file's key is the same as the records' key.
- If all record types in the file do not have the same key fields, the key for the file consists of the key fields *common* to all record types. For example, a file has three record types and the key fields are defined as follows:
  - REC1 contains key field A.
  - REC2 contains key fields A and B.
  - REC3 contains key fields A, B, and C.

  The file's key is field A–the key field common to all record types.
- If no key field is common to all record types, there is no key for the file.

In an ILE RPG/400 program, you can specify a search argument on certain file operation codes to identify the record you want to process. The ILE RPG/400 program compares the search argument with the key of the file or record, and processes the specified operation on the record whose key matches the search argument.

Valid Search Arguments

You can specify a search argument in the ILE RPG/400 operations CHAIN, DELETE, READE, READPE, SETGT, and SETLL that specify a file name or a record name.

For an operation to a file name, the maximum number of fields that you can specify in a search argument is equal to the total number of key fields valid for the file's key. For example, if all record types in a file do not contain all of the same key fields, you can use a key list (KLIST) to specify a search argument that is composed only of the number of fields common to all record types in the file. If a file contains three record types, the key fields are defined as follows:

– REC1 contains key field A.
– REC2 contains key fields A and B.
– REC3 contains key fields A, B, and C.

The search argument can only be a single field with attributes identical to field A because field A is the only key field common to all record types. The search argument cannot contain a floating point, variable length, or null capable field.

For an operation to a record name, the maximum number of fields that you can specify in a search argument is equal to the total number of key fields valid for that record type.

If the search argument consists of one field, you can specify a literal, a field name, or a KLIST name with one KFLD. If the search argument is composed of more than one field (a composite key), you must specify a KLIST with multiple KFLDs.

The attributes of each field in the search argument must be identical to the attributes of the corresponding field in the file or record key. The attributes include the length, the data type and the number of decimal positions. The attributes are listed in the key-field-information data table of the compiler listing. See the example in "Key Field Information."

In all these file operations (CHAIN, DELETE, READE, READPE, SETGT, and SETLL), you can also specify a search argument that contains fewer than the total number of fields valid for the file or record. Such a search argument refers to a partial key.

Referring to a Partial Key

The rules for the specification of a search argument that refers to a partial key are as follows:

- The search argument is composed of fields that correspond to the leftmost (high-order) fields of the key for the file or record.
- Only the rightmost fields can be omitted from the key list (KLIST) for a search argument that refers to a partial key. For example, if the total key for a file or record is composed of key fields A, B, and C, the valid search arguments that refer to a partial key are field A, and fields A and B.
- Each field in the search argument must be identical in attributes to the corresponding key field in the file or record. The attributes include the length, data type, the number of decimal positions, and format (for example, packed or zoned).
- A search argument cannot refer to a portion of a key field.

If a search argument refers to a partial key, the file is positioned at the first record that satisfies the search argument or the record retrieved is the first record that satisfies the search argument. For example, the SETGT and SETLL operations position the file at the first record on the access path that satisfies the operation and the search argument. The CHAIN operation retrieves the first record on the access path that satisfies the search argument. The DELETE operation deletes the first record on the access path that satisfies the search argument. The READE operation retrieves the next record if the portion of the key of that record (or the record of the specified type) on the access path matches the search argument. The READPE operation retrieves the prior record if the portion of the key of that record (or the record of the specified type) on the access path matches the search argument. For more information on the above operation codes, see the *ILE RPG/400 Reference*.

Using Program-Described Disk Files

Program-described files, which are identified by an F in position 22 of the file description specifications, can be described as indexed files, as sequential files, or as record-address files.

Indexed File

An indexed file is a program-described DISK file whose access path is built on key values. You must create the access path for an indexed file by using data description specifications.

An indexed file is identified by an I in position 35 of the file description specifications.

The key fields identify the records in an indexed file. You specify the length of the key field in positions 29 through 33, the format of the key field in position 34, and the starting location of the key field in the KEYLOC keyword of the file description specifications.

An indexed file can be processed sequentially by key, sequentially within limits, or randomly by key.

Valid Search Arguments

For a program-described file, a search argument must be a single field. For the CHAIN and DELETE operations, the search argument must be the same length as the key field that is defined on the file description specifications for the indexed file. For the other file operations, the search argument may be a partial field.

The DDS specifies the fields to be used as a key field. The KEYLOC keyword of the file description specifications specify the starting position of the first key field. The entry in positions 29 through 33 of the file description specifications must specify the length of the key as defined in the DDS.

Figure 91 and Figure 92 show examples of how to use the DDS to describe the access path for indexed files.

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ..*
A..........T.Name++++++.Len++TDpB......Functions++++++++++++++++++++*
A          R FORMATA                   PFILE(ORDDTLP)
A                                      TEXT('Access Path for Indexed +
A                                      File')
A            FLDA         14
A            ORDER         5  0
A            FLDB        101
A          K ORDER
A*
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... *
FFilename++IPEASFRlen+LKlen+AIDevice+.Keywords+++++++++++++++++++++++++++++
FORDDTLL   IP   F  118     3PIDISK    KEYLOC(15)
F*
```

Figure 91. DDS and corresponding File-Description Specification to Define the Access Path for an Indexed File

You must use data description specifications to create the access path for a program-described indexed file.

In the DDS for the record format FORMATA for the logical file ORDDTLL, the field ORDER, which is five digits long, is defined as the key field, and is in packed format. The definition of ORDER as the key field establishes the keyed access for this file. Two other fields, FLDA and FLDB, describe the remaining positions in this record as character fields.

The program-described input file ORDDTLL is described on the file description specifications as an indexed file. Positions 29 through 33 must specify the number of positions in the record required for the key field as defined in the DDS: three positions. The KEYLOC keyword specifies position 15 as the starting position of the key field in the record. Because the file is defined as program-described by the F in position 22, the ILE RPG/400 compiler does not retrieve the external field-level description of the file at compilation time. Therefore, you must describe the fields in the record on the input specifications.

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ..*
A..........T.Name++++++.Len++TDpB......Functions++++++++++++++++++++*
A          R FORMAT                    PFILE(ORDDTLP)
A                                      TEXT('Access Path for Indexed +
A                                      File')
A            FLDA         14
A            ORDER         5
A            ITEM          5
A            FLDB         96
A          K ORDER
A          K ITEM
```

Figure 92. (Part 1 of 2). Using Data Description Specifications to Define the Access Path (Composite Key) for an Indexed File

In this example, the data description specifications define two key fields for the record format FORMAT in the logical file ORDDTLL. For the two fields to be used as a composite key for a program described indexed file, the key fields must be contiguous in the record.

On the file description specifications, the length of the key field is defined as 10 in positions 29 through 33 (the combined number of positions required for the ORDER and ITEM fields). The starting position of the key field is described as 15 using the keyword KEYLOC (starting in position 44). The starting position must specify the first position of the first key field.

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... *
FFilename++IPEASFRlen+LKlen+AIDevice+.Keywords+++++++++++++++++++++++++++++
FORDDTLL   IP   F  120    10AIDISK    KEYLOC(15)

*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... *
DName+++++++++++ETDsFrom+++To/L+++IDc.Keywords+++++++++++++++++++++++++++++
DKEY         DS
D K1                      1      5
D K2                      6     10

*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... *
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C                   MOVE      ORDER         K1
C                   MOVE      ITEM          K2
C     KEY           CHAIN     ORDDTLL                            99
```

Figure 93. (Part 2 of 2). Using Data Description Specifications to Define the Access Path (Composite Key) for an Indexed File

When the DDS specifies a composite key, you must build a search argument in the program to CHAIN to the file. (A KLIST cannot be used for a program-described file.) One way is to create a data structure (using definition specifications) with subfields equal to the key fields defined in the DDS. Then, in the calculations, set the subfields equal to the value of the key fields, and use the data-structure name as the search argument in the CHAIN operation.

In this example, the MOVE operations set the subfields K1 and K2 equal to the value of ORDER and ITEM, respectively. The data-structure name (KEY) is then used as the search argument in the CHAIN operation.

Sequential File

Sequential files are files where the order of the records in the file is based on the order the records are placed in the file (that is, in arrival sequence). For example, the tenth record placed in the file occupies the tenth record position.

Sequential files can be processed randomly by relative record number, consecutively, or by a record-address file. You can use either the SETLL or SETGT operation code to set limits on the file.

Record Address File

You can use a record-address file to process another file. A record-address file can contain (1) limits records that are used to process a file sequentially within limits, or (2) relative record numbers that are used to process a file by relative record numbers. The record-address file itself must be processed sequentially.

A record-address file is identified by an R in position 18 of the file description specifications. If the record-address file contains relative record numbers, position 35 must contain a T. The name of the file to be processed by the record-address file must be specified on the file description specification. You identify the file using the keyword RAFDATA(*file-name*).

Limits Records

For sequential-within-limits processing, the record-address file contains limits records. A limits record contains the lowest record key and the highest record key of the records in the file to be read.

The format of the limits records in the record-address file is as follows:

- The low key begins in position 1 of the record; the high key immediately follows the low key. No blanks can appear between the keys.
- Each record in the record-address file can contain only one set of limits. The record length must be greater than or equal to twice the length of the record key.
- The low key and the high key in the limits record must be the same length. The length of the keys must be equal to the length of the key field of the file to be processed.
- A blank entry equal in length to the record key field causes the ILE RPG/400 compiler to read the next record in the record-address file.

Relative Record Numbers

For relative-record-number processing, the record-address file contains relative record numbers. Each record retrieved from the file being processed is based on a relative record number in the record-address file. A record-address file containing relative record numbers cannot be used for limits processing. Each relative record number in the record-address file is a multi-byte binary field where each field contains a relative record number. You can specify the record-address file length as 4, 3, or blank, depending on the source of the file. When using a record-address file from the AS/400 environment, specify the record-address file length as 4, since each field is 4 bytes in length. When using a record-address file created in the System/36 environment, specify the record-address file length as 3, since each field is 3 bytes in length. If you specify the record-address file length as blank, the compiler will check the primary record length at run time and determine whether to treat the record-address file as 3 byte or as 4 byte. A minus 1 (-1 or hexadecimal FFFFFFFF) relative-record-number value stops the use of a relative-record-address file record. End of file occurs when all records from the record-address file have been processed.

Methods for Processing Disk Files

The methods of disk file processing include:

- Consecutive processing
- Sequential-by-key processing
- Random-by-key processing
- Sequential-within-limits processing.
- Relative-record-number processing

Figure 94 shows the valid entries for positions 28, 34, and 35 of the file description specification for the various file types and processing methods. The subsequent text describes each method of processing.

Processing Method	Limits Processing (Pos. 28)	Record Address Type (Pos. 34)	File Organization (Pos. 35)
Externally Described Files			
With Keys			
Sequentially	Blank	K	Blank
Randomly	Blank	K	Blank
Sequential within limits (by record-address file)	L	K	Blank
Without Keys			
Randomly/consecutively	Blank	Blank	Blank
Program Described Files			
With Keys (indexed file)			
Sequentially	Blank	A, D, G, P, T, or Z	I
Randomly	Blank	A, D, G, P, T, or Z	I
Sequential within limits (by record-address file)	L	A, D, G, P, T, or Z	I
Without Keys			
Randomly/consecutively	Blank	Blank	Blank
By record-address file	Blank	Blank	Blank
As record-address file (relative record numbers)	Blank	Blank	T
As record-address limits file	Blank	A, D, G, P, T, Z, or Blank	Blank

Figure 94. Processing Methods for DISK Files

Consecutive Processing

During consecutive processing, records are read in the order they appear in the file.

For output and input files that do not use random functions (such as SETLL, SETGT, CHAIN, or ADD), the ILE RPG/400 compiler defaults to or operates as though SEQONLY(*YES) had been specified on the CL command OVRDBF (Override with Database File). (The ILE RPG/400 compiler does not operate as though SEQONLY(*YES) had been specified for update files.) SEQONLY(*YES) allows multiple records to be placed in internal data management buffers; the records are then passed to the ILE RPG/400 compiler one at a time on input.

If, in the same job or activation group, two logical files use the same physical file, and one file is processed consecutively and one is processed for random update, a record could be updated that has already been placed in the buffer that is presented to the program. In this case, when the record is processed from the consecutive file, the record does not reflect the updated data. To prevent this problem, use the CL command OVRDBF and specify the option SEQONLY(*NO), which indicates that you do not want multiple records transferred for a consecutively processed file.

For more information on sequential only processing, see the *DB2/400 Database Programming*.

Sequential-by-Key Processing

For the sequential-by-key method of processing, records are read from the file in key sequence.

The sequential-by-key method of processing is valid for keyed files used as primary, secondary, or full procedural files.

For output files and for input files that do not use random functions (such as SETLL, SETGT, CHAIN, or ADD) and that have only one record format, the ILE RPG/400 compiler defaults to or operates as though SEQONLY(*YES) had been specified on the CL command OVRDBF. (The ILE RPG/400 compiler does not operate as though SEQONLY(*YES) had been specified for update files.) SEQONLY(*YES) allows multiple records to be placed in internal data management buffers; the records are then passed to the ILE RPG/400 compiler one at a time on input.

If, in the same job, two files use the same physical file, and one file is processed sequentially and one is processed for random update, a record could be updated that has already been placed in the buffer that is presented to the program. In this case, when the record is processed from the sequential file, the record does not reflect the updated data. To prevent this problem, use the CL command OVRDBF and specify the option SEQONLY(*NO), which indicates that you do not want multiple records transferred for a sequentially processed file.

For more information on sequential only processing, see *DB2/400 Database Programming*.

Examples of Sequential-by-Key Processing

The following three examples show you different ways of using the sequential-by-key method of processing data.

DATA DESCRIPTION SPECIFICATIONS (DDS)

Figure 95 and Figure 96 shows the data description specifications (DDS) for the physical files used by the examples. Figure 97 on page 218 shows the DDS for the logical file used by the first three examples.

```
A*****************************************************************
A* DESCRIPTION:  This is the DDS for the physical file EMPMST.   *
A*               It contains one record format called EMPREC.    *
A*               This file contains one record for each employee *
A*               of the company.                                 *
A*****************************************************************
A*
A          R EMPREC
A            ENUM           5  0       TEXT('EMPLOYEE NUMBER')
A            ENAME         20          TEXT('EMPLOYEE NAME')
A            ETYPE          1          TEXT('EMPLOYEE TYPE')
A            EDEPT          3  0       TEXT('EMPLOYEE DEPARTMENT')
A            ENHRS          3  1       TEXT('EMPLOYEE NORMAL WEEK HOURS')
A          K ENUM
```

Figure 95. DDS for database file EMPMST (physical file)

```
A*****************************************************************
A* DESCRIPTION:  This is the DDS for the physical file TRWEEK.   *
A*               It contains one record format called RCWEEK.    *
A*               This file contains all weekly entries made to   *
A*               the time reporting system.                      *
A*****************************************************************
A*
A          R RCWEEK
A            ENUM           5  0       TEXT('EMPLOYEE NUMBER')
A            WEEKNO         2  0       TEXT('WEEK NUMBER OF CURRENT YEAR')
A            EHWRK          4  1       TEXT('EMPLOYEE HOURS WORKED')
A          K ENUM
A          K WEEKNO
```

Figure 96. DDS for database file TRWEEK (physical file)

```
A*****************************************************************
A* RELATED FILES:  EMPMST    (Physical File)                     *
A*                 TRWEEK    (Physical File)                     *
A*   DESCRIPTION:  This is the DDS for the logical file EMPL1.   *
A*                 It contains two record formats called         *
A*                 EMPREC and RCWEEK.                            *
A*****************************************************************
A          R EMPREC                    PFILE(EMPMST)
A          K ENUM
A*
A          R RCWEEK                    PFILE(TRWEEK)
A          K ENUM
A          K WEEKNO
```

Figure 97. DDS for database file EMPL1 (logical file)

EXAMPLE PROGRAM 1 (Sequential-by-Key Using Primary File)

In this example, the employee master record (EMPREC) and the weekly hours worked record (RCWEEK) are contained in the same logical file EMPL1. The EMPL1 file is defined as a primary input file and is read sequentially by key. In the data description specifications for the file, the key for the EMPREC record is defined as the ENUM (employee number) field, and the key for the RCWEEK record is defined as the ENUM field plus the WEEKNO (week number) field, which is a composite key.

```
 *****************************************************************
 *  PROGRAM NAME:  YTDRPT1                                       *
 * RELATED FILES:  EMPL1     (Logical File)                      *
 *                 PRINT     (Printer File)                      *
 *   DESCRIPTION:  This program shows an example of processing   *
 *                 records using the sequential-by-key method.   *
 *                 This program prints out each employee's       *
 *                 information and weekly hours worked.          *
 *****************************************************************

FPRINT     O    F   80        PRINTER
FEMPL1     IP   E           K DISK

 *  A record-identifying indicator is assigned to each record; these
 *  record-identifying indicators are used to control processing for
 *  the different record types.

IEMPREC        01
I*
IRCWEEK        02
I*
```

Figure 98 (Part 1 of 2). Sequential-by-Key Processing, Example 1

```
      *  Since the EMPL1 file is read sequentially by key, for
      *  a valid employee number, the ENUM in a RCWEEK record
      *  must be the same as the ENUM in the last retrieved EMPREC
      *  record.  This must be checked for and is done here by saving
      *  ENUMs of the EMPREC record into the field EMPNO and comparing
      *  it with the ENUMs read from RCWEEK records.
      *  If the ENUM is a valid one, *IN12 will be seton. *IN12 is
      *  used to control the printing of the RCWEEK record.

     C                   SETOFF                                       12
     C   01              MOVE      ENUM          EMPNO              5 0
     C*
     C                   IF        (*IN02='1') AND (ENUM=EMPNO)
     C                   SETON                                        12
     C                   ENDIF

     OPRINT     H    1P                      2  6
     O                                          40 'EMPLOYEE WEEKLY WORKING '
     O                                          52 'HOURS REPORT'
     O          H    01                      1
     O                                          12 'EMPLOYEE: '
     O                       ENAME              32
     O          H    01                      1
     O                                          12 'SERIAL #: '
     O                       ENUM               17
     O                                          27 'DEPT: '
     O                       EDEPT              30
     O                                          40 'TYPE: '
     O                       ETYPE              41
     O          H    01                      1
     O                                          20 'WEEK #'
     O                                          50 'HOURS WORKED'
     O          D    12                      1
     O                       WEEKNO             18
     O                       EHWRK         3    45
```

Figure 98 (Part 2 of 2). Sequential-by-Key Processing, Example 1

EXAMPLE PROGRAM 2 (Sequential-by-Key Using READ)

This example is the same as the previous example except that the EMPL1 file is defined as a full-procedural file, and the reading of the file is done by the READ operation code.

```
*****************************************************************
*  PROGRAM NAME:  YTDRPT2                                        *
* RELATED FILES:  EMPL1    (Logical File)                        *
*                 PRINT    (Printer File)                        *
*   DESCRIPTION:  This program shows an example of processing   *
*                 records using the read operation code.         *
*                 This program prints out each employee's        *
*                 information and weekly hours worked.           *
*****************************************************************

FPRINT     O    F   80        PRINTER
FEMPL1     IF   E           K DISK

 *  The two records (EMPREC and RCWEEK) are contained in the same
 *  file, and a record-identifying indicator is assigned to each
 *  record.  The record-identifying indicators are used to control
 *  processing for the different record types.  No control levels
 *  or match fields can be specified for a full-procedural file.

IEMPREC        01
I*
IRCWEEK        02
I*

 *  The READ operation code reads a record from the EMPL1 file. An
 *  end-of-file indicator is specified in positions 58 and 59.  If
 *  the end-of-file indicator 99 is set on by the READ operation,
 *  the program branches to the EOFEND tag and processes the end-of-
 *  file routine.

C                  SETOFF                                     12
C                  READ      EMPL1                                 99
C   99             GOTO      EOFEND
C*
C   01             MOVE      ENUM          EMPNO            5 0
C*
C                  IF        (*IN02='1') AND (ENUM=EMPNO)
C                  SETON                                      12
C                  ENDIF

 *  Since EMPL1 is defined as a full-procedural file, indicator
 *  *INLR has to be seton to terminate the program after processing
 *  the last record.

C     EOFEND       TAG
C   99             SETON                                      LR
```

Figure 99 (Part 1 of 2). Sequential-by-Key Processing, Example 2

```
OPRINT    H   1P                2  6
O                                   40 'EMPLOYEE WEEKLY WORKING '
O                                   52 'HOURS REPORT'
O         H   01                1
O                                   12 'EMPLOYEE: '
O                       ENAME       32
O         H   01                1
O                                   12 'SERIAL #: '
O                       ENUM        17
O                                   27 'DEPT: '
O                       EDEPT       30
O                                   40 'TYPE: '
O                       ETYPE       41
O         H   01                1
O                                   20 'WEEK #'
O                                   50 'HOURS WORKED'
O         D   12                1
O                       WEEKNO      18
O                       EHWRK     3 45
```

Figure 99 (Part 2 of 2). Sequential-by-Key Processing, Example 2

EXAMPLE PROGRAM 3 (Matching-Record Technique)

In this example, the TRWEEK file is defined as a secondary input file. The EMPREC and RCWEEK records are processed as matching records, with the ENUM field in both records assigned the match level value of M1. Record-identifying indicators 01 and 02 are assigned to the records to control the processing for the different record types.

```
*****************************************************************
*  PROGRAM NAME:  YTDRPT5                                       *
* RELATED FILES:  EMPMST   (Physical File)                      *
*                 TRWEEK   (Physical File)                      *
*                 PRINT    (Printer File)                       *
*   DESCRIPTION:  This program shows an example of processing   *
*                 records using the matching record method.     *
*                 This program prints out each employee's       *
*                 information, weekly worked hours and amount   *
*                 of overtime.                                  *
*****************************************************************

FPRINT    O    F  80        PRINTER
FEMPMST   IP   E           K DISK
FTRWEEK   IS   E           K DISK

IEMPREC       01
I                                           ENUM            M1
IRCWEEK       02
I                                           ENUM            M1
```

Figure 100 (Part 1 of 2). Sequential-by-Key Processing, Example 5

```
C   01             Z-ADD    0           TOTHRS            5 1
C   01             Z-ADD    0           TOTOVT            5 1
C   01             SETOFF                                    12
C*
C   MR             IF       (*IN02='1')
C                  ADD      EHWRK       TOTHRS
C     EHWRK        SUB      ENHRS       OVTHRS            4 111
C   11             ADD      OVTHRS      TOTOVT
C                  SETON                                     12
C                  ENDIF

OPRINT     H    1P                    2  6
O                                        50 'YTD PAYROLL SUMMARY'
O          D    01                    1
O                                        12 'EMPLOYEE: '
O                      ENAME             32
O          D    01                    1
O                                        12 'SERIAL #: '
O                      ENUM              17
O                                        27 'DEPT: '
O                      EDEPT             30
O                                        40 'TYPE: '
O                      ETYPE             41
O          D    02 MR                 1
O                                         8 'WEEK #'
O                      WEEKNO            10
O                                        32 'HOURS WORKED = '
O                      EHWRK        3    38

  *  These 2 detail output lines are processed if *IN01 is on
  *  and no matching records found (that means no RCWEEK records
  *  for that employee found). Obviously, the total fields
  *  (TOTHRS and TOTOVT) are equal to zeros in this case.

O          D    01NMR                 1
O                                        70 'YTD HOURS WORKED = '
O                      TOTHRS       3    78
O          D    01NMR                 1
O                                        70 'YTD OVERTIME HOURS = '
O                      TOTHRS       3    78

  *  These 2 total output lines are processed before performing
  *  detail calcualations. Therefore, the total fields
  *  (TOTHRS and TOTOVT) for the employee in the last retrieved
  *  record will be printed out if the specified indicators are on.

O          T    01 12                 1
O         OR    LR 12
O                                        70 'YTD HOURS WORKED = '
O                      TOTHRS       3    78
O          T    01 12                 1
O         OR    LR 12
O                                        70 'YTD OVERTIME HOURS = '
O                      TOTOVT       3    78
```

Figure 100 (Part 2 of 2). Sequential-by-Key Processing, Example 5

Random-by-Key Processing

For the random-by-key method of processing, a search argument that identifies the key of the record to be read is specified in factor 1 of the calculation specifications for the CHAIN operation. Figure 102 on page 224 shows an example of an externally described DISK file being processed randomly by key. The specified record can be read from the file either during detail calculations or during total calculations.

The random-by-key method of processing is valid for a full procedural file designated as an input file or an update file.

For an externally-described file, position 34 of the file description specification must contain a K, which indicates that the file is processed according to an access path that is built on keys.

The data description specifications (DDS) for the file specifies the field that contains the key value (the key field). Position 35 of the file description specification must be blank.

A program-described file must be designated as an indexed file (I in position 35), and position 34 of the file description specification must contain an A, D, G, P, T, or Z. The length of the key field is identified in positions 29-33 of the file description specification, and the starting location of the key field is specified on the KEYLOC keyword. Data description specifications must be used to create the access path for a program described input file (see “Indexed File” on page 211).

Example of Random-by-Key Processing

The following is an example of how to use the random-by-key method of processing data. Figure 95 on page 217 and Figure 101 show the data description specifications (DDS) for the physical files used by EMSTUPD (Figure 102 on page 224).

```
A****************************************************************
A*  RELATED PGMS:  EMSTUPD                                      *
A*  DESCRIPTIONS:  This is the DDS for the physical file CHANGE. *
A*                 It contains one record format called CHGREC.  *
A*                 This file contains new data that is used to   *
A*                 update the EMPMST file.                       *
A****************************************************************
A*
A          R CHGREC
A            ENUM           5  0       TEXT('EMPLOYEE NUMBER')
A            NNAME         20          TEXT('NEW NAME')
A            NTYPE          1          TEXT('NEW TYPE')
A            NDEPT          3  0       TEXT('NEW DEPARTMENT')
A            NNHRS          3  1       TEXT('NEW NORMAL WEEK HOURS')
A          K ENUM
```

Figure 101. DDS for database file CHANGE (physical file)

EXAMPLE PROGRAM

In this example, the EMPMST file is defined as an Update Full-Procedural file. The update file CHANGE is to be processed by keys. The DDS for each of the externally-described files (EMPMST and CHANGE) identify the ENUM field as the key field. The read/update processes are all controlled by the operations specified in the Calculation Specifications.

```
 *****************************************************************
 *  PROGRAM NAME:  EMSTUPD                                       *
 * RELATED FILES:  EMPMST   (Physical File)                      *
 *                 CHANGE   (Physical File)                      *
 *   DESCRIPTION:  This program shows the processing of records  *
 *                 using the random-by-key method. The CHAIN     *
 *                 operation code is used.                       *
 *                 The physical file CHANGE contains all the     *
 *                 changes made to the EMPMST file.  Its record  *
 *                 format name is CHGREC.  There may be some     *
 *                 fields in the CHGREC that are left blank,     *
 *                 in that case, no changes are made to those    *
 *                 fields.                                       *
 *****************************************************************

FCHANGE    IP   E           K DISK
FEMPMST    UF   E           K DISK

 *  As each record is read from the primary input file, CHANGE,
 *  the employee number (ENUM) is used as the search argument
 *  to chain to the corresponding record in the EMPMST file.
 *  *IN03 will be set on if no corresponding record is found, which
 *  occurs when an invalid ENUM is entered into the CHGREC record.

C     ENUM          CHAIN     EMPREC                             03
C   03              GOTO      NEXT
C     NNAME         IFNE      *BLANK
C                   MOVE      NNAME         ENAME
C                   ENDIF
C     NTYPE         IFNE      *BLANK
C                   MOVE      NTYPE         ETYPE
C                   ENDIF
C     NDEPT         IFNE      *ZERO
C                   MOVE      NDEPT         EDEPT
C                   ENDIF
C     NNHRS         IFNE      *ZERO
C                   MOVE      NNHRS         ENHRS
C                   ENDIF
C                   UPDATE    EMPREC
C*
C     NEXT          TAG
```

Figure 102. Random-by-Key Processing of an Externally-Described File

Sequential-within-Limits Processing

Sequential-within-limits processing by a record-address file is specified by an L in position 28 of the file description specifications and is valid for a file with a keyed access.

You can specify sequential-within-limits processing for an input or an update file that is designated as a primary, secondary, or full-procedural file. The file can be externally-described or program-described (indexed). The file should have keys in ascending sequence.

To process a file sequentially within limits from a record-address file, the program reads:

- A limits record from the record-address file
- Records from the file being processed within limits with keys greater than or equal to the low-record key and less than or equal to the high-record key in the limits record. If the two limits supplied by the record-address file are equal, only the records with the specified key are retrieved.

The program repeats this procedure until the end of the record-address file is reached.

Examples of Sequential-within-Limits Processing

Figure 103 shows an example of an indexed file being processed sequentially within limits. Figure 105 on page 227 shows the same example with externally-described files instead of program-described files.

Figure 95 on page 217 shows the data description specifications (DDS) for the physical file used by the program ESWLIM1 (Figure 103) and ESWLIM2 (Figure 105 on page 227).

EXAMPLE PROGRAM 1 (Sequential-within-Limits Processing)

EMPMST is processed sequentially within limits (L in position 28) by the record address file LIMITS. Each set of limits from the record-address file consists of the low and high employee numbers of the records in the EMPMST file to be processed. Because the employee number key field (ENUM) is five digits long, each set of limits consists of two 5-digits keys. (Note that ENUM is in packed format, therefore, it requires three positions instead of five.)

```
 *****************************************************************
 *  PROGRAM NAME:  ESWLIM1                                       *
 * RELATED FILES:  EMPMST   (Physical File)                      *
 *                 LIMITS   (Physical File)                      *
 *                 PRINT    (Printer File)                       *
 *   DESCRIPTION:  This program shows the processing of an       *
 *                 indexed file sequentially within limits.      *
 *                 This program prints out information for the   *
 *                 employees whose employee numbers are within   *
 *                 the limits given in the file LIMITS.          *
 *****************************************************************

FLIMITS    IR   F    6     3  DISK     RAFDATA(EMPMST)
FEMPMST    IP   F   28L    3PIDISK    KEYLOC(1)
FPRINT     O    F   80        PRINTER
```

Figure 103 (Part 1 of 2). Sequential-within-Limits Processing of an Externally-Described File

```
 *  Input specifications must be used to describe the records in the
 *  program-described file EMPMST.

IEMPMST    NS  01
I                            P    1    3 0ENUM
I                                 4   23  ENAME
I                                24   24  ETYPE
I                            P   25   26 0EDEPT

 *  As EMPMST is processed within each set of limits, the corres-
 *  ponding records are printed.  Processing of the EMPMST file is
 *  complete when the record-address file LIMITS reaches end of file.

OPRINT     H    1P                  1
O                                        12 'SERIAL #'
O                                        22 'NAME'
O                                        45 'DEPT'
O                                        56 'TYPE'
O          D    01                  1
O                        ENUM            10
O                        ENAME           35
O                        EDEPT           45
O                        ETYPE           55
```

Figure 103 (Part 2 of 2). Sequential-within-Limits Processing of an Externally-Described File

EXAMPLE PROGRAM 2 (Sequential-within-Limits Processing)

Figure 104 shows the data description specifications (DDS) for the record-address limits file used by the program ESWLIM2 (Figure 105 on page 227).

```
A*****************************************************************
A* RELATED PROGRAMS:  ESWLIM                                     *
A*      DESCRIPTION:  This is the DDS for the physical file      *
A*                    LIMITS.                                    *
A*                    It contains a record format named LIMIT.   *
A*****************************************************************
A
A          R LIMIT
A            LOW            5  0
A            HIGH           5  0
```

Figure 104. DDS for record address file LIMITS (physical file)

This program performs the same job as the previous program. The only difference is that the physical file EMPMST is defined as an externally-described file instead of a program-described file.

```
*****************************************************************
*  PROGRAM NAME:  ESWLIM2                                       *
* RELATED FILES:  EMPMST   (Physical File)                      *
*                 LIMITS   (Physical File)                      *
*                 PRINT    (Printer File)                       *
*   DESCRIPTION:  This program shows the processing of an       *
*                 externally described file sequentially        *
*                 within limits.                                *
*                 This program prints out information for the   *
*                 employees whose employee numbers are within   *
*                 the limits given in the file LIMITS.          *
*****************************************************************

FLIMITS    IR   F    6     3  DISK    RAFDATA(EMPMST)
FEMPMST    IP   E     L     K DISK
FPRINT     O    F   80        PRINTER

 *  Input Specifications are optional for an externally described
 *  file.  Here, *IN01 is defined as the record-identifying
 *  indicator for the record-format EMPREC to control the
 *  processing of this record.

IEMPREC        01

OPRINT     H    1P                      1
O                                           12 'SERIAL #'
O                                           22 'NAME'
O                                           45 'DEPT'
O                                           56 'TYPE'
O          D    01                      1
O                        ENUM               10
O                        ENAME              35
O                        EDEPT              45
O                        ETYPE              55
O*
```

Figure 105. Sequential-within-Limits Processing of a Program-Described File

Relative-Record-Number Processing

Random input or update processing by relative record number applies to full procedural files only. The desired record is accessed by the CHAIN operation code.

Relative record numbers identify the positions of the records relative to the beginning of the file. For example, the relative record numbers of the first, fifth, and seventh records are 1, 5, and 7, respectively.

For an externally-described file, input or update processing by relative record number is determined by a blank in position 34 of the file description specifications and the use of the CHAIN operation code. Output processing by relative record number is determined by a blank in position 34 and the use of the RECNO keyword on the file description specification line for the file.

Use the RECNO keyword on a file description specifications to specify a numeric field that contains the relative record number that specifies where a new record is to be added to this file. The RECNO field must be defined as numeric with zero decimal positions. The field length must be large enough to contain the largest record number for the file. A RECNO field must be specified if new records are to be placed in the file by using output specifications or a WRITE operation.

When you update or add a record to a file by relative record number, the record must already have a place in the member. For an update, that place can be a valid existing record; for a new record, that place can be a deleted record.

You can use the CL command INZPFM to initialize records for use by relative record number. The current relative record number is placed in the RECNO field for all retrieval operations or operations that reposition the file (for example, SETLL, CHAIN, READ).

Valid File Operations

Figure 106 on page 229 shows the valid file operation codes allowed for DISK files processed by keys and Figure 107 on page 230 for DISK files processed by non-keyed methods. The operations shown in these figures are valid for externally-described DISK files and program-described DISK files.

Before running your program, you can override a file to another file. In particular, you can override a sequential file in your program to an externally-described, keyed file. (The file is processed as a sequential file.) You can also override a keyed file in your program to another keyed file, providing the key fields are compatible. For example, the overriding file must not have a shorter key field than you specified in your program.

Note: When a database record is deleted, the physical record is marked as deleted. Deleted records can occur in a file if the file has been initialized with deleted records using the Initialize Physical File Member (INZPFM) command. Once a record is deleted, it cannot be read. However, you can use the relative record-number to position to the record and then write over its contents.

File-Description Specifications Positions					Calculation Specifications Positions
17	18	20	28[1]	34[2]	26-35
I	P/S			K/A/P/G/ D/T/Z	CLOSE, FEOD, FORCE
I	P/S	A		K/A/P/G/ D/T/Z	WRITE, CLOSE, FEOD, FORCE
I	P/S		L	K/A/P/G/ D/T/Z	CLOSE, FEOD, FORCE
U	P/S			K/A/P/G/ D/T/Z	UPDATE, DELETE, CLOSE, FEOD, FORCE
U	P/S	A		K/A/P/G/ D/T/Z	UPDATE, DELETE, WRITE, CLOSE, FEOD, FORCE
U	P/S		L	K/A/P/G/ D/T/Z	UPDATE, CLOSE, FEOD, FORCE
I	F			K/A/P/G/ D/T/Z	READ, READE, READPE, READP, SETLL, SETGT, CHAIN, OPEN, CLOSE, FEOD
I	F	A		K/A/P/G/ D/T/Z	WRITE, READ, READPE, READE, READP, SETLL, SETGT, CHAIN, OPEN, CLOSE, FEOD
I	F		L	K/A/P/G/ D/T/Z	READ, OPEN, CLOSE, FEOD
U	F			K/A/P/G/ D/T/Z	READ, READE, READPE, READP, SETLL, SETGT, CHAIN, UPDATE, DELETE, OPEN, CLOSE, FEOD
U	F	A		K/A/P/G/ D/T/Z	WRITE, UPDATE, DELETE, READ, READE, READPE, READP, SETLL, SETGT, CHAIN, OPEN, CLOSE, FEOD
U	F		L	K/A/P/G/ D/T/Z	READ, UPDATE, OPEN, CLOSE, FEOD
O	Blank	A		K/A/P/G/ D/T/Z	WRITE (add new records to a file), OPEN, CLOSE, FEOD
O	Blank			K/A/P/G/ D/T/Z	WRITE (initial load of a new file)[3], OPEN, CLOSE, FEOD

Note: [1]An L must be specified in position 28 to specify sequential-within-limits processing by a record-address file for an input or an update file.

Note: [2]Externally described files require a K in position 34; program-described files require an A,P,G,D,T, or Z in position 34 and an I in position 35.

Note: [3]An A in position 20 is not required for the initial loading of records into a new file. If A is specified in position 20, ADD must be specified on the output specifications. The file must have been created with the OS/400 CREATE FILE command.

Figure 106. Valid File Operations for Keyed Processing Methods (Random by Key, Sequential by Key, Sequential within Limits)

File-Description Specifications Positions					**Calculation Specifications Positions**
17	18	20	34	44-80	26-35
I	P/S		Blank		CLOSE, FEOD, FORCE
I	P/S		Blank	RECNO	CLOSE, FEOD, FORCE
U	P/S		Blank		UPDATE, DELETE, CLOSE, FEOD, FORCE
U	P/S		Blank	RECNO	UPDATE, DELETE, CLOSE, FEOD, FORCE
I	F		Blank		READ, READP, SETLL, SETGT, CHAIN, OPEN, CLOSE, FEOD
I	F		Blank	RECNO	READ, READP, SETLL, SETGT,
U	F		Blank		READ, READP, SETLL, SETGT, CHAIN, UPDATE, DELETE, OPEN, CLOSE, FEOD
U	F		Blank	RECNO	READ, READP, SETLL, SETGT, CHAIN, UPDATE, DELETE, OPEN, CLOSE, FEOD
U	F	A	Blank	RECNO	WRITE (overwrite a deleted record), READ, READP, SETLL, SETGT, CHAIN, UPDATE, DELETE, OPEN, CLOSE, FEOD
I	R		A/P/G/ D/T/Z/ Blank[1]		OPEN, CLOSE, FEOD
I	R		Blank[2]		OPEN, CLOSE, FEOD
O	Blank	A	Blank	RECNO	WRITE[3] (add records to a file), OPEN, CLOSE, FEOD
O	Blank		Blank	RECNO	WRITE[4] (initial load of a new file), OPEN, CLOSE, FEOD
O	Blank		Blank	Blank	WRITE (sequentially load or extend a file), OPEN, CLOSE, FEOD
Note: [1]If position 34 is blank for a record-address-limits file, the format of the keys in the record-address file is the same as the format of the keys in the file being processed.					
Note: [2]A record-address file containing relative record numbers requires a T in position 35.					
Note: [3]The RECNO field that contains the relative record number must be set prior to the WRITE operation or if ADD is specified on the output specifications.					
Note: [4]An A in position 20 is not required for the initial loading of the records into a new file; however, if A is specified in position 20, ADD must be specified on output specifications. The file must have been created with one of the OS/400 file creation commands.					

Figure 107. Valid File Operations for Non-keyed Processing Methods (Sequential, Random by Relative Record Number, and Consecutive)

Using Commitment Control

This section describes how to use commitment control to process file operations as a group. With commitment control, you ensure one of two outcomes for the file operations:

- all of the file operations are successful (a commit operation)
- none of the file operations has any effect (a rollback operation).

In this way, you process a group of operations as a unit.

To use commitment control, you do the following:

- On the AS/400:
  1. Prepare for using commitment control:. Use the CL commands CRTJRN (Create Journal), CRTJRNRCV (Create Journal Receiver) and STRJRNPF (Journal Physical File).
  2. Notify the AS/400 when to start and end commitment control: Use the CL commands STRCMTCTL (Start Commitment Control) and ENDCMTCTL (End Commitment Control). See the *CL Reference* for information on these commands.
- In the RPG program:
  1. Specify commitment control (COMMIT) on the file-description specifications of the files you want under commitment control.
  2. Use the COMMIT (commit) operation code to apply a group of changes to files under commitment control, or use the ROLBK (Roll Back) operation code to eliminate the pending group of changes to files under commitment control.

Note: Commitment control applies only to database files.

Starting and Ending Commitment Control

The CL command STRCMTCTL notifies the system that you want to start commitment control.

The LCKLVL(Lock Level) parameter allows you to select the level at which records are locked under commitment control. See "Commitment Control Locks" on page 232 and *CL Programming* for further details on lock levels.

You can make commitment control conditional, in the sense that the decision whether to process a file under commitment control is made at run time. For further information, see "Specifying Conditional Commitment Control" on page 235.

When you complete a group of changes with a COMMIT operation, you can specify a label to identify the end of the group. In the event of an abnormal job end, this identification label is written to a file, message queue, or data area so that you know which group of changes is the last group to be completed successfully. You specify this file, message queue, or data area on the STRCMTCTL command.

Before you call any program that processes files specified for commitment control, issue the STRCMTCTL command. If you call a program that opens a file specified for commitment control before you issue the STRCMTCTL command, the opening of the file will fail.

The CL command ENDCMTCTL notifies the system that your activation group or job has finished processing files under commitment control. See the *CL Reference* for further information on the STRCMTCTL and ENDCMTCTL commands.

Commitment Control Locks

On the STRCMTCTL command, you specify a level of locking, either LCKLVL(*ALL), LCKLVL(*CHG), or LCKLVL(*CS). When your program is operating under commitment control and has processed an input or output operation on a record in a file under commitment control, the record is locked by commitment control as follows:

- Your program can access the record.
- Another program in your activation group or job, with this file under commitment control, can read the record. If the file is a shared file, the second program can also update the record.
- Another program in your activation group or job that does not have this file under commitment control cannot read or update the record.
- Another program in a separate activation group or job, with this file under commitment control, can read the record if you specified LCKLVL(*CHG), but it cannot read the record if you specified LCKLVL(*ALL). With either lock level, the next program cannot update the record.
- Another program that does not have this file under commitment control and that is not in your activation group or job can read but not update the record.
- Commitment control locks are different than normal locks, depend on the LCKLVL specified, and can only be released by the COMMIT and ROLBK operations.

The COMMIT and ROLBK operations release the locks on the records. The UNLOCK operation will not release records locked using commitment control. See the *CL Reference* for details on lock levels.

The number of entries that can be locked under commitment control before the COMMIT or ROLBK operations are required may be limited. For more information, see *Backup and Recovery – Advanced*, SC41-3305.

Note: The SETLL and SETGT operations will lock a record in the same cases where a read operation (not for update) would lock a record for commitment control.

Commitment Control Scoping

When commitment control is started by using the STRCMTCTL command, the system creates a **commitment definition**. A commitment definition contains information pertaining to the resources being changed under commitment control within that job. Each commitment definition is known only to the job that issued the STRCMTCTL command and is ended when you issue the ENDCMTCTL command.

The scope for commitment definition indicates which programs within the job use that commitment definition. A commitment definition can be scoped at the activation group level or at the job level.

The default scope for a commitment definition is to the activation group of the program issuing the STRCMTCTL command, that is, at the activation group level. Only programs that run within that activation group will use that commitment definition. OPM programs will use the *DFTACTGRP commitment definition. ILE programs will use the activation group they are associated with.

You specify the scope for a commitment definition on the commitment scope (CMTSCOPE) parameter of the STRCMTCTL command. For further information on the commitment control scope within ILE, refer to "Data Management Scoping" in *ILE Concepts*, and also *Data Management*.

Specifying Files for Commitment Control

To indicate that a DISK file is to run under commitment control, enter the keyword COMMIT in the keyword field of the file description specification.

When a program specifies commitment control for a file, the specification applies only to the input and output operations made by this program for this file. Commitment control does not apply to operations other than input and output operations. It does not apply to files that do not have commitment control specified in the program doing the input or output operation.

When more than one program accesses a file as a shared file, all or none of the programs must specify the file to be under commitment control.

Using the COMMIT Operation

The COMMIT operation tells the system that you have completed a group of changes to the files under commitment control. The ROLBK operation eliminates the current group of changes to the files under commitment control. For information on how to specify these operation codes and what each operation does, see the *ILE RPG/400 Reference*.

If the system fails, it implicitly issues a ROLBK operation. You can check the identity of the last successfully completed group of changes using the label you specify in factor 1 of the COMMIT operation code, and the notify-object you specify on the STRCMTCTL command.

At the end of an activation group or job, or when you issue the ENDCMTCTL command, the OS/400 system issues an implicit ROLBK, which eliminates any changes since the last ROLBK or COMMIT operation that you issued. To ensure that all your file operations have effect, issue a COMMIT operation before ending an activation group or job operating under commitment control.

The OPEN operation permits input and output operations to be made to a file and the CLOSE operation stops input and output operations from being made to a file. However, the OPEN and CLOSE operations do not affect the COMMIT and ROLBK operations. A COMMIT or ROLBK operation affects a file, even after the file has been closed. For example, your program may include the following steps:

1. Issue COMMIT (for files already opened under commitment control).
2. Open a file specified for commitment control.
3. Perform some input and output operations to this file.
4. Close the file.
5. Issue ROLBK.

The changes made at step 3 are rolled back by the ROLBK operation at step 5, even though the file has been closed at step 4. The ROLBK operation could be issued from another program in the same activation group or job.

A program does not have to operate all its files under commitment control, and to do so may adversely affect performance. The COMMIT and ROLBK operations have no effect on files that are not under commitment control.

Note: When multiple devices are attached to an application program, and commitment control is in effect for the files this program uses, the COMMIT or ROLBK operations continue to work on a file basis and not by device. The database may be updated with partially completed COMMIT blocks or changes that other users have completed may be eliminated. It is your responsibility to ensure this does not happen.

Example of Using Commitment Control

This example illustrates the specifications and CL commands required for a program to operate under commitment control.

To prepare for using commitment control, you issue the following CL commands:

1. `CRTJRNRCV JRNRCV (RECEIVER)`

   This command creates a journal receiver RECEIVER.

2. `CRTJRN JRN(JOURNAL) JRNRCV(RECEIVER)`

   This command creates a journal JOURNAL and attaches the journal receiver RECEIVER.

3. `STRJRNPF FILE(MASTER TRANS) JRN(JOURNAL)`

   This command directs journal entries for the file MASTER and the file TRANS to the journal JOURNAL.

In your program, you specify COMMIT for the file MASTER and the file TRANS:

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... *
FFilename++IPEASFRlen+LKlen+AIDevice+.Keywords+++++++++++++++++++++++++
FMASTER    UF   E       K      DISK    COMMIT
FTRANS     UF   E       K      DISK    COMMIT
F*

*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... *
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C                   :
C                   :
C*
C*  Use the COMMIT operation to complete a group of operations if
C*  they were successful or rollback the changes if they were not
C*  successful.
C*
C                   UPDATE    MAST_REC                             90
C                   UPDATE    TRAN_REC                             91
C                   IF        *IN90 OR *IN91
C                   ROLBK
C                   ELSE
C                   COMMIT
C                   ENDIF
```

Figure 108. Example of Using Commitment Control

To operate your program (named REVISE) under commitment control, you issue the commands:

1. `STRCMTCTL LCKLVL(*ALL)`

   This command starts commitment control with the highest level of locking.

2. `CALL REVISE`

   This command calls the program REVISE.

3. `ENDCMTCTL`

   This command ends commitment control and causes an implicit Roll Back operation.

Specifying Conditional Commitment Control

You can write a program so that the decision to open a file under commitment control is made at run time. By implementing conditional commitment control, you can avoid writing and maintaining two versions of the same program: one which operates under commitment control, and one which does not.

The COMMIT keyword has an optional parameter which allows you to specify conditional commitment control. You enter the COMMIT keyword in the keyword section of the file description specifications for the file(s) in question. The ILE RPG/400 compiler implicitly defines a one-byte character field with the same name as the one specified as the parameter. If the parameter is set to '1', the file will run under commitment control.

The COMMIT keyword parameter must be set prior to opening the file. You can set the parameter by passing in a value when you call the program or by explicitly setting it to '1' in the program.

For shared opens, if the file in question is already open, the COMMIT keyword parameter has no effect, even if it is set to '1'.

Figure 109 is an example showing conditional commitment control.

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... *
FFilename++IPEASFRlen+LKlen+AIDevice+.Keywords+++++++++++++++++++++++++
FMASTER    UF   E       K      DISK    COMMIT(COMITFLAG)
FTRANS     UF   E       K      DISK    COMMIT(COMITFLAG)
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... *
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C*  If COMITFLAG = '1' the files are opened under commitment control,
C*  otherwise they are not.
C     *ENTRY        PLIST
C                   PARM                    COMITFLAG
C                   :
C                   :
C*
C*  Use the COMMIT operation to complete a group of operations if
C*  they were successful or rollback the changes if they were not
C*  successful.  You only issue the COMIT or ROLBK if the files
c*  were opened for commitment control (ie. COMITFLAG = '1')
C*
C                   UPDATE    MAST_REC                             90
C                   UPDATE    TRAN_REC                             91

C                   IF        COMITFLAG = '1'

C                   IF        *IN90 OR *IN91
C                   ROLBK
C                   ELSE
C                   COMMIT
C                   ENDIF

C                   ENDIF
C*
```

Figure 109. Example of Using Conditional Commitment Control

Commitment Control in the Program Cycle

Commitment control is intended for full procedural files, where the input and output is under your control. Do not use commitment control with primary and secondary files, where input and output is under the control of the RPG program cycle. The following are some of the reasons for this recommendation:

- You cannot issue a COMMIT operation for the last total output in your program.
- It is difficult to program within the cycle for recovery from a locked-record condition.
- Level indicators are not reset by the ROLBK operation.
- After a ROLBK operation, processing matching records may produce a sequence error.

DDM Files

ILE RPG/400 programs access files on remote systems through **distributed data management** (DDM). DDM allows application programs on one system to use files stored on a remote system as database files. No special statements are required in ILE RPG/400 programs to support DDM files.

A **DDM file** is created by a user or program on a local (source) system. This file (with object type *FILE) identifies a file that is kept on a remote (target) system. The DDM file provides the information needed for a local system to locate a remote system and to access the data in the source file. For more information about using DDM and creating DDM files, refer to *Distributed Data Management*.

Using Pre-V3R1 DDM Files

If you are using a pre-Version 3 Release 1.0 DDM file, the key comparison is not done at the Data Management level during a READE, READPE, or SETLL operation, or during sequential-within-limits processing by a record address file. The READE, READPE, or SETLL operation or sequential-within-limits processing by a record address file, will instead compare the keys using the *HEX collating sequence.

This may give different results than expected when the content of the fields on which the access path is built differs from the actual content of the fields. Some of the DDS features that cause the key comparison to differ are:

- ALTSEQ specified for the file
- ABSVAL, ZONE, UNSIGNED. or DIGIT keywords on key fields
- Variable length, Date, Time, or Timestamp key fields
- The SRTSEQ for the file is not *HEX

In addition, if the sign of a numeric field is different from the system preferred sign, the key comparison will also differ.

The first time that the key comparison is not done at the Data Management level on a pre-V3R1 DDM file during the READE, READPE, or SETLL operation or during sequential-within-limits processing by a record address file, an informational message (RNI2002) will be issued.

Note: The performance of I/O operations that have the possibility of not finding a record (SETLL, CHAIN, SETGT, READE, READPE), will be slower than the pre-Version 3 Release 1.0 equivalent.

Chapter 16. Accessing Externally-Attached Devices

You can access externally-attached devices from RPG by using device files. **Device files** are files that provide access to externally-attached hardware such as printers, tape units, diskette units, display stations, and other systems that are attached by a communications line.

This chapter describes how to access externally-attached devices using RPG device names PRINTER, SEQ, and SPECIAL. For information on display stations and ICF devices see Chapter 17, "Using WORKSTN Files" on page 255

Types of Device Files

Before your program can read or write to the devices on the system, a device description that identifies the hardware capabilities of the device to the operating system must be created when the device is configured. A device file specifies how a device can be used. By referring to a specific device file, your RPG program uses the device in the way that it is described to the system. The device file formats output data from your RPG program for presentation to the device, and formats input data from the device for presentation to your RPG program.

You use the device files listed in Table 14 to access the associated externally attached devices:

Table 14. AS/400 Device Files, Related CL commands, and RPG Device Name

Device File	Associated Externally Attached Device	CL commands	RPG Device Name
Printer Files	Provide access to printer devices and describe the format of printed output.	CRTPRTF CHGPRTF OVRPRTF	PRINTER
Tape Files	Provide access to data files which are stored on tape devices.	CRTTAPF CHGTAPF OVRTAPF	SEQ
Diskette Files	Provide access to data files which are stored on diskette devices.	CRTDKTF CHGDKTF OVRDKTF	DISK
Display Files	Provide access to display devices.	CRTDSPF CHGDSPF OVRDSPF	WORKSTN
ICF Files	Allow a program on one system to communicate with a program on the same system or another system.	CRTICFF CHGICFF OVRICFF	WORKSTN

The device file contains the file description, which identifies the device to be used; it does not contain data.

Accessing Printer Devices

PRINTER files of ILE RPG/400 programs associate with the printer files on the AS/400 system:

Printer files allow you to print output files. This chapter provides information on how to specify and use printer files in ILE RPG/400 programs.

Specifying PRINTER Files

To indicate that you want your program to access printer files, specify PRINTER as the device name for the file in a File Description specification. Each file must have a unique file name. A maximum of eight printer files is allowed per program.

PRINTER files can be either externally-described or program-described. Overflow indicators OA-OG and OV, fetch overflow, space/skip entries, and the PRTCTL keyword are not allowed for an externally-described PRINTER file. See the *ILE RPG/400 Reference* for the valid output specification entries for an externally-described file. See the *DDS Reference* for information about the DDS for externally-described printer files.

For an externally-described PRINTER file, you can specify the DDS keyword INDARA. If you try to use this keyword for a program-described PRINTER file, you get a run-time error.

You can use the CL command CRTPRTF (Create Print File) to create a printer file (see *CL Reference* for further information on the CRTPRTF command); or you can also use the IBM-supplied file names. See *Data Management* for more information on these file names.

The file operation codes that are valid for a PRINTER file are WRITE, OPEN, CLOSE, and FEOD. For a complete description of these operation codes, see the *ILE RPG/400 Reference*.

Handling Page Overflow

An important consideration when you use a PRINTER file is page overflow. For an externally-described PRINTER file, you are responsible for handling page overflow. Do one of the following:

- Specify an indicator, *IN01 through *IN99, as the overflow indicator using the keyword OFLIND(*overflow indicator*) in the Keywords field of the file description specifications.
- Check the printer device feedback section of the INFDS for line number and page overflow. Refer to the *ILE RPG/400 Reference* for more information.
- Count the number of output lines per page.
- Check for a file exception/error by specifying an indicator in positions 73 and 74 of the calculation specifications that specify the output operation, or by specifying an INFSR that can handle the error. The INFDS has detailed information on the file exception/error. See Chapter 11, “Handling Exceptions” on page 153 for further information on exception and error handling.

For either a program-described or an externally-described file, you can specify an indicator, *IN01 through *IN99, using the keyword OFLIND(*overflow indicator*) on the File Description specification. This indicator is set on when a line is printed on the overflow line, or the overflow line is reached or passed during a space or skip operation. Use the indicator to condition your response to the overflow condition. The indicator does not condition the RPG overflow logic as an overflow indicator (*INOA through *INOG, *INOV) does. You are responsible for setting the indicator off.

For both program-described and externally-described files, the line number and page number are available in the printer feedback section of the INFDS for the file. To access this information specify the INFDS keyword on the file specification. On the specification, define the line number in positions 367-368 and define the page number in positions 369-372 of the data structure. Both the line number and the page number fields must be defined as binary with no decimal positions. Because the INFDS will be updated after every output operation to the printer file, these fields can be used to determine the current line and page number without having line-count logic in the program.

Note: If you override a printer file to a different device, such as a disk, the printer feedback section of the INFDS will not be updated, and your line count logic will not be valid.

For a program-described PRINTER file, the following sections on overflow indicators and fetch overflow logic apply.

Using Overflow Indicators in Program-Described Files

An overflow indicator (OA through OG, OV) is set on when the last line on a page has been printed or passed. An overflow indicator can be used to specify the lines to be printed on the next page. Overflow indicators can be specified only for program-described PRINTER files and are used primarily to condition the printing of heading lines. An overflow indicator is specified using the keyword OFLIND on the file description specifications and can be used to condition operations in the calculation specifications (positions 9 through 11) and output specifications (positions 21 through 29). If an overflow indicator is not specified, the compiler assigns the first unused overflow indicator to the PRINTER file. Overflow indicators can also be specified as resulting indicators on the calculation specifications (positions 71 through 76).

The compiler sets on an overflow indicator only the first time an overflow condition occurs on a page. An overflow condition exists whenever one of the following occurs:

- A line is printed past the overflow line.
- The overflow line is passed during a space operation.
- The overflow line is passed during a skip operation.

Table 15 on page 242 shows the results of the presence or absence of an overflow indicator on the file description and output specifications.

The following considerations apply to overflow indicators used on the output specifications:

- Spacing past the overflow line sets the overflow indicator on.
- Skipping past the overflow line to any line on the same page sets the overflow indicator on.

- Skipping past the overflow line to any line on the new page does not set the overflow indicator on unless a skip-to is specified past the specified overflow line.
- A skip to a new page specified on a line not conditioned by an overflow indicator sets the overflow indicator off after the forms advance to a new page.
- If you specify a skip to a new line and the printer is currently on that line, a skip does not occur. The overflow indicator is set to off, unless the line is past the overflow line.
- When an OR line is specified for an output print record, the space and skip entries of the preceding line are used. If they differ from the preceding line, enter space and skip entries on the OR line.
- Control level indicators can be used with an overflow indicator so that each page contains information from only one control group. See Figure 111 on page 243.
- For conditioning an overflow line, an overflow indicator can appear in either an AND or an OR relationship. For an AND relationship, the overflow indicator must appear on the main specification line for that line to be considered an overflow line. For an OR relationship, the overflow indicator can be specified on either the main specification line or the OR line. Only one overflow indicator can be associated with one group of output indicators. For an OR relationship, only the conditioning indicators on the specification line where an overflow indicator is specified is used for the conditioning of the overflow line.
- If an overflow indicator is used on an AND line, the line is *not* an overflow line. In this case, the overflow indicator is treated like any other output indicator.
- When the overflow indicator is used in an AND relationship with a record identifying indicator, unusual results are often obtained because the record type might not be the one read when overflow occurred. Therefore, the record identifying indicator is not on, and all lines conditioned by both overflow and record identifying indicators do not print.
- An overflow indicator conditions an exception line (E in position 17), and conditions fields within the exception record.

Table 15. Results of the Presence or Absence of an Overflow Indicator

File Description Specifications Positions 44-80	**Output Specifications Positions 21-29**	**Action**
No entry	No entry	First unused overflow indicator used to condition skip to next page at overflow.
No entry	Entry	Error at compile time; overflow indicator dropped from output specifications. First unused overflow indicator used to condition skip to next page at overflow.
OFLIND(*indicator*)	No entry	Continuous printing; no overflow recognized.
OFLIND(*indicator*)	Entry	Processes normal overflow.

Example of Printing Headings on Every Page

Figure 110 shows an example of the coding necessary for printing headings on every page: first page, every overflow page, and each new page to be started because of a change in control fields (L2 is on). The first line allows the headings to be printed at the top of a new page (skip to 06) only when an overflow occurs (OA is on and L2 is not on).

The second line allows printing of headings on the new page only at the beginning of a new control group (L2 is on). This way, duplicate headings caused by both L2 and OA being on at the same time do not occur. The second line allows headings to be printed on the first page after the first record is read because the first record always causes a control break (L2 turns on) if control fields are specified on the record.

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... *
OFilename++DF..N01N02N03Excnam++++B++A++Sb+Sa+.............................
OPRINT     H    OANL2                3  6
O..............N01N02N03Field+++++++++YB.End++PConstant/editword/DTformat++
O         OR    L2
O                                         8 'DATE'
O                                        18 'ACCOUNT'
O                                        28 'N A M E'
O                                        46 'BALANCE'
O*
```

Figure 110. Printing a Heading on Every Page

Example of Printing a Field on Every Page

Figure 111 shows the necessary coding for the printing of certain fields on every page; a skip to 06 is done either on an overflow condition or on a change in control level (L2). The NL2 indicator prevents the line from printing and skipping twice in the same cycle.

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... *
OFilename++DF..N01N02N03Excnam++++B++A++Sb+Sa+.............................
OPRINT     D    OANL2                3  6
O         OR    L2
O..............N01N02N03Field+++++++++YB.End++PConstant/editword/DTformat++
O                        ACCT             8
O*
```

Figure 111. Printing a Field on Every Page

Using the Fetch-Overflow Routine in Program-Described Files

When there is not enough space left on a page to print the remaining detail, total, exception, and heading lines conditioned by the overflow indicator, the fetch overflow routine can be called. This routine causes an overflow. To determine when to fetch the overflow routine, study all possible overflow situations. By counting lines and spaces, you can calculate what happens if overflow occurs on each detail, total, and exception line.

The fetch-overflow routine allows you to alter the basic ILE RPG/400 overflow logic to prevent printing over the perforation and to let you use as much of the page as possible. During the regular program cycle, the compiler checks only once, immediately after total output, to see if the overflow indicator is on. When the fetch overflow function is specified, the compiler checks overflow on each line for which fetch overflow is specified.

Figure 112 shows the normal processing of overflow printing when fetch overflow is set on and when it is set off.

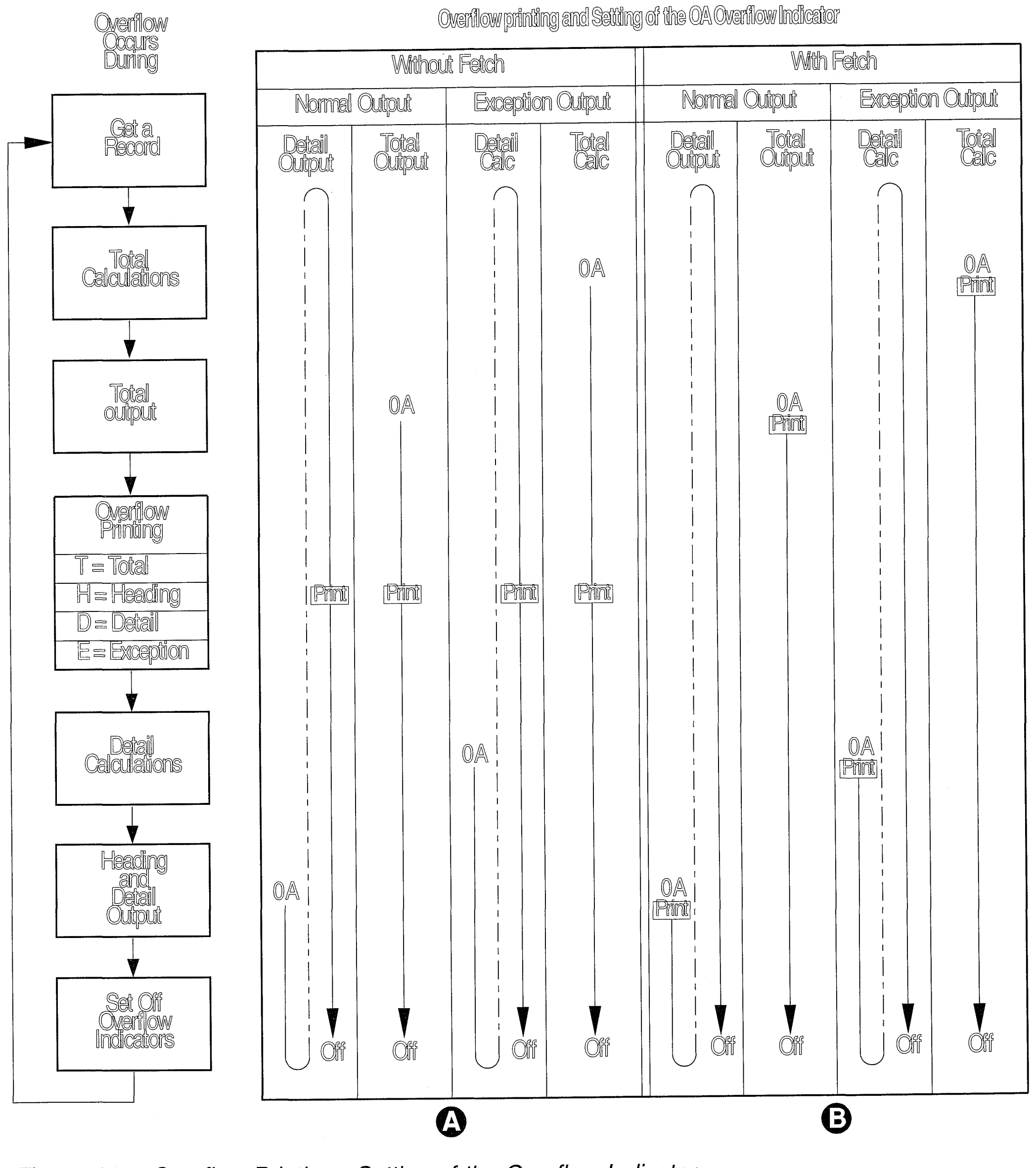


Figure 112. Overflow Printing: Setting of the Overflow Indicator

A When fetch overflow is not specified, the overflow lines print after total output. No matter when overflow occurs (OA is on), the overflow indicator OA remains on through overflow output time and is set off after heading and detail output time.

B When fetch overflow is specified, the overflow lines are written before the output line for which fetch overflow was specified, if the overflow indicator OA is on. When OA is set on, it remains on until after heading and detail output time. The overflow lines are not written a second time at overflow output time unless overflow is sensed again since the last time the overflow lines were written.

Specifying Fetch Overflow

Specify fetch overflow with an F in position 18 of the output specifications on any detail, total, or exception lines for a PRINTER file. The fetch overflow routine does not automatically cause forms to advance to the next page.

During output, the conditioning indicators on an output line are tested to determine if the line is to be written. If the line is to be written and an F is specified in position 18, the compiler tests to determine if the overflow indicator is on. If the overflow indicator is on, the overflow routine is fetched and the following operations occur:

1. Only the overflow lines for the file with the fetch specified are checked for output.
2. All total lines conditioned by the overflow indicator are written.
3. Forms advance to a new page when a skip to a line number less than the line number the printer is currently on is specified in a line conditioned by an overflow indicator.
4. Heading, detail, and exception lines conditioned by the overflow indicator are written.
5. The line that fetched the overflow routine is written.
6. Any detail and total lines left to be written for that program cycle are written.

Position 18 of each OR line must contain an F if the overflow routine is to be used for each record in the OR relationship. Fetch overflow cannot be used if an overflow indicator is specified in positions 21 through 29 of the same specification line. If this is the case, the overflow routine is not fetched.

Example of Specifying Fetch Overflow

Figure 113 shows the use of fetch overflow.

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... *
OFilename++DF..N01N02N03Excnam++++B++A++Sb+Sa+.............................
OPRINTER   H    OA                     3 05
0..............N01N02N03Field+++++++++YB.End++PConstant/editword/DTformat++
0                                           15 'EMPLOYEE TOTAL'
0          TF   L1                     1
0                       EMPLTOT             25
0          T    L1                     1
0                       EMPLTOT             35
0          T    L1                     1
0                       EMPLTOT             45
0          TF   L1                     1
0                       EMPLTOT             55
0          T    L1                     1
0                       EMPLTOT             65
0          T    L1                     1
0                       EMPLTOT             75
0          T    L1                     1
0*
```

Figure 113. Use of Fetch Overflow

The total lines with an F coded in position 18 can fetch the overflow routine. They only do so if overflow is sensed prior to the printing of one of these lines. Before fetch overflow is processed, a check is made to determine whether the overflow indicator is on. If it is on, the overflow routine is fetched, the heading line conditioned by the overflow indicator is printed, and the total operations are processed.

Changing Forms Control Information in a Program-Described File

The PRTCTL (printer control) keyword allows you to change forms control information and to access the current line value within the program for a program-described PRINTER file. Specify the keyword PRTCTL(*data structure name*) on the File Description specification for the PRINTER file.

You can specify two types of PRTCTL data structures in your source: an OPM-defined data structure, or an ILE data structure. The default is to use the ILE data structure layout which is shown in Table 16. To use the OPM-defined data structure layout, specify PRTCTL(*data-structure name*:*COMPAT). The OPM PRTCTL data structure layout is shown in Table 17 on page 247.

The ILE PRTCTL data structure must be defined on the Definition specifications. It requires a minimum of 15 bytes and must contain at least the following five subfields specified in the following order:

Table 16. Layout of ILE PRTCTL Data Structure

Positions	Subfield Contents
1-3	A three-position character field that contains the space-before value (valid values: blank or 0-255)
4-6	A three-position character field that contains the space-after value (valid values: blank or 0-255)
7-9	A three-position character field that contains the skip-before value (valid values: blank or 0-255)
10-12	A three-position character field that contains the skip-after value (valid values: blank or 0-255)
13-15	A three-digit numeric field with zero decimal positions that contains the current line count value.

The OPM PRTCTL data structure must be defined on the Definition specifications and must contain at least the following five subfields specified in the following order:

Table 17. Layout of OPM PRTCTL Data Structure

Positions	Subfield Contents
1	A one-position character field that contains the space-before value (valid values: blank or 0-3)
2	A one-position character field that contains the space-after value (valid values: blank or 0-3)
3-4	A two-position character field that contains the skip-before value (valid values: blank, 1-99, A0-A9 for 100-109, B0-B2 for 110-112)
5-6	A two-position character field that contains the skip-after value (valid values: blank, 1-99, A0-A9 for 100-109, B0-B2 for 110-112)
7-9	A two-digit numeric field with zero decimal positions that contains the current line count value.

The values contained in the first four subfields of the ILE PRTCTL data structure are the same as those allowed in positions 40 through 51 (space and skip entries) of the output specifications. If the space/skip entries (positions 40 through 51) of the output specifications are blank, and if subfields 1 through 4 are also blank, the default is to space 1 after. If the PRTCTL keyword is specified, it is used only for the output records that have blanks in positions 40 through 51. You can control the space and skip value (subfields 1 through 4) for the PRINTER file by changing the values in these subfields of the PRTCTL data structure while the program is running.

Subfield 5 contains the current line count value. The compiler does not initialize subfield 5 until after the first output line is printed. The compiler then changes subfield 5 after each output operation to the file.

Example of Changing Forms Control Information

Figure 114 on page 248 shows an example of the coding necessary to change the forms control information using the PRTCTL keyword.

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... *
FFilename++IPEASFRlen+LKlen+AIDevice+.Keywords+++++++++++++++++++++++++++++
FPRINT     O    F  132        PRINTER PRTCTL(LINE)


*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... *
DName+++++++++++ETDsFrom+++To/L+++IDc.Keywords+++++++++++++++++++++++++++++
DLINE           DS
D SpBefore              1      3
D SpAfter               4      6
D SkBefore              7      9
D SkAfter              10     12
D CurLine              13     15  0


*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... *
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C                   EXCEPT
C   01CurLine       COMP      10                                     49
C   01
CAN 49              MOVE      '3'           SpAfter


*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... *
OFilename++DF..N01N02N03Excnam++++B++A++Sb+Sa+...........................
OPRINT     E    01
O..............N01N02N03Field+++++++++YB.End++PConstant/editword/DTformat++
O                       DATA                25
```

Figure 114. Example of the PRTCTL Option

On the file description specifications, the PRTCTL keyword is specified for the PRINT file. The name of the associated data structure is LINE.

The LINE data structure is defined on the input specifications as having only those subfields that are predefined for the PRTCTL data structure. The first four subfields in positions 1 through 12 are used to supply space and skip information that is generally specified in positions 40 through 51 of the output specifications. The PRTCTL keyword allows you to change these specifications within the program.

In this example, the value in the SpAfter subfield is changed to 3 when the value in the CurLine (current line count value) subfield is equal to 10. (Assume that indicator 01 was set on as a record identifying indicator.)

Accessing Tape Devices

Use the SEQ device specifications whenever you write to a tape file. To write variable-length records to a tape file, use the RCDBLKFMT parameter of the CL command CRTTAPF or OVRTAPF. When you use the RCDBLKFMT parameter, the length of each record to be written to tape is determined by:

- the highest end position specified in the output specifications for the record or,
- if you do not specify an end position, the compiler calculates the record length from the length of the fields.

Read variable-length records from tape just like you would read records from any sequentially organized file. Ensure the record length specified on the file description specification accommodates the longest record in the file.

Accessing Display Devices

You use display files to exchange information between your program and a display device such as a workstation. A display file is used to define the format of the information that is to be presented on a display, and to define how the information is to be processed by the system on its way to and from the display.

See Chapter 17, "Using WORKSTN Files" on page 255 for a discussion on how to use WORKSTN files.

Using Sequential Files

Sequential files in an ILE RPG/400 program associate with any sequentially organized file on the AS/400 system, such as:

- Database file
- Diskette file
- Printer file
- Tape file.

The file name of the SEQ file in the file description specifications points to an AS/400 file. The file description of the AS/400 file specifies the actual I/O device e.g. tape, printer and diskette.

You can also use the CL override commands, for example OVRDBF, OVRDKTF and OVRTAPF, to specify the actual I/O device when the program is run.

Specifying a Sequential File

A sequential (SEQ) device specification, entered in positions 36 through 42 in the file description specification, indicates that the input or output is associated with a sequentially-organized file. Refer to Figure 116 on page 250. The actual device to be associated with the file while running the program can be specified by a OS/400 override command or by the file description that is pointed to by the file name. If SEQ is specified in a program, no device-dependent functions such as space/skip, or CHAIN can be specified.

The following figure shows the operation codes allowed for a SEQ file.

File Description Specifications Positions 17	18	Calculation Specifications Positions 26-35
I	P/S	CLOSE, FEOD
I	F	READ, OPEN, CLOSE, FEOD
O		WRITE, OPEN, CLOSE, FEOD

Note: No print control specifications are allowed for a sequential file.

Figure 115. Valid File Operation Codes for a Sequential File

Example of Specifying a Sequential File

Figure 116 shows an example of how to specify a SEQ file in an ILE RPG/400 source member.

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+ ...*
FFilename++IPEASFRlen+LKlen+AIDevice+.Keywords+++++++++++++++++++++++++++++
FTIMECDS   IP   E            DISK
FPAYOTIME  O    F  132       SEQ
 *
```

Figure 116. SEQ Device

A SEQ device is specified for the PAYOTIME file. When the program is run, you can use a OS/400 override command to specify the actual device (such as printer, tape, or diskette) to be associated with the file while the program is running. For example, diskette can be specified for some program runs while printer can be specified for others. The file description, pointed to by the file name, can specify the actual device, in which case an override command need not be used.

Using SPECIAL Files

The RPG device name SPECIAL (positions 36 - 42 of the file description specifications) allows you to specify an input and/or output device that is not directly supported by the ILE RPG/400 operations. The input and output operations for the file are controlled by a user-written routine. The name of the user-written routine, must be identified in the file description specifications using the keyword PGMNAME('*program name*').

ILE RPG/400 calls this user-written routine to open the file, read and write the records, and close the file. ILE RPG/400 also creates a parameter list for use by the user-written routine. The parameter list contains:

option code parameter (option)
return status parameter (status)
error-found parameter (error)
record area parameter (area).

This parameter list is accessed by the ILE RPG/400 compiler and by the user-written routine; it cannot be accessed by the program that contains the SPECIAL file.

The following describes the parameters in this RPG-created parameter list:

Option The option parameter is a one-position character field that indicates the action the user-written routine is to process. Depending on the operation being processed on the SPECIAL file (OPEN, CLOSE, FEOD, READ, WRITE, DELETE, UPDATE), one of the following values is passed to the user-written routine from ILE RPG/400:

Value Passed	**Description**
O	Open the file.
C	Close the file.
F	Force the end of file.
R	Read a record and place it in the area defined by the area parameter.
W	The ILE RPG/400 program has placed a record in the area defined by the area parameter; the record is to be written out.
D	Delete the record.
U	The record is an update of the last record read.

Status The status parameter is a one-position character field that indicates the status of the user-written routine when control is returned to the ILE RPG/400 program. Status must contain one of the following return values when the user-written routine returns control to the ILE RPG/400 program:

Return Value	**Description**
0	Normal return. The requested action was processed.
1	The input file is at end of file, and no record has been returned. If the file is an output file, this return value is an error.
2	The requested action was not processed; error condition exists.

Error The error parameter is a five-digit zoned numeric field with zero decimal positions. If the user-written routine detects an error, the error parameter contains an indication or value representing the type of error. The value is placed in the first five positions of location *RECORD in the INFDS when the status parameter contains 2.

Area The area parameter is a character field whose length is equal to the record length associated with the SPECIAL file. This field is used to pass the record to or receive the record from the ILE RPG/400 program.

You can add additional parameters to the RPG-created parameter list. Specify the keyword PLIST(*parameter list name*) on the file description specifications for the SPECIAL file. See Figure 118 on page 253. Then use the PLIST operation in the calculation specifications to define the additional parameters.

The user-written routine, specified by the keyword PGMNAME of the file description specifications for the SPECIAL file, must contain an entry parameter list that includes both the RPG-created parameters and the user-specified parameters.

If the SPECIAL file is specified as a primary file, the user-specified parameters must be initialized before the first primary read. You can initialize these parameters with a factor 2 entry on the PARM statements or by the specification of a compile-time array or an array element as a parameter.

Figure 117 shows the file operation codes that are valid for a SPECIAL file.

File Description Specifications Positions 17	18	Calculation Specifications Positions 26-35
I	P/S	CLOSE, FEOD
C	P/S	WRITE, CLOSE, FEOD
U	P/S	UPDAT, DELETE, CLOSE FEOD
O		WRITE, OPEN, CLOSE, FEOD
I	F	READ, WRITE, OPEN, CLOSE, FEOD
C	F	READ, WRITE, OPEN, CLOSE, FEOD
U	F	READ, UPDATE, DELETE, OPEN, CLOSE, FEOD

Figure 117. Valid File Operations for a SPECIAL File

Example of Using a Special File

Figure 118 on page 253 shows how to use the RPG device name SPECIAL in a program. In this example, a file description found in the file EXCPTN is associated with the device SPECIAL.

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... *
FFilename++IPEASFRlen+LKlen+AIDevice+.Keywords+++++++++++++++++++++++++++++
FEXCPTN    O    F   20        SPECIAL PGMNAME('USERIO')
F                                     PLIST(SPCL)

*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... *
DName+++++++++++ETDsFrom+++To/L+++IDc.Functions++++++++++++++++++++++++++++
D OUTBUF          DS
D  FLD                        1     20

*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... *
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....
C     SPCL          PLIST
C                   PARM                    FLD1
C                   MOVEL     'HELLO'       FLD
C                   MOVE      '1'           FLD1                 1
C                   WRITE     EXCPTN        OUTBUF
C                   MOVE      '2'           FLD1                 1
C                   WRITE     EXCPTN        OUTBUF
C                   SETON                                            LR
```

Figure 118. SPECIAL Device

Figure 119 shows the user-written program USERIO.

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... *
CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....

 *-------------------------------------------------------------------*
 * The first 4 parameters are ILE RPG/400 created parameter list. *
 * The rest are defined by the programmer-defined PLIST.           *
 *-------------------------------------------------------------------*
C     *ENTRY        PLIST
C                   PARM                    OPTION               1
C                   PARM                    STATUS               1
C                   PARM                    ERROR                5 0
C                   PARM                    AREA                20
C                   PARM                    FLD1                 1

 *-------------------------------------------------------------------*
 * The user written program will perform the file I/O according    *
 * to the option passed.                                            *
 *-------------------------------------------------------------------*
C                   SELECT
C                   WHEN      OPTION = 'O'
C*  perform OPEN operation
C                   WHEN      OPTION = 'W'
C*  perform WRITE operation
C                   WHEN      OPTION = 'C'
C*  perform CLOSE operation
C                   ENDSL
C                   RETURN
```

Figure 119. User-written program USERIO.

The I/O operations for the SPECIAL device are controlled by the user-written program USERIO. The parameters specified for the programmer-defined PLIST(SPCL) are added to the end of the RPG-created parameter list for the SPECIAL device. The programmer-specified parameters can be accessed by the user ILE RPG/400 program and the user-written routine USERIO; whereas the

RPG-created parameter list can be accessed only by internal ILE RPG/400 logic and the user-written routine.

Chapter 17. Using WORKSTN Files

Interactive applications on the AS/400 generally involve communication with:

- One or more work station users via display files
- One or more programs on a remote system via ICF files
- One or more devices on a remote system via ICF files.

Display files are objects of type *FILE with attribute of DSPF on the AS/400 system. You use display files to communicate interactively with users at display terminals. Like database files, display files can be either externally-described or program-described.

ICF files are objects of type *FILE with attribute of ICFF on the AS/400 system. You use ICF files to communicate with (send data to and receive data from) other application programs on remote systems (AS/400 or non-AS/400). An ICF file contains the communication formats required for sending and receiving data between systems. You can write programs that use ICF files which allow you to communicate with (send data to and receive data from) other application programs on remote systems.

When a file in an RPG program is identified with the WORKSTN device name then that program can communicate interactively with a work-station user or use the Intersystem Communications Function (ICF) to communicate with other programs. This chapter describes how to use:

- Intersystem Communications Function (ICF)
- Externally-described WORKSTN files
- Program-described WORKSTN files
- Multiple-device files.

Intersystem Communications Function

To use the ICF, define a WORKSTN file in your program that refers to an ICF device file. Use either the system supplied file `QICDMF` or a file created using the OS/400 command CRTICFF.

You code for ICF by using the ICF as a file in your program. The ICF is similar to a display file and it contains the communications formats required for the sending and receiving of data between systems.

For further information on the ICF, refer to *ICF Programming*.

Using Externally-Described WORKSTN Files

An RPG WORKSTN file can use an externally-described display-device file or ICF-device file, which contains file information and a description of the fields in the records to be written. The most commonly used externally-described WORKSTN file is a display file. (For information about describing and creating display files, refer to the *DDS Reference*.)

In addition to the field descriptions (such as field names and attributes), the DDS for a display-device file are used to:

- Format the placement of the record on the screen by specifying the line-number and position-number entries for each field and constant.
- Specify attention functions such as underlining and highlighting fields, reverse image, or a blinking cursor.
- Specify validity checking for data entered at the display work station. Validity-checking functions include detecting fields where data is required, detecting mandatory fill fields, detecting incorrect data types, detecting data for a specific range, checking data for a valid entry, and processing modules 10 or 11 check-digit verification.
- Control screen management functions, such as determining if fields are to be erased, overlaid, or kept when new data is displayed.
- Associate indicators 01 through 99 with command attention keys or command function keys. If a function key is described as a command function key (CF), both the response indicator and the data record (with any modifications entered on the screen) are returned to the program. If a function key is described as a command attention key (CA), the response indicator is returned to the program but the data record remains unmodified. Therefore, input-only character fields are blank and input-only numeric field are filled with zeros, unless these fields have been initialized otherwise.
- Assign an edit code (EDTCDE) or edit word (EDTWRD) keyword to a field to specify how the field's values are to be displayed.
- Specify subfiles.

A display-device-record format contains three types of fields:

- *Input fields.* Input fields are passed from the device to the program when the program reads a record. Input fields can be initialized with a default value. If the default value is not changed, the default value is passed to the program. Input fields that are not initialized are displayed as blanks into which the work-station user can enter data.
- *Output fields.* Output fields are passed from the program to the device when the program writes a record to a display. Output fields can be provided by the program or by the record format in the device file.
- *Output/input (both) fields.* An output/input field is an output field that can be changed. It becomes an input field if it is changed. Output/input fields are passed from the program when the program writes a record to a display and passed to the program when the program reads a record from the display. Output/input fields are used when the user is to change or update the data that is written to the display from the program.

If you specify the keyword INDARA in the DDS for a WORKSTN file, the RPG program passes indicators to the WORKSTN file in a separate indicator area, and not in the input/output buffer.

For a detailed description of an externally-described display-device file and for a list of valid DDS keywords, see the *DDS Reference*.

Figure 120 on page 257 shows an example of the DDS for a display-device file.

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ..*
AAN01N02N03T.Name++++++RLen++TDpBLinPosFunctions++++++++++++++++++++*
A** ITEM MASTER INQUIRY
A                                      REF(DSTREF) 1
A          R PROMPT                    TEXT('Item Prompt Format')
A 73N61                                OVERLAY 2
A                                      CA03(98 'End of Program') 3
A                                  1  2'Item Inquiry'
A                                  3  2'Item Number'
A            ITEM      R        I  3 15PUTRETAIN 4
A 61                                   ERRMSG('Invalid Item Number' 61) 5
A          R RESPONSE                  TEXT('Response Format')
A                                      OVERLAY 2
A                                      LOCK 6
A                                  5  2'Description'
A            DESCRP    R           5 15
A                                  5 37'Price'
A            PRICE     R           5 44
A                                  7  2'Warehouse Location' 7
A            WHSLOC    R           7 22
A                                  9  2'On Hand'
A            ONHAND    R           9 10
A                                  9 19'Allocated' 8
A            ALLOC     R           9 30
A                                  9 40'Available'
A            AVAIL     R           9 51
A*
```

Figure 120. Example of the Data Description Specifications for a Display Device File

This display device file contains two record formats: PROMPT and RESPONSE.

1 The attributes for the fields in this file are defined in the DSTREF field reference file.

2 The OVERLAY keyword is used so that both record formats can be used on the same display.

3 Function key 3 is associated with indicator 98, which is used by the programmer to end the program.

4 The PUTRETAIN keyword allows the value that is entered in the ITEM field to be kept in the display. In addition, the ITEM field is defined as an input field by the I in position 38. ITEM is the only input field in these record formats. All of the other fields in the record are output fields since position 38 is blank for each of them.

5 The ERRMSG keyword identifies the error message that is displayed if indicator 61 is set on in the program that uses this record format.

6 The LOCK keyword prevents the work-station user from using the keyboard when the RESPONSE record format is initially-displayed.

7 The constants such as ‘Description’, ‘Price’, and ‘Warehouse Location’ describe the fields that are written out by the program.

8 The line and position entries identify where the fields or constants are written on the display.

Specifying Function Key Indicators on Display Device Files

The function key indicators, KA through KN and KP through KY are valid for a program that contains a display device WORKSTN file if the associated function key is specified in the DDS.

The function key indicators relate to the function keys as follows: function key indicator KA corresponds to function key 1, KB to function key 2 . . . KX to function key 23, and KY to function key 24.

Function keys are specified in the DDS with the CFxx (command function) or CAxx (command attention) keyword. For example, the keyword CF01 allows function key 1 to be used. When you press function key 1, function key indicator KA is set on in the RPG program. If you specify the function key as CF01 (99), both function key indicator KA and indicator 99 are set on in the RPG program. If the work-station user presses a function key that is not specified in the DDS, the OS/400 system informs the user that an incorrect key was pressed.

If the work-station user presses a specified function key, the associated function key indicator in the RPG program is set on when fields are extracted from the record (move fields logic) and all other function key indicators are set off. If a function key is not pressed, all function key indicators are set off at move fields time. The function key indicators are set off if the user presses the Enter key.

Specifying Command Keys on Display Device Files

You can specify the command keys Help, Roll Up, Roll Down, Print, Clear, and Home in the DDS for a display device file with the keywords HELP, ROLLUP, ROLLDOWN, PRINT, CLEAR, and HOME.

Command keys are processed by an RPG program whenever the compiler processes a READ or an EXFMT operation on a record format for which the appropriate keywords are specified in the DDS. When the command keys are in effect and a command key is pressed, the OS/400 system returns control to the RPG program. If a response indicator is specified in the DDS for the command selected, that indicator is set on and all other response indicators that are in effect for the record format and the file are set off.

If a response indicator is not specified in the DDS for a command key, the following happens:

- For the Print key without *PGM specified, the print function is processed.
- For the Roll Up and Roll Down keys used with subfiles, the displayed subfile rolls up or down, within the subfile. If you try to roll beyond the start or end of a subfile, you get a run-time error.
- For the Print Key specified with *PGM, Roll Up and Roll Down keys used without subfiles, and for the Clear, Help, and Home keys, one of the *STATUS values 1121-1126 is set, respectively, and processing continues.

Processing an Externally-Described WORKSTN File

When an externally-described WORKSTN file is processed, the OS/400 system transforms data from the program to the format specified for the file and displays the data. When data is passed to the program, the data is transformed to the format used by the program.

The OS/400 system provides device-control information for processing input/output operations for the device. When an input record is requested from the device, the OS/400 system issues the request, and then removes device-control information from the data before passing the data to the program. In addition, the OS/400 system can pass indicators to the program indicating which fields, or if any fields, in the record have been changed.

When the program requests an output operation, it passes the output record to the OS/400 system. The OS/400 system provides the necessary device-control information to display the record. It also adds any constant information specified for the record format when the record is displayed.

When a record is passed to a program, the fields are arranged in the order in which they are specified in the DDS. The order in which the fields are displayed is based on the display positions (line numbers and position) assigned to the fields in the DDS. The order in which the fields are specified in the DDS and the order in which they appear on the screen need not be the same.

For more information on processing WORKSTN files, see "Valid WORKSTN File Operations" on page 265.

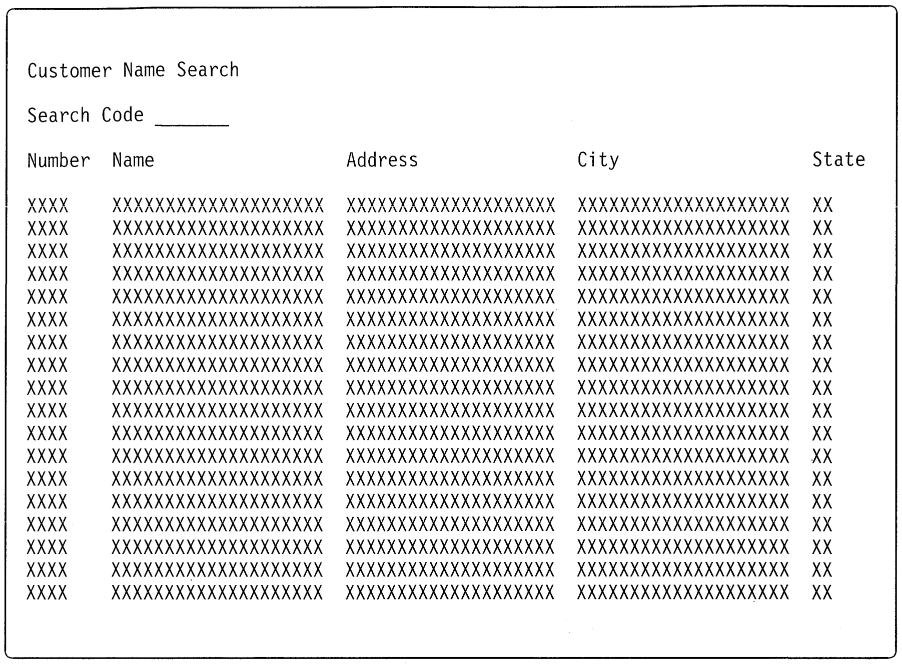
Using Subfiles

Subfiles can be specified in the DDS for a display-device file to allow you to handle multiple records of the same type on the display. (See Figure 121 on page 260.) A subfile is a group of records that is read from or written to a display-device file. For example, a program reads records from a database file and creates a subfile of output records. When the entire subfile has been written, the program sends the entire subfile to the display device in one write operation. The work-station user can change data or enter additional data in the subfile. The program then reads the entire subfile from the display device into the program and processes each record in the subfile individually.

Records that you want to be included in a subfile are specified in the DDS for the file. The number of records that can be included in a subfile must also be specified in the DDS. One file can contain more than one subfile, and up to 12 subfiles can be active concurrently. Two subfiles can be displayed at the same time.

The DDS for a subfile consists of two record formats: a subfile-record format and a subfile control-record format. The subfile-record format contains the field information that is transferred to or from the display file under control of the subfile control-record format. The subfile control-record format causes the physical read, write, or control operations of a subfile to take place. Figure 122 on page 261 shows an example of the DDS for a subfile-record format, and Figure 123 on page 262 shows an example of the DDS for a subfile control-record format.

For a description of how to use subfile keywords, see the *DDS Reference*.



```
Customer Name Search

Search Code ______

Number  Name                  Address               City                  State

XXXX    XXXXXXXXXXXXXXXXXXXX  XXXXXXXXXXXXXXXXXXXX  XXXXXXXXXXXXXXXXXXXX  XX
XXXX    XXXXXXXXXXXXXXXXXXXX  XXXXXXXXXXXXXXXXXXXX  XXXXXXXXXXXXXXXXXXXX  XX
XXXX    XXXXXXXXXXXXXXXXXXXX  XXXXXXXXXXXXXXXXXXXX  XXXXXXXXXXXXXXXXXXXX  XX
XXXX    XXXXXXXXXXXXXXXXXXXX  XXXXXXXXXXXXXXXXXXXX  XXXXXXXXXXXXXXXXXXXX  XX
XXXX    XXXXXXXXXXXXXXXXXXXX  XXXXXXXXXXXXXXXXXXXX  XXXXXXXXXXXXXXXXXXXX  XX
XXXX    XXXXXXXXXXXXXXXXXXXX  XXXXXXXXXXXXXXXXXXXX  XXXXXXXXXXXXXXXXXXXX  XX
XXXX    XXXXXXXXXXXXXXXXXXXX  XXXXXXXXXXXXXXXXXXXX  XXXXXXXXXXXXXXXXXXXX  XX
XXXX    XXXXXXXXXXXXXXXXXXXX  XXXXXXXXXXXXXXXXXXXX  XXXXXXXXXXXXXXXXXXXX  XX
XXXX    XXXXXXXXXXXXXXXXXXXX  XXXXXXXXXXXXXXXXXXXX  XXXXXXXXXXXXXXXXXXXX  XX
XXXX    XXXXXXXXXXXXXXXXXXXX  XXXXXXXXXXXXXXXXXXXX  XXXXXXXXXXXXXXXXXXXX  XX
XXXX    XXXXXXXXXXXXXXXXXXXX  XXXXXXXXXXXXXXXXXXXX  XXXXXXXXXXXXXXXXXXXX  XX
XXXX    XXXXXXXXXXXXXXXXXXXX  XXXXXXXXXXXXXXXXXXXX  XXXXXXXXXXXXXXXXXXXX  XX
XXXX    XXXXXXXXXXXXXXXXXXXX  XXXXXXXXXXXXXXXXXXXX  XXXXXXXXXXXXXXXXXXXX  XX
XXXX    XXXXXXXXXXXXXXXXXXXX  XXXXXXXXXXXXXXXXXXXX  XXXXXXXXXXXXXXXXXXXX  XX
XXXX    XXXXXXXXXXXXXXXXXXXX  XXXXXXXXXXXXXXXXXXXX  XXXXXXXXXXXXXXXXXXXX  XX
XXXX    XXXXXXXXXXXXXXXXXXXX  XXXXXXXXXXXXXXXXXXXX  XXXXXXXXXXXXXXXXXXXX  XX
XXXX    XXXXXXXXXXXXXXXXXXXX  XXXXXXXXXXXXXXXXXXXX  XXXXXXXXXXXXXXXXXXXX  XX
XXXX    XXXXXXXXXXXXXXXXXXXX  XXXXXXXXXXXXXXXXXXXX  XXXXXXXXXXXXXXXXXXXX  XX
```

Figure 121. Subfile Display

To use a subfile for a display device file in an RPG program, you must specify the SFILE keyword on a file description specification for the WORKSTN file. The format of the SFILE keyword is SFILE(*record format name*:*RECNO field name*). The WORKSTN file must be an externally-described file (E in position 22).

You must specify for the SFILE keyword the name of the subfile record format (not the control-record format) and the name of the field that contains the relative record number to be used in processing the subfile.

In an RPG program, relative record number processing is defined as part of the SFILE definition. The SFILE definition implies a full-procedural update file with ADD for the subfile. Therefore, the file operations that are valid for the subfile are not dependent on the definition of the main WORKSTN file. That is, the WORKSTN file can be defined as a primary file or a full-procedural file.

Use the CHAIN, READC, UPDATE, or WRITE operation codes with the subfile record format to transfer data between the program and the subfile. Use the READ, WRITE, or EXFMT operation codes with the subfile control-record format to transfer data between the program and the display device or to process subfile control operations.

Subfile processing follows the rules for relative-record-number processing. The RPG program places the relative-record number of any record retrieved by a READC operation into the field named in the second position of the SFILE keyword. This field is also used to specify the record number that the RPG program uses for WRITE operation to the subfile or for output operations that use ADD. The RECNO field name specified for the SFILE keyword must be defined as numeric with zero decimal positions. The field must have enough positions to contain the largest record number for the file. (See the SFLSIZ keyword in the *DDS Reference*.) The WRITE operation code and the ADD specification on the output specifications require that a relative-record-number field be specified in the second position of the SFILE keyword on the file description specification.

If a WORKSTN file has an associated subfile, all implicit input operations and explicit calculation operations that refer to the file name are processed against the main WORKSTN file. Any operations that specify a record format name that is not designated as a subfile are processed on the main WORKSTN file.

If you press a specified function key during a read of a non-subfile record, subsequent reads of a subfile record will cause the corresponding function key indicator to be set on again, even if the function key indicator has been set off between the reads. This will continue until a non-subfile record is read from the WORKSTN file.

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ..*
AAN01N02N03T.Name++++++RLen++TDpBLinPosFunctions++++++++++++++++++++*
A** CUSTOMER NAME SEARCH
A                                      REF(DSTREF) 1
A          R SUBFIL                    SFL 2
A                                      TEXT('Subfile Record')
A            CUST      R             7  3
A            NAME      R             7 10
A            ADDR      R             7 32 3
A            CITY      R             7 54
A            STATE     R             7 77
A*
```

Figure 122. Data Description Specifications for a Subfile Record Format

The data description specifications (DDS) for a subfile record format describe the records in the subfile:

1 The attributes for the fields in the record format are contained in the field reference file DSTREF as specified by the REF keyword.

2 The SFL keyword identifies the record format as a subfile.

3 The line and position entries define the location of the fields on the display.

Use of Subfiles

Some typical ways you can make use of subfiles include:

- Display only. The work-station user reviews the display.
- Display with selection. The user requests more information about one of the items on the display.
- Modification. The user changes one or more of the records.
- Input only, with no validity checking. A subfile is used for a data entry function.
- Input only, with validity checking. A subfile is used for a data entry function, but the records are checked.
- Combination of tasks. A subfile can be used as a display with modification, plus the input of new records.

The following figure shows an example of data description specifications for a subfile control-record format. For an example of using a subfile in an RPG program, see “Search by Zip Code” on page 284.

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ..*
AAN01N02N03T.Name++++++RLen++TDpBLinPosFunctions++++++++++++++++++++*
A          R FILCTL                    SFLCTL(SUBFIL)
A N70                                  SFLCLR
A  70                                  SFLDSPCTL
A  71                                  SFLDSP
A                                      SFLSIZ(15)
A                                      SFLPAG(15)
A                                      TEXT('Subfile Control Record')
A                                      OVERLAY
A  71                                  ROLLUP(97 'Continue Search')
A                                      CA01(98 'End of Program')
A                                      HELP(99 'Help Key')
A                                  1  2'Customer Name Search'
A                                  3  2'Search Code'
A            SRHCOD    R        I  3 14PUTRETAIN
A                                  5  2'Number'
A                                  5 10'Name'
A                                  5 32'Address'
A                                  5 54'City'
A                                  5 76'State'
A*
```

Figure 123. Data Description Specifications for a Subfile Control-Record Format

The subfile control-record format defines the attributes of the subfile, the search input field, constants, and function keys. The keywords you can use indicate the following:

- SFLCTL names the associated subfile (SUBFIL).
- SFLCLR indicates when the subfile should be cleared (when indicator 70 is off).
- SFLDSPCTL indicates when to display the subfile control record (when indicator 70 is on).
- SFLDSP indicates when to display the subfile (when indicator 71 is on).
- SFLSIZ indicates the total number of records to be included in the subfile (15).
- SFLPAG indicates the total number of records in a page (15).
- ROLLUP indicates that indicator 97 is set on in the program when the user presses the Roll Up key.
- HELP allows the user to press the Help key for a displayed message that describes the valid function keys.
- PUTRETAIN allows the value that is entered in the SRHCOD field to be kept in the display.

In addition to the control information, the subfile control-record format also defines the constants to be used as column headings for the subfile record format.

Using Program-Described WORKSTN Files

You can use a program-described WORKSTN file with or without a format name specified on the output specifications. The format name, if specified, refers to the name of a data description specifications record format. This record format describes:

- How the data stream sent from an RPG program is formatted on the screen

- What data is sent
- What ICF functions to perform.

If a format name is used, input and output specifications must be used to describe the input and output records.

You can specify PASS(*NOIND) on a file description specification for a program-described WORKSTN file. The PASS(*NOIND) keyword indicates that the RPG program will not additionally pass indicators to data management on output or receive them on input. It is your responsibility to pass indicators by describing them as fields (in the form *INxx, *IN, or *IN(x)) in the input or output record. They must be specified in the sequence required by the data description specifications (DDS). You can use the DDS listing to determine this sequence.

Using a Program-Described WORKSTN File with a Format Name

The following specifications apply to using a format name for a program-described WORKSTN file.

Output Specifications: On the output specifications, you must specify the WORKSTN file name in positions 7 through 16. The format name, which is the name of the DDS record format, is specified as a literal or named constant in positions 53 through 80 on the succeeding field description line. K1 through K10 must be specified (right-adjusted) in positions 47 through 51 on the line containing the format name. The K identifies the entry as a length rather than an end position, and the number indicates the length of the format name. For example, if the format name is CUSPMT, the entry in positions 47 through 51 is K6. (Leading zeros following the K are allowed.) The format name cannot be conditioned (indicators in positions 21 through 29 are not valid).

Output fields must be located in the output record in the same order as defined in the DDS; however, the field names do not have to be the same. The end position entries for the fields refer to the end position in the output record passed from the RPG program to data management, and not to the location of the fields on the screen.

To pass indicators on output, do one of the following:

- Specify the keyword INDARA in the DDS for the WORKSTN file. Do not use the PASS(*NOIND) keyword on the file description specification and do not specify the indicators on the output specifications. The program and file use a separate indicator area to pass the indicators.
- Specify the PASS(*NOIND) keyword on the file description specification. Specify the indicators in the output specifications as fields in the form *INxx. The indicator fields must precede other fields in the output record, and they must appear in the order specified by the WORKSTN file DDS. You can determine this order from the DDS listing.

Input Specifications: The input specifications describe the record that the RPG program receives from the display or ICF device. The WORKSTN file name must be specified in positions 7 through 16. Input fields must be located in the input record in the same sequence as defined in the DDS; however, the field names do not have to be the same. The field location entries refer to the location of the fields in the input record.

To receive indicators on input, do one of the following:

- Specify the keyword INDARA in the DDS for the WORKSTN file. Do not use the PASS(*NOIND) keyword on the file description specification and do not specify the indicators on the input specifications. The program and file use a separate indicator area to pass the indicators.
- Specify the PASS(*NOIND) keyword on the file description specification. Specify the indicators in the input specifications as fields in the form *INxx. They must appear in the input record in the order specified by the WORKSTN file DDS. You can determine this order from the DDS listing.

A record identifying indicator should be assigned to each record in the file to identify the record that has been read from the WORKSTN file. A hidden field with a default value can be specified in the DDS for the record identification code.

Calculation Specifications: The operation code READ is valid for a program-described WORKSTN file that is defined as a combined, full-procedural file. See Figure 124 on page 265. The file name must be specified in factor 2 for this operation. A format must exist at the device before any input operations can take place. This requirement can be satisfied on a display device by conditioning an output record with 1P or by writing the first format to the device in another program (for example, in the CL program). The EXFMT operation is not valid for a program-described WORKSTN file. You can also use the EXCEPT operation to write to a WORKSTN file.

Additional Considerations: When using a format name with a program-described WORKSTN file, you must also consider the following:

- The name specified in positions 53 through 80 of the output specifications is assumed to be the name of a record format in the DDS that was used to create the file.
- If a Kn specification is present for an output record, it must also be used for any other output records for that file. If a Kn specification is not used for all output records to a file, a run-time error will occur.

Using a Program-Described WORKSTN File without a Format Name

When a record-format name is not used, a program-described display-device file describes a file containing one record-format description with one field. The fields in the record must be described within the program that uses the file.

When you create the display file by using the Create Display File command, the file has the following attributes:

- A variable record length can be specified; therefore, the actual record length must be specified in the using program. (The maximum record length allowed is the screen size minus one.)
- No indicators are passed to or from the program.
- No function key indicators are defined.
- The record is written to the display beginning in position 2 of the first available line.

Input File: For an input file, the input record, which is treated by the OS/400 device support as a single input field, is initialized to blanks when the file is opened. The cursor is positioned at the beginning of the field, which is position 2 on the display.

Output File: For an output file, the OS/400 device support treats the output record as a string of characters to be sent to the display. Each output record is written as the next sequential record in the file; that is, each record displayed overlays the previous record displayed.

Combined File: For a combined file, the record, which is treated by the OS/400 device support as a single field, appears on the screen and is both the output record and the input record. Device support initializes the input record to blanks, and the cursor is placed in position 2.

For more information on program-described-display-device files, see *Data Management*.

Valid WORKSTN File Operations

Figure 124 shows the valid file operation codes for a WORKSTN file.

File-Description Specifications Positions		**Calculation Specifications Positions**
17	18	26-35
I	P/S	CLOSE, ACQ, REL, NEXT, POST, FORCE
I	P/S	WRITE[1], CLOSE, ACQ, REL, NEXT, POST, FORCE
I	F	READ, OPEN, CLOSE, ACQ, REL, NEXT, POST
C	F	READ, WRITE[1], EXFMT[2], OPEN, CLOSE, ACQ, REL, NEXT, POST, UPDATE[3], CHAIN[3], READC[3]
O	Blank	WRITE[1], OPEN, CLOSE, ACQ, REL, POST
Note: [1]The WRITE operation is not valid for a program-described file used with a format name.		
Note: [2]If the EXFMT operation is used, the file must be externally described (an E in position 19 of the file description specifications).		
Note: [3]For subfile record formats, the UPDATE, CHAIN, and READC operations are also valid.		

Figure 124. Valid File Operation Codes for a WORKSTN File

The following further explains the EXFMT, READ, and WRITE operation codes when used to process a WORKSTN file.

EXFMT Operation

The EXFMT operation is a combination of a WRITE followed by a READ to the same record format (it corresponds to a data management WRITE-READ operation). If you define a WORKSTN file on the file description specifications as a full-procedural (F in position 18) combined file (C in position 17) that uses externally-described data (E in position 22) the EXFMT (execute format) operation code can be used to write and read from the display.

READ Operation

The READ operation is valid for a full-procedural combined file or a full-procedural input file that uses externally-described data or program-described data. The READ operation retrieves a record from the display. However, a format must exist at the device before any input operations can occur. This requirement can be satisfied on a display device by conditioning an output record with the 1P indicator, by writing the first format to the device from another program, or, if the read is by record-format name, by using the keyword INZRCD on the record description in the DDS.

WRITE Operation

The WRITE operation writes a new record to a display and is valid for a combined file or an output file. Output specifications and the EXCEPT operation can also be used to write to a WORKSTN file. See the *ILE RPG/400 Reference* for a complete description of each of these operation codes.

Multiple-Device Files

Any RPG WORKSTN file with at least one of the keywords DEVID, SAVEIND, MAXDEV(*FILE) or SAVEDS specified on the file description specification is a multiple-device file. Through a multiple-device file, your program may access more than one device.

The RPG program accesses devices through program devices, which are symbolic mechanisms for directing operations to an actual device. When you create a file (using the DDS and commands such as the create file commands), you consider such things as which device is associated with a program device, whether or not a file has a requesting program device, which record formats will be used to invite devices to respond to a READ-by-file-name operation, and how long this READ operation will wait for a response. For detailed information on the options and requirements for creating a multiple-device file, see the chapter on display files in *Data Management*, and information on ICF files in *ICF Programming*, and the manuals you are referred to in these two publications.

With multiple-device files, you make particular use of the following operation codes:

- In addition to opening a file, the OPEN operation implicitly acquires the device you specify when you create the file.
- The ACQ (acquire) operation acquires any other devices for a multiple-device file.
- The REL (release) operation releases a device from the file.
- The WRITE operation, when used with the DDS keyword INVITE, invites a program device to respond to subsequent read-from-invited- program-devices operations. See the sections on inviting a program device in *ICF Programming* and *Data Management*.
- The READ operation either processes a read-from-invited-program-devices operation or a read-from-one-program-device operation. When no NEXT operation is in effect, a program-cycle-read or READ-by-file-name operation waits for input from any of the devices that have been invited to respond (read-from-invited-program-device). Other input and output operations, including a READ-by-file-name after a NEXT operation, and a READ-by-format-name, process a read-from-one-program-device operation using the program device

indicated in a special field. (The field is named in the DEVID keyword of the file description specification lines.)

This device may be the device used on the last input operation, a device you specify, or the requesting program device. See the sections on reading from invited program devices and on reading from one program device in *ICF Programming* and *Data Management*.

- The NEXT operation specifies which device is to be used in the next READ-by-file-name operation or program-cycle-read operation.
- The POST operation puts information in the INFDS information data structure. The information may be about a specific device or about the file. (The POST operation is not restricted to use with multiple-device files.)

See the *ILE RPG/400 Reference* for details of the RPG operation codes.

On the file description specification you can specify several keywords to control the processing of multiple-device files.

- The MAXDEV keyword indicates whether it is a single or multiple device file.

  Specify MAXDEV(*FILE) to process a multiple device file with the maximum number of devices taken from the definition of the file being processed. Specify MAXDEV(*ONLY) to process only one device.

- The DEVID keyword allows you to specify the name of a program device to which input and output operations are directed.

  When a read-from-one-program-device or WRITE operation is issued, the device used for the operation is the device specified as the parameter to the DEVID keyword. This field is initialized to blanks and is updated with the name of the device from which the last successful input operation occurred. It can also be set explicitly by moving a value to it. The ACQ operation code does not affect the value of this field. If the DEVID keyword is not specified, the input operation is performed against the device from which the last successful input operation occurred. A blank device name is used if a read operation has not yet been performed successfully from a device.

  When a read-from-one-program device or WRITE operation is issued with a blank device name, the RPG compiler implicitly uses the device name of the requestor device for the program. If you call an RPG program interactively and acquire an ICF device against which you want to perform one of these operations, you must explicitly move the device name of the ICF device into the field name specified with the DEVID keyword prior to performing the operation. If this is not done, the device name used will either be blank (in which case the interactive requestor device name is used), or the device name used is the one from the last successful input operation. Once you have performed an I/O operation to the ICF device, you do not need to modify the value again unless an input operation completes successfully with a different device.

- The SAVEDS keyword indicates a data structure that is saved and restored for each device acquired to a file. The SAVEIND keyword indicates a set of indicators to be saved and restored for each device acquired to a file. Before an input operation, the current set of indicators and data structure are saved. After the input operation, the RPG compiler restores the indicators and data structure for the device associated with the operation. This may be a different set of indicators or data structure than was available before the input operation.

- The INFDS keyword specifies the file information data structure for the WORKSTN file. The RPG *STATUS field and the major/minor return code for the I/O operation can be accessed through this data structure. Particularly when ICF is being used, both fields are useful for detecting errors that occurred during I/O operations to multiple-device files.

  Note: When specifying these control options, you must code the MAXDEV option before the DEVID, SAVEIND or SAVEDS options.

Chapter 18. Example of an Interactive Application

This chapter illustrates some common workstation applications and their ILE RPG/400 coding.

The application program presented in this chapter consists of four modules. Each module illustrates a common use for WORKSTN files. The first module (CUSMAIN) provides the main menu for the program. Based on the user's selection, it calls the procedure in the appropriate module which provides the function requested.

Each module uses a WORKSTN file to prompt the user for input and display information on the screen. Each module, except for the main module CUSMAIN, also uses a logical file which presents a *view* of the master database file. This view consists of only the fields of the master file which the module requires for its processing.

Note: Each module, except CUSMAIN, can be compiled as a free standing program, that is, they can each be used as an independent program.

Table 18. Description of Each Module in the Interactive Application Example

Module	**Description**
"Main Menu Inquiry" on page 271	An example of a basic menu inquiry program that uses a WORKSTN file to display menu choices and accept input.
"File Maintenance" on page 274	An example of a maintenance program which allows customer records in a master file to be updated, deleted, added, and displayed.
"Search by Zip Code" on page 284	An example program which uses WORKSTN subfile processing to display all matched records for a specified zip code.
"Search and Inquiry by Name" on page 291	An example program which uses WORKSTN subfile processing to display all matched records for a specified customer name, and then allows the user to select a record from the subfile to display the complete customer information.

Database Physical File

Figure 125 shows the data description specifications (DDS) for the master customer file. This file contains important information for each customer, such as name, address, account balance, and customer number. Every module which requires customer information uses this database file (or a logical view of it).

```
A*****************************************************************
A*     FILE NAME:  CUSMST                                        *
A*  RELATED PGMS:  CUSMNT, SCHZIP, SCHNAM                        *
A* RELATED FILES:  CUSMSTL1, CUSMSTL2, CUSMSTL3 (LOGICAL FILES)  *
A*   DESCRIPTION:  THIS IS THE PHYSICAL FILE CUSMST. IT HAS      *
A*                 ONE RECORD FORMAT CALLED CUSREC.              *
A*****************************************************************
A* CUSTOMER MASTER FILE -- CUSMST
A          R CUSREC
A            CUST           5  0       TEXT('CUSTOMER NUMBER')
A            NAME          20          TEXT('CUSTOMER NAME')
A            ADDR1         20          TEXT('CUSTOMER ADDRESS')
A            ADDR2         20          TEXT('CUSTOMER ADDRESS')
A            CITY          20          TEXT('CUSTOMER CITY')
A            STATE          2          TEXT('CUSTOMER STATE')
A            ZIP            5  0       TEXT('CUSTOMER ZIP CODE')
A            ARBAL         10  2       TEXT('ACCOUNTS RECEIVABLE BALANCE')
```

Figure 125. DDS for master database file CUSMST (physical file)

Main Menu Inquiry

The following illustrates a simple inquiry program using a WORKSTN file to display menu choices and accept input.

MAINMENU: DDS for a Display Device File

The DDS for the MAINMENU display device file specifies file level entries and describe one record format: HDRSCN. The file level entries define the screen size (DSPSIZ), input defaults (CHGINPDFT), print key (PRINT), and a separate indicator area (INDARA).

The HDRSCN record format contains the constant 'CUSTOMER MAIN INQUIRY', which identifies the display. It also contains the keywords TIME and DATE, which will display the current time and date on the screen. The CA keywords define the function keys that can be used and associate the function keys with indicators in the RPG program.

```
A*****************************************************************
A*     FILE NAME:  MAINMENU                                      *
A*  RELATED PGMS:  CUSMAIN                                       *
A*   DESCRIPTION:  THIS IS THE DISPLAY FILE MAINMENU. IT HAS 1   *
A*                 RECORD FORMAT CALLED HDRSCN.                  *
A*****************************************************************
A                                      DSPSIZ(24 80 *DS3)
A                                      CHGINPDFT(CS)
A                                      PRINT(QSYSPRT)
A                                      INDARA
A          R HDRSCN
A                                      CA03(03 'END OF INQUIRY')
A                                      CA05(05 'MAINTENANCE MODE')
A                                      CA06(06 'SEARCH BY ZIP MODE')
A                                      CA07(07 'SEARCH BY NAME MODE')
A                                  2  4TIME
A                                      DSPATR(HI)
A                                  2 28'CUSTOMER MAIN INQUIRY'
A                                      DSPATR(HI)
A                                      DSPATR(RI)
A                                  2 70DATE
A                                      EDTCDE(Y)
A                                      DSPATR(HI)
A                                  6  5'Press one of the following'
A                                  6 32'PF keys.'
A                                  8 22'F3 End Job'
A                                  9 22'F5 Maintain Customer File'
A                                 10 22'F6 Search Customer by Zip Code'
A                                 11 22'F7 Search Customer by Name'
```

Figure 126. DDS for display device file MAINMENU

In addition to describing the constants, fields, line numbers, and horizontal positions for the screen, the record formats also define the display attributes for these entries.

Note: Normally, the field attributes are defined in a field-reference file rather than in the DDS for a file. The attributes are shown on the DDS so you can see what they are.

CUSMAIN: RPG Source

```
*****************************************************************
*  PROGRAM NAME:  CUSMAIN                                       *
* RELATED FILES:  MAINMENU (DSPF)                               *
*  RELATED PGMS:  CUSMNT   (ILE RPG PGM)                        *
*                 SCHZIP   (ILE RPG PGM)                        *
*                 SCHNAM   (ILE RPG PGM)                        *
*   DESCRIPTION:  THIS IS A CUSTOMER MAIN INQUIRY PROGRAM.      *
*                 IT PROMPTS THE USER TO CHOOSE FROM ONE OF THE *
*                 FOLLOWING ACTIONS:                            *
*                 1.MAINTAINS (ADD, UPDATE, DELETE AND DISPLAY) *
*                   CUSTOMER RECORD.                            *
*                 2.SEARCH CUSTOMER RECORD BY ZIP CODE.         *
*                 3.SEARCH CUSTOMER RECORD BY NAME.             *
*****************************************************************

FMAINMENU  CF   E             WORKSTN

C                   EXFMT     HDRSCN
C*
C                   DOW       NOT *IN03
C                   SELECT
C                   WHEN      *IN05
C                   CALLB     'CUSMNT'
C                   WHEN      *IN06
C                   CALLB     'SCHZIP'
C                   WHEN      *IN07
C                   CALLB     'SCHNAM'
C                   ENDSL
C                   EXFMT     HDRSCN
C                   ENDDO
C*
C                   SETON                                        LR
```

Figure 127. Source for module CUSMAIN

This module illustrates the use of the CALLB opcode. The appropriate RPG module (CUSMNT, SCHZIP, or SCHNAM) is called by CUSMAIN depending on the user's menu item selection.

To create the program object:

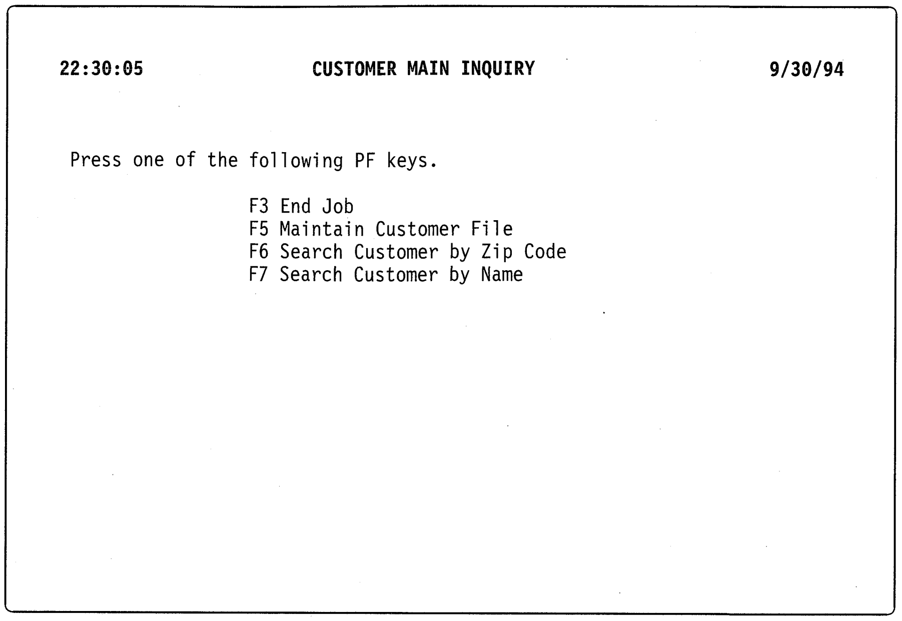
1. Create a module for each source member (CUSMAIN, CUSMNT, SCHZIP, and SCHNAM) using CRTRPGMOD.
2. Create the program by entering:

   ```
   CRTPGM PGM(MYPROG) MODULE(CUSMAIN CUSMNT SCHZIP SCHNAM) ENTMOD(*FIRST)
   ```

   Note: The *FIRST option specifies that the first module in the list, CUSMAIN, is selected as the program entry procedure.
3. Call the program by entering:

   ```
   CALL MYPROG
   ```

The "main menu" will appear as in Figure 128 on page 273.



```
22:30:05                CUSTOMER MAIN INQUIRY                9/30/94


 Press one of the following PF keys.

                  F3 End Job
                  F5 Maintain Customer File
                  F6 Search Customer by Zip Code
                  F7 Search Customer by Name
```

Figure 128. Customer Main Inquiry prompt screen

File Maintenance

The following illustrates a maintenance program using the WORKSTN file. It allows you to add, delete, update, and display records of the master customer file.

CUSMSTL1: DDS for a Logical File

```
A*****************************************************************
A*     FILE NAME:  CUSMSTL1                                      *
A*  RELATED PGMS:  CUSMNT                                        *
A* RELATED FILES:  CUSMST  (PHYSICAL FILE)                        *
A*   DESCRIPTION:  THIS IS LOGICAL FILE CUSMSTL1.                *
A*                 IT CONTAINS ONE RECORD FORMAT CALLED CMLREC1. *
A*                 LOGICAL VIEW OF CUSTOMER MASTER FILE (CUSMST) *
A*                 BY CUSTOMER NUMBER (CUST)                     *
A*****************************************************************
A          R CMLREC1                   PFILE(CUSMST)
A            CUST
A            NAME
A            ADDR1
A            ADDR2
A            CITY
A            STATE
A            ZIP
A          K CUST
```

Figure 129. DDS for logical file CUSMSTL1

The DDS for the database file used by this program describe one record format: CMLREC1. Each field in the record format is described, and the CUST field is identified as the key field for the record format.

MNTMENU: DDS for a Display Device File

```
A*****************************************************************
A*     FILE NAME:  MNTMENU                                       *
A*  RELATED PGMS:  CUSMNT                                        *
A* RELATED FILES:  CUSMSTL1   (LOGICAL FILE)                     *
A*   DESCRIPTION:  THIS IS THE DISPLAY FILE MNTMENU. IT HAS 3    *
A*                 RECORD FORMATS.                               *
A*****************************************************************
A                                      REF(CUSMSTL1)
A                                      CHGINPDFT(CS)
A                                      PRINT(QSYSPRT)
A                                      INDARA
```

Figure 130 (Part 1 of 3). DDS for display device file MNTMENU

```
     A          R HDRSCN
     A                                       TEXT('PROMPT FOR CUST NUMBER')
     A                                       CA03(03 'END MAINTENANCE')
     A                                       CA05(05 'ADD MODE')
     A                                       CA06(06 'UPDATE MODE')
     A                                       CA07(07 'DELETE MODE')
     A                                       CA08(08 'DISPLAY MODE')
     A            MODE           8A  O  1  4DSPATR(HI)
     A                                  1 13'MODE'
     A                                       DSPATR(HI)
     A                                  2  4TIME
     A                                       DSPATR(HI)
     A                                  2 28'CUSTOMER FILE MAINTENANCE'
     A                                       DSPATR(HI RI)
     A                                  2 70DATE
     A                                       EDTCDE(Y)
     A                                       DSPATR(HI)
     A            CUST      R      Y  I 10 25DSPATR(CS)
     A                                       CHECK(RZ)
     A  51                                   ERRMSG('CUSTOMER ALREADY ON +
     A                                       FILE' 51)
     A  52                                   ERRMSG('CUSTOMER NOT ON FILE' +
     A                                       52)
     A                                 10 33'<--Enter Customer Number'
     A                                       DSPATR(HI)
     A                                 23  4'F3 End Job'
     A                                 23 21'F5 Add'
     A                                 23 34'F6 Update'
     A                                 23 50'F7 Delete'
     A                                 23 66'F8 Display'
     A          R CSTINQ
     A                                       TEXT('DISPLAY CUST INFO')
     A                                       CA12(12 'PREVIOUS SCREEN')
     A            MODE           8A  O  1  4DSPATR(HI)
     A                                  1 13'MODE'
     A                                       DSPATR(HI)
     A                                  2  4TIME
     A                                       DSPATR(HI)
     A                                  2 28'CUSTOMER FILE MAINTENANCE'
     A                                       DSPATR(HI)
     A                                       DSPATR(RI)
     A                                  2 70DATE
     A                                       EDTCDE(Y)
     A                                       DSPATR(HI)
     A                                  4 14'Customer:'
     A                                       DSPATR(HI)
     A                                       DSPATR(UL)
```

Figure 130 (Part 2 of 3). DDS for display device file MNTMENU

```
A            CUST      R        O  4 25DSPATR(HI)
A            NAME      R        B  6 25DSPATR(CS)
A  04                                  DSPATR(PR)
A            ADDR1     R        B  7 25DSPATR(CS)
A  04                                  DSPATR(PR)
A            ADDR2     R        B  8 25DSPATR(CS)
A  04                                  DSPATR(PR)
A            CITY      R        B  9 25DSPATR(CS)
A  04                                  DSPATR(PR)
A            STATE     R        B 10 25DSPATR(CS)
A  04                                  DSPATR(PR)
A            ZIP       R        B 10 40DSPATR(CS)
A                                      EDTCDE(Z)
A  04                                  DSPATR(PR)
A                                 23  2'F12 Cancel'
A            MODE1          8   O 23 13
A          R CSTBLD                    TEXT('ADD CUST RECORD')
A                                      CA12(12 'PREVIOUS SCREEN')
A            MODE           8   O  1  4DSPATR(HI)
A                                  1 13'MODE'     DSPATR(HI)
A                                  2  4TIME
A                                      DSPATR(HI)
A                                  2 28'CUSTOMER FILE MAINTENANCE'
A                                      DSPATR(HI RI)
A                                  2 70DATE
A                                      EDTCDE(Y)
A                                      DSPATR(HI)
A                                  4 14'Customer:' DSPATR(HI UL)
A            CUST      R        O  4 25DSPATR(HI)
A                                  6 20'Name'     DSPATR(HI)
A            NAME      R        I  6 25
A                                  7 17'Address'  DSPATR(HI)
A            ADDR1     R        I  7 25
A                                  8 17'Address'  DSPATR(HI)
A            ADDR2     R        I  8 25
A                                  9 20'City'     DSPATR(HI)
A            CITY      R        I  9 25
A                                 10 19'State'    DSPATR(HI)
A            STATE     R        I 10 25
A                                 10 36'Zip'      DSPATR(HI)
A            ZIP       R     Y  I 10 40
A                                 23  2'F12 Cancel Addition'
```

Figure 130 (Part 3 of 3). DDS for display device file MNTMENU

The DDS for the MNTMENU display device file contains three record formats: HDRSCN, CSTINQ, and CSTBLD. The HDRSCN record prompts for the customer number and the mode of processing. The CSTINQ record is used for the Update, Delete, and Display modes. The fields are defined as output/input (B in position 38). The fields are protected when Display or Delete mode is selected (DSPATR(PR)). The CSTBLD record provides only input fields (I in position 38) for a new record.

The HDRSCN record format contains the constant 'Customer File Maintenance'. The ERRMSG keyword defines the messages to be displayed if an error occurs. The CA keywords define the function keys that can be used and associate the function keys with indicators in the RPG program.

CUSMNT: RPG Source

```
*****************************************************************
*  PROGRAM NAME:  CUSMNT                                        *
* RELATED FILES:  CUSMSTL1 (LF)                                 *
*                 MNTMENU  (DSPF)                               *
*   DESCRIPTION:  THIS PROGRAM SHOWS A CUSTOMER MASTER          *
*                 MAINTENANCE PROGRAM USING A WORKSTN FILE.     *
*                 THIS PROGRAM ALLOWS THE USER TO ADD, UPDATE,  *
*                 DELETE AND DISPLAY CUSTOMER RECORDS.          *
*                 PF3 IS USED TO QUIT THE PROGRAM.              *
*****************************************************************

FCUSMSTL1  UF A E           K DISK
FMNTMENU   CF   E             WORKSTN

C     CSTKEY        KLIST
C                   KFLD                    CUST

********************************************************************
*      MAINLINE                                                    *
********************************************************************

C                   MOVE      'DISPLAY '    MODE
C                   EXFMT     HDRSCN
C*
C                   DOW       NOT *IN03
C                   EXSR      SETMOD
C*
C     CUST          IFNE      *ZERO
C     MODE          CASEQ     'ADD'         ADDSUB
C     MODE          CASEQ     'UPDATE'      UPDSUB
C     MODE          CASEQ     'DELETE'      DELSUB
C     MODE          CASEQ     'DISPLAY'     INQSUB
C                   ENDCS
C                   ENDIF
C*
C                   EXFMT     HDRSCN
C                   ENDDO
C                   MOVE      *ON           *INLR
```

Figure 131 (Part 1 of 4). Source for module CUSMNT

```
 *****************************************************************
 *    SUBROUTINE - ADDSUB                                        *
 *    PURPOSE    - ADD NEW CUSTOMER TO FILE                      *
 *****************************************************************

C     ADDSUB        BEGSR
C     CSTKEY        CHAIN     CMLREC1                          50
C                   IF        NOT *IN50
C                   MOVE      *ON           *IN51
C                   ELSE
C                   MOVE      *OFF          *IN51
C                   MOVE      *BLANK        NAME
C                   MOVE      *BLANK        ADDR1
C                   MOVE      *BLANK        ADDR2
C                   MOVE      *BLANK        CITY
C                   MOVE      *BLANK        STATE
C                   Z-ADD     *ZERO         ZIP
C                   EXFMT     CSTBLD
C                   IF        NOT *IN12
C                   WRITE     CMLREC1
C                   ENDIF
C                   ENDIF
C                   ENDSR

 *****************************************************************
 *    SUBROUTINE - UPDSUB                                        *
 *    PURPOSE    - UPDATE CUSTOMER MASTER RECORD                 *
 *****************************************************************

C     UPDSUB        BEGSR
C                   MOVE      *OFF          *IN04
C     CSTKEY        CHAIN     CMLREC1                          52
C                   IF        NOT *IN52
C                   EXFMT     CSTINQ
C                   IF        NOT *IN12
C                   UPDATE    CUSMSTL1
C                   ELSE
C                   UNLOCK    CUSMSTL1
C                   ENDIF
C                   ENDIF
C                   ENDSR
```

Figure 131 (Part 2 of 4). Source for module CUSMNT

```
     *******************************************************************
     *    SUBROUTINE - DELSUB                                          *
     *    PURPOSE    - DELETE CUSTOMER MASTER RECORD                   *
     *******************************************************************

    C     DELSUB        BEGSR
    C                   MOVE      *ON           *IN04
    C     CSTKEY        CHAIN     CMLREC1                          52
    C                   IF        NOT *IN52
    C                   EXFMT     CSTINQ
    C                   IF        NOT *IN12
    C                   DELETE    CMLREC1
    C                   ELSE
    C                   UNLOCK    CUSMSTL1
    C                   ENDIF
    C                   ENDIF
    C                   ENDSR

     *******************************************************************
     *    SUBROUTINE - INQSUB                                          *
     *    PURPOSE    - DISPLAY CUSTOMER MASTER RECORD                  *
     *******************************************************************

    C     INQSUB        BEGSR
    C                   MOVE      *ON           *IN04
    C     CSTKEY        CHAIN     CMLREC1                          52
    C                   IF        NOT *IN52
    C                   EXFMT     CSTINQ
    C                   UNLOCK    CUSMSTL1
    C                   ENDIF
    C                   ENDSR
```

Figure 131 (Part 3 of 4). Source for module CUSMNT

```
      *****************************************************************
      *   SUBROUTINE - SETMOD                                         *
      *   PURPOSE    - SET MAINTENANCE MODE                           *
      *****************************************************************

     C     SETMOD        BEGSR
     C                   SELECT
     C                   WHEN      *IN05
     C                   MOVE      'ADD     '    MODE
     C                   WHEN      *IN06
     C                   MOVE      'UPDATE  '    MODE
     C                   WHEN      *IN07
     C                   MOVE      'DELETE  '    MODE
     C                   WHEN      *IN08
     C                   MOVE      'DISPLAY '    MODE
     C                   ENDSL
     C*
     C                   MOVE      MODE          MODE1
     C                   ENDSR
```

Figure 131 (Part 4 of 4). Source for module CUSMNT

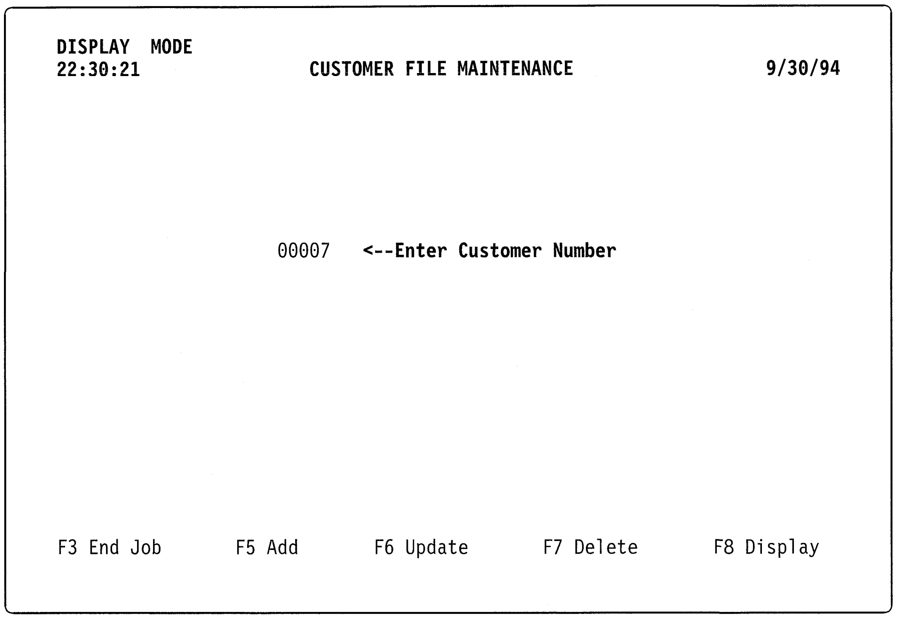
This program maintains a customer master file for additions, changes, and deletions. The program can also be used for inquiry.

The program first sets the default (display) mode of processing and displays the customer maintenance prompt screen. The workstation user can press F3, which turns on indicator 03, to request end of job. Otherwise, to work with customer information, the user enters a customer number and presses Enter. The user can change the mode of processing by pressing F5 (ADD), F6 (UPDATE), F7 (DELETE), or F8 (DISPLAY).

To add a new record to the file, the program uses the customer number as the search argument to chain to the master file. If the record does not exist in the file, the program displays the CSTBLD screen to allow the user to enter a new customer record. If the record is already in the file, an error message is displayed. The user can press F12, which sets on indicator 12, to cancel the add operation and release the record. Otherwise, to proceed with the add operation, the user enters information for the new customer record in the input fields and writes the new record to the master file.

To update, delete, or display an existing record, the program uses the customer number as the search argument to chain to the master file. If a record for that customer exists in the file, the program displays the customer file inquiry screen CSTINQ. If the record is not in the file, an error message is displayed. If the mode of processing is display or delete, the input fields are protected from modification. Otherwise, to proceed with the customer record, the user can enter new information in the customer record input fields. The user can press F12, which sets on indicator 12, to cancel the update or delete operation, and release the record. Display mode automatically releases the record when Enter is pressed.

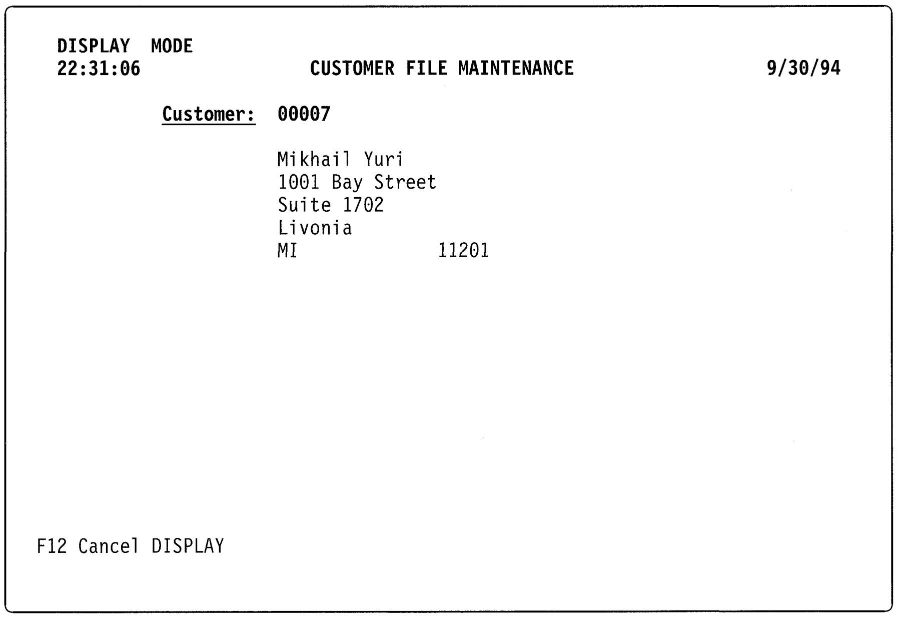
In Figure 132 on page 281, the workstation user responds to the prompt by entering customer number 00007 to display the customer record.



```
DISPLAY  MODE
22:30:21            CUSTOMER FILE MAINTENANCE              9/30/94




                    00007   <--Enter Customer Number







F3 End Job     F5 Add      F6 Update     F7 Delete     F8 Display
```

Figure 132. 'Customer File Maintenance' Display Mode prompt screen

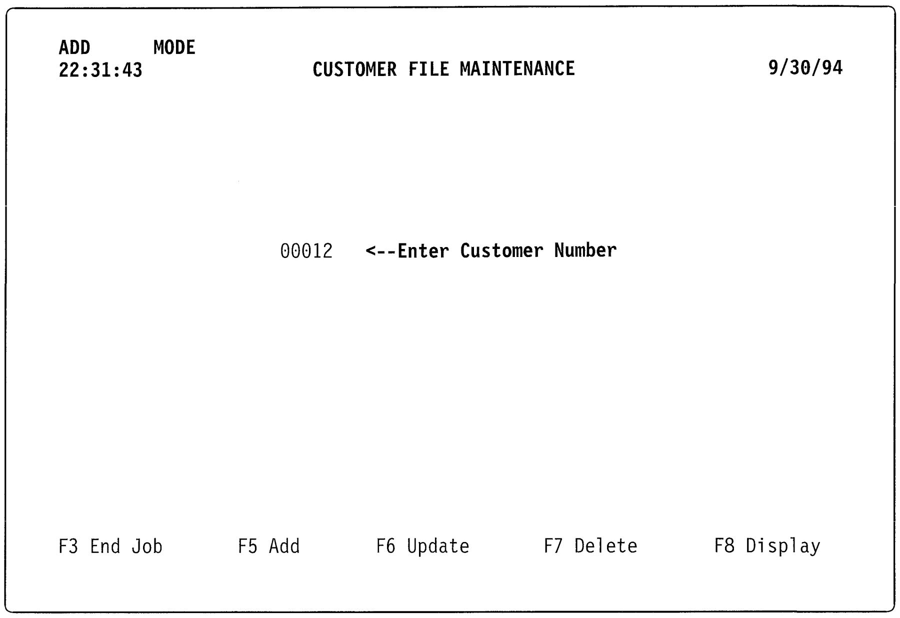
Because the customer record for customer number 00007 exists in the Master File, the data is displayed as show in Figure 133.



```
DISPLAY  MODE
22:31:06            CUSTOMER FILE MAINTENANCE              9/30/94

          Customer:  00007

                     Mikhail Yuri
                     1001 Bay Street
                     Suite 1702
                     Livonia
                     MI           11201








 F12 Cancel DISPLAY
```

Figure 133. 'Customer File Maintenance' Display Mode screen

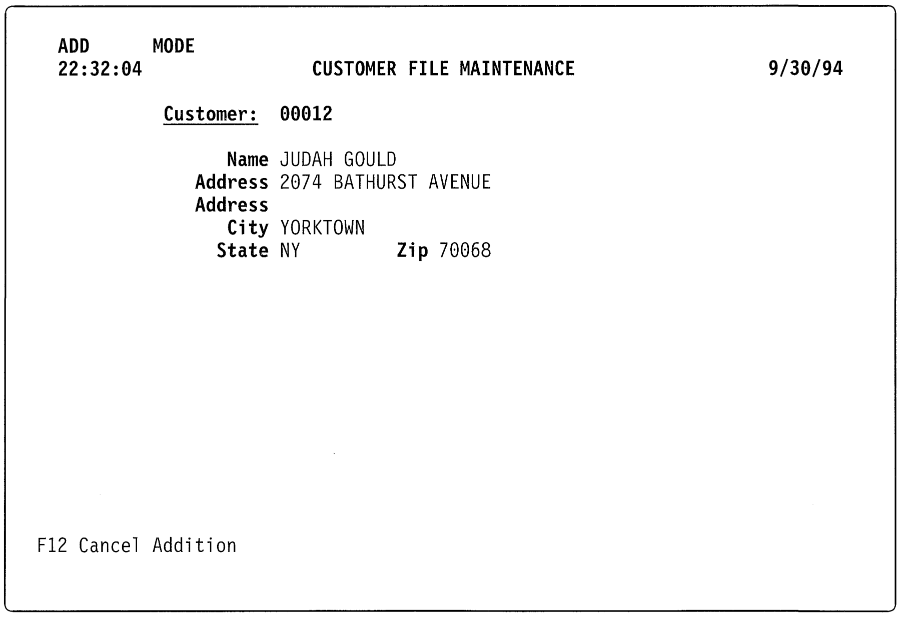
The workstation user responds to the add prompt by entering a new customer number as shown in Figure 134 on page 282.



```
ADD      MODE
22:31:43                CUSTOMER FILE MAINTENANCE                  9/30/94



                      00012   <--Enter Customer Number




F3 End Job      F5 Add      F6 Update      F7 Delete      F8 Display
```

Figure 134. 'Customer File Maintenance' Add Mode prompt screen

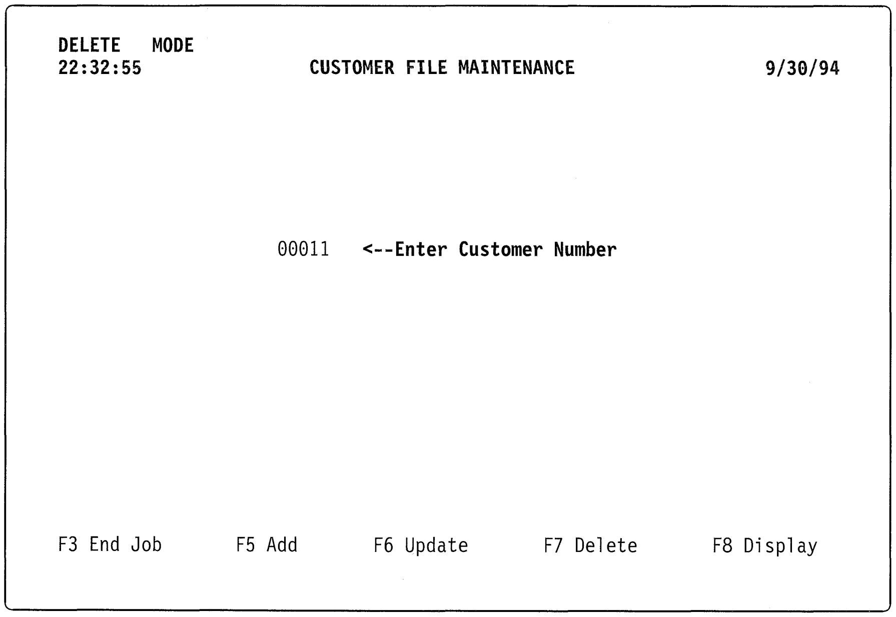
In Figure 135 a new customer is added to the Customer Master File.



```
ADD      MODE
22:32:04                CUSTOMER FILE MAINTENANCE                  9/30/94

          Customer:  00012

               Name JUDAH GOULD
            Address 2074 BATHURST AVENUE
            Address
               City YORKTOWN
              State NY         Zip 70068




F12 Cancel Addition
```

Figure 135. 'Customer File Maintenance' Add Mode prompt screen

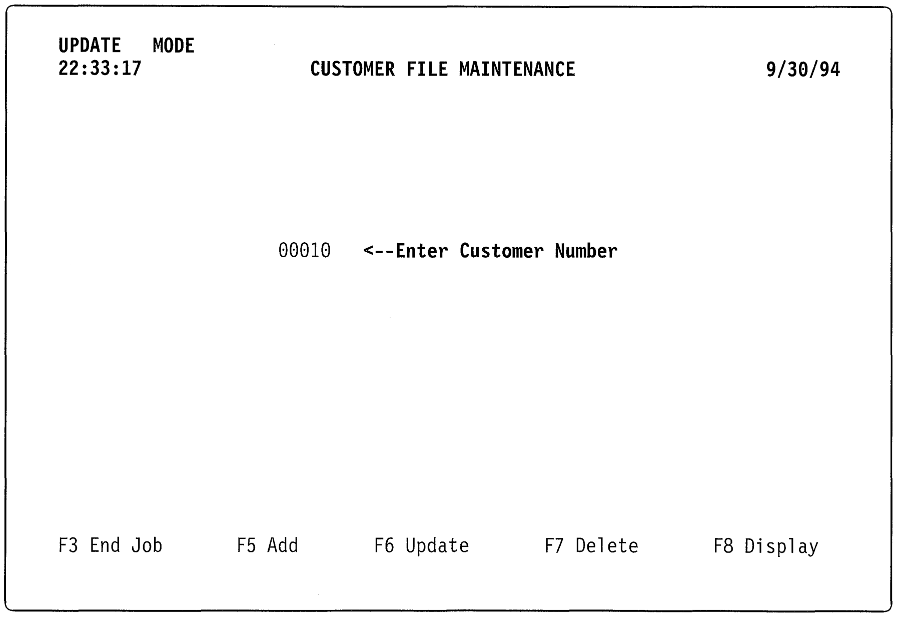
The workstation user responds to the delete prompt by entering a customer number as shown in Figure 136 on page 283.



```
DELETE   MODE
22:32:55             CUSTOMER FILE MAINTENANCE               9/30/94



                   00011   <--Enter Customer Number





F3 End Job     F5 Add      F6 Update     F7 Delete     F8 Display
```

Figure 136. 'Customer File Maintenance' Delete Mode prompt screen

The workstation user responds to the update prompt by entering a customer number as shown in Figure 137.



```
UPDATE   MODE
22:33:17             CUSTOMER FILE MAINTENANCE               9/30/94



                   00010   <--Enter Customer Number





F3 End Job     F5 Add      F6 Update     F7 Delete     F8 Display
```

Figure 137. 'Customer File Maintenance' Update Mode prompt screen

Search by Zip Code

The following illustrates WORKSTN subfile processing (display only). Subfiles are used to display all matched records for a specified zip code.

CUSMSTL2: DDS for a Logical File

```
A*****************************************************************
A*     FILE NAME:  CUSMSTL2                                      *
A*  RELATED PGMS:  SCHZIP                                        *
A* RELATED FILES:  CUSMST   (PHYSICAL FILE)                      *
A*   DESCRIPTION:  THIS IS LOGICAL FILE CUSMSTL2.                *
A*                 IT CONTAINS ONE RECORD FORMAT CALLED CMLREC2. *
A*                 LOGICAL VIEW OF CUSTOMER MASTER FILE (CUSMST) *
A*                 BY CUSTOMER ZIP CODE (ZIP)                    *
A*****************************************************************
A          R CMLREC2                   PFILE(CUSMST)
A            ZIP
A            NAME
A            ARBAL
A          K ZIP
```

Figure 138. DDS for logical file CUSMSTL2

The DDS for the database file used by this program describe one record format: CMLREC2. The logical file CUSMSTL2 keyed by zip code is based on the physical file CUSMST, as indicated by the PFILE keyword. The record format created by the logical file will include only those fields specified in the logical file DDS. All other fields will be excluded.

SZIPMENU: DDS for a Display Device File

```
A*****************************************************************
A*     FILE NAME:  SZIPMENU                                      *
A*  RELATED PGMS:  SCHZIP                                        *
A* RELATED FILES:  CUSMSTL2   (LOGICAL FILE)                     *
A*   DESCRIPTION:  THIS IS THE DISPLAY FILE SZIPMENU. IT HAS 6   *
A*                 RECORD FORMATS.                               *
A*****************************************************************
A                                      REF(CUSMSTL2)
A                                      CHGINPDFT(CS)
A                                      PRINT(QSYSPRT)
A                                      INDARA
A                                      CA03(03 'END OF JOB')
```

Figure 139 (Part 1 of 3). DDS for display device file SZIPMENU

```
A          R HEAD
A                                      OVERLAY
A                                  2  4TIME
A                                      DSPATR(HI)
A                                  2 28'CUSTOMER SEARCH BY ZIP'
A                                      DSPATR(HI RI)
A                                  2 70DATE
A                                      EDTCDE(Y)
A                                      DSPATR(HI)
A          R FOOT1
A                                 23  6'ENTER - Continue'
A                                      DSPATR(HI)
A                                 23 29'F3 - End Job'
A                                      DSPATR(HI)
A          R FOOT2
A                                 23  6'ENTER - Continue'
A                                      DSPATR(HI)
A                                 23 29'F3 - End Job'
A                                      DSPATR(HI)
A                                 23 47'F4 - RESTART ZIP CODE'
A                                      DSPATR(HI)
A          R PROMPT
A                                      OVERLAY
A                                  4  4'Enter Zip Code'
A                                      DSPATR(HI)
A            ZIP       R     Y  I  4 19DSPATR(CS)
A                                      CHECK(RZ)
A  61                                  ERRMSG('ZIP CODE NOT FOUND' +
A                                      61)
A          R SUBFILE                   SFL
A            NAME      R           9  4
A            ARBAL     R           9 27EDTCDE(J)
A          R SUBCTL                    SFLCTL(SUBFILE)
A  55                                  SFLCLR
A N55                                  SFLDSPCTL
A N55                                  SFLDSP
A                                      SFLSIZ(13)
A                                      SFLPAG(13)
A                                      ROLLUP(95 'ROLL UP')
A                                      OVERLAY
A                                      CA04(04 'RESTART ZIP CDE')
A                                  4  4'Zip Code'
```

Figure 139 (Part 2 of 3). DDS for display device file SZIPMENU

```
A            ZIP       R          0  4 14DSPATR(HI)
A                                    7  4'Customer Name'
A                                         DSPATR(HI UL)
A                                    7 27'A/R Balance'
A                                         DSPATR(HI UL)
```

Figure 139 (Part 3 of 3). DDS for display device file SZIPMENU

The DDS for the SZIPMENU display device file contains six record formats: HEAD, FOOT1, FOOT2, PROMPT, SUBFILE, and SUBCTL.

The PROMPT record format requests the user to enter a zip code. If the zip code is not found in the file, an error message is displayed. The user can press F3, which sets on indicator 03, to end the program.

The SUBFILE record format must be defined immediately preceding the subfile-control record format SUBCTL. The subfile record format, which is defined with the keyword SFL, describes each field in the record, and specifies the location where the first record is to appear on the display (here, on line 9).

The subfile-control record format contains the following unique keywords:

- SFLCTL identifies this format as the control record format and names the associated subfile record format.
- SFLCLR describes when the subfile is to be cleared of existing records (when indicator 55 is on). This keyword is needed for additional displays.
- SFLDSPCTL indicates when to display the subfile-control record format (when indicator 55 is off).
- SFLDSP indicates when to display the subfile (when indicator 55 is off).
- SFLSIZ specifies the total size of the subfile. In this example, the subfile size is 13 records that are displayed on lines 9 through 21.
- SFLPAG defines the number of records on a page. In this example, the page size is the same as the subfile size.
- ROLLUP indicates that indicator 95 is set on in the program when the roll up function is used.

The OVERLAY keyword defines this subfile-control record format as an overlay format. This record format can be written without the OS/400 system erasing the screen first. F4 is valid for repeating the search with the same zip code. (This use of F4 allows a form of roll down.)

SCHZIP: RPG Source

```
*****************************************************************
*  PROGRAM NAME:  SCHZIP                                        *
* RELATED FILES:  CUSMSTL2 (LOGICAL FILE)                       *
*                 SZIPMENU (WORKSTN FILE)                       *
*   DESCRIPTION:  THIS PROGRAM SHOWS A CUSTOMER MASTER SEARCH   *
*                 PROGRAM USING WORKSTN SUBFILE PROCESSING.     *
*                 THIS PROGRAM PROMPTS THE USER FOR THE ZIP CODE*
*                 AND DISPLAYS THE CUSTOMER MASTER RECORD BY    *
*                 ZIP CODE.                                     *
*                 ROLL UP KEY CAN BE USED TO LOOK AT ANOTHER    *
*                 PAGE. PF3 IS USED TO QUIT THE PROGRAM.        *
*****************************************************************

FCUSMSTL2  IF   E           K DISK
FSZIPMENU  CF   E             WORKSTN SFILE(SUBFILE:RECNUM)

C     CSTKEY        KLIST
C                   KFLD                    ZIP

 ********************************************************************
 *    MAINLINE                                                      *
 ********************************************************************

C                   WRITE     FOOT1
C                   WRITE     HEAD
C                   EXFMT     PROMPT
C*
C                   DOW       NOT *IN03
C                   MOVE      *OFF          *IN61
C     CSTKEY        SETLL     CMLREC2                                20
C                   IF        NOT *IN20
C                   MOVE      *ON           *IN61
C                   ELSE
C                   EXSR      SFLPRC
C                   END
C                   IF        (NOT *IN03) AND (NOT *IN04)
C                   IF        NOT *IN61
C                   WRITE     FOOT1
C                   WRITE     HEAD
C                   ENDIF
C                   EXFMT     PROMPT
C                   ENDIF
C                   ENDDO
C*
C                   SETON                                        LR
```

Figure 140 (Part 1 of 3). Source for module SCHZIP

```
 *****************************************************************
 *    SUBROUTINE - SFLPRC                                         *
 *    PURPOSE    - PROCESS SUBFILE AND DISPLAY                    *
 *****************************************************************

C     SFLPRC        BEGSR
C     NXTPAG        TAG
C                   EXSR      SFLCLR
C                   EXSR      SFLFIL
C     SAMPAG        TAG
C                   WRITE     FOOT2
C                   WRITE     HEAD
C                   EXFMT     SUBCTL
C                   IF        *IN95
C                   IF        NOT *IN71
C                   GOTO      NXTPAG
C                   ELSE
C                   GOTO      SAMPAG
C                   ENDIF
C                   ENDIF
C                   ENDSR

 *****************************************************************
 *    SUBROUTINE - SFLFIL                                         *
 *    PURPOSE    - FILL SUBFILE                                   *
 *****************************************************************

C     SFLFIL        BEGSR
C                   DOW       NOT *IN21
C     ZIP           READE     CMLREC2                              71
C                   IF        *IN71
C                   MOVE      *ON           *IN21
C                   ELSE
C                   ADD       1             RECNUM
C                   WRITE     SUBFILE                              21
C                   ENDIF
C                   ENDDO
C                   ENDSR
```

Figure 140 (Part 2 of 3). Source for module SCHZIP

```
 *****************************************************************
 *    SUBROUTINE - SFLCLR                                        *
 *    PURPOSE    - CLEAR SUBFILE RECORDS                         *
 *****************************************************************

C     SFLCLR        BEGSR
C                   MOVE      *ON           *IN55
C                   WRITE     SUBCTL
C                   MOVE      *OFF          *IN55
C                   MOVE      *OFF          *IN21
C                   Z-ADD     *ZERO         RECNUM            5 0
C                   ENDSR
```

Figure 140 (Part 3 of 3). Source for module SCHZIP

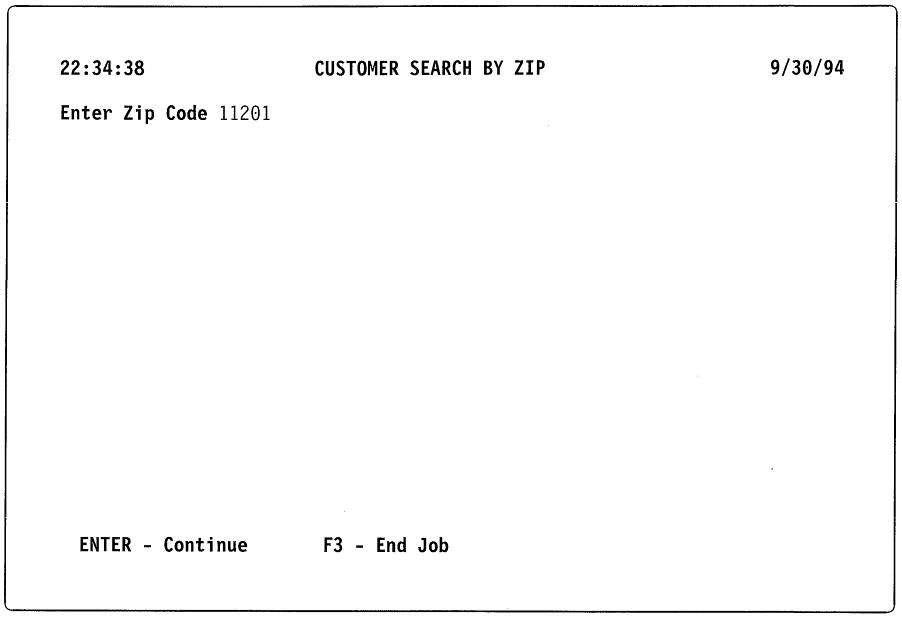
The file description specifications identify the disk file to be searched and the display device file to be used (SZIPMENU). The SFILE keyword for the WORKSTN file identifies the record format (SUBFILE) that is to be used as a subfile. The relative-record-number field (RECNUM) specified controls which record within the subfile is being accessed.

The program displays the PROMPT record format and waits for the workstation user's response. F3 sets on indicator 03, which controls the end of the program. The zip code (ZIP) is used to position the CUSMSTL2 file by the SETLL operation. Notice that the record format name CMLREC2 is used in the SETLL operation instead of the file name CUSMSTL2. If no record is found, an error message is displayed.

The SFLPRC subroutine handles the processing for the subfile: clearing, filling, and displaying. The subfile is prepared for additional requests in subroutine SFLCLR. If indicator 55 is on, no action occurs on the display, but the main storage area for the subfile records is cleared. The SFLFIL routine fills the subfile with records. A record is read from the CUSMSTL2 file. If the zip code is the same, the record count (RECNUM) is incremented and the record is written to the subfile. This subroutine is repeated until either the subfile is full (indicator 21 on the WRITE operation) or end of file occurs on the CUSMSTL2 file (indicator 71 on the READE operation). When the subfile is full or end of file occurs, the subfile is written to the display by the EXFMT operation by the subfile-control record control format. The user reviews the display and decides whether:

- To end the program by pressing F3.
- To restart the zip code by pressing F4. The PROMPT record format is not displayed, and the subfile is displayed starting over with the same zip code.
- To fill another page by pressing ROLL UP. If end of file has occurred on the CUSMSTL2 file, the current page is re-displayed; otherwise, the subfile is cleared and the next page is displayed.
- To continue with another zip code by pressing ENTER. The PROMPT record format is displayed. The user can enter a zip code or end the program.

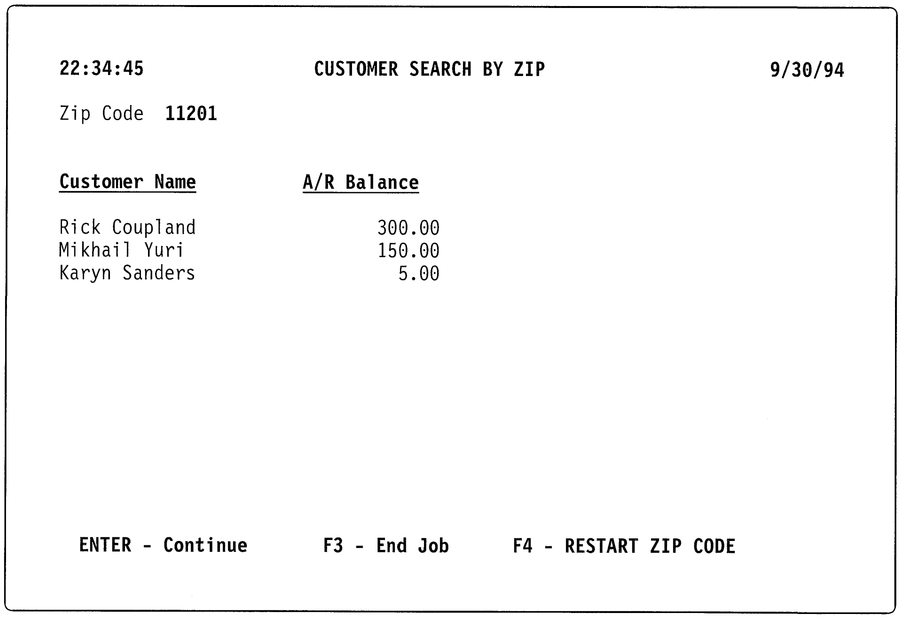
In Figure 141 on page 290, the user enters a zip code in response to the prompt.



```
22:34:38                 CUSTOMER SEARCH BY ZIP                  9/30/94

Enter Zip Code 11201



  ENTER - Continue        F3 - End Job
```

Figure 141. 'Customer Search by Zip' prompt screen

The subfile is written to the screen as shown in Figure 142.



```
22:34:45                 CUSTOMER SEARCH BY ZIP                  9/30/94

Zip Code  11201

Customer Name          A/R Balance

Rick Coupland               300.00
Mikhail Yuri                150.00
Karyn Sanders                 5.00



  ENTER - Continue        F3 - End Job       F4 - RESTART ZIP CODE
```

Figure 142. 'Customer Search by Zip' screen

Search and Inquiry by Name

The following illustrates WORKSTN subfile processing (display with selection). Subfiles are used to display all matched records for a specified customer name, and then the user is allowed to make a selection from the subfile, such that additional information about the customer can be displayed.

CUSMSTL3: DDS for a Logical File

```
A*****************************************************************
A*     FILE NAME:  CUSMSTL3                                      *
A*  RELATED PGMS:  SCHNAM                                        *
A* RELATED FILES:  CUSMST                                        *
A*   DESCRIPTION:  THIS IS THE LOGICAL FILE CUSMSTL3. IT HAS     *
A*                 ONE RECORD FORMAT CALLED CUSREC.              *
A*                 LOGICAL VIEW OF CUSTOMER MASTER FILE (CUSMST) *
A*                 BY NAME (NAME)                                *
A*****************************************************************
A          R CUSREC                    PFILE(CUSMST)
A          K NAME
A*
A*****************************************************************
A* NOTE: SINCE THE RECORD FORMAT OF THE PHYSICAL FILE (CUSMST)   *
A*       HAS THE SAME RECORD-FORMAT-NAME, NO LISTING OF FIELDS   *
A*       IS REQUIRED IN THIS DDS FILE.                           *
A*****************************************************************
```

Figure 143. DDS for logical file CUSMSTL3

The DDS for the database file used in this program defines one record format named CUSREC and identifies the NAME field as the key fields.

SNAMMENU: DDS for a Display Device File

```
A*****************************************************************
A*     FILE NAME:  SNAMMENU                                      *
A*  RELATED PGMS:  SCHNAM                                        *
A* RELATED FILES:  CUSMSTL3   (LOGICAL FILE)                     *
A*   DESCRIPTION:  THIS IS THE DISPLAY FILE SNAMMENU. IT HAS 7   *
A*                 RECORD FORMATS.                               *
A*****************************************************************
A                                      REF(CUSMSTL3)
A                                      CHGINPDFT(CS)
A                                      PRINT(QSYSPRT)
A                                      INDARA
A                                      CA03(03 'END OF JOB')
```

Figure 144 (Part 1 of 3). DDS for display device file SNAMMENU

```
     A          R HEAD
     A                                            OVERLAY
     A                                        2  4TIME
     A                                            DSPATR(HI)
     A                                        2 25'CUSTOMER SEARCH & INQUIRY BY NAME'
     A                                            DSPATR(HI UL)
     A                                        2 70DATE
     A                                            EDTCDE(Y)
     A                                            DSPATR(HI)
     A          R FOOT1
     A                                       23  6'ENTER - Continue'
     A                                            DSPATR(HI)
     A                                       23 29'F3 - End Job'
     A                                            DSPATR(HI)
     A          R FOOT2
     A                                       23  6'ENTER - Continue'
     A                                            DSPATR(HI)
     A                                       23 29'F3 - End Job'
     A                                            DSPATR(HI)
     A                                       23 47'F4 - Restart Name'
     A                                            DSPATR(HI)
     A          R PROMPT
     A                                            OVERLAY
     A                                        5  4'Enter Search Name'
     A                                            DSPATR(HI)
     A            SRCNAM    R        I  5 23REFFLD(NAME CUSMSTL3)
     A                                            DSPATR(CS)
     A          R SUBFILE                         SFL
     A                                            CHANGE(99 'FIELD CHANGED')
     A            SEL           1A  B  9  8DSPATR(CS)
     A                                            VALUES(' ' 'X')
     A            ZIP       R        O  9 54
     A            CUST      R        O  9 43
     A            NAME      R        O  9 17
     A          R SUBCTL                          SFLCTL(SUBFILE)
     A                                            SFLSIZ(0013)
     A                                            SFLPAG(0013)
     A  55                                        SFLCLR
     A N55                                        SFLDSPCTL
     A N55                                        SFLDSP
     A                                            ROLLUP(95 'ROLL UP')
     A                                            OVERLAY
     A                                            CA04(04 'RESTART SEARCH NAME')
     A                                        5  4'Search Name'
```

Figure 144 (Part 2 of 3). DDS for display device file SNAMMENU

```
     A          SRCNAM    R        O  5 17REFFLD(NAME CUSMSTL3)
     A                                     DSPATR(HI)
     A                                  7  6'Select'
     A                                     DSPATR(HI)
     A                                  8  6' "X"       Customer Name '
     A                                     DSPATR(HI)
     A                                     DSPATR(UL)
     A                                  8 42' Number     Zip Code   '
     A                                     DSPATR(HI)
     A                                     DSPATR(UL)
     A        R CUSDSP
     A                                     OVERLAY
     A                                     CA04(04 'RESTART ZIP CDE')
     A                                  6 25'Customer'
     A          CUST           5S 00  6 35DSPATR(HI)
     A                                  8 25'Name'
     A          NAME          20A  O  8 35DSPATR(HI)
     A                                 10 25'Address'
     A          ADDR1         20A  O 10 35DSPATR(HI)
     A          ADDR2         20A  O 11 35DSPATR(HI)
     A                                 13 25'City'
     A          CITY          20A  O 13 35DSPATR(HI)
     A                                 15 25'State'
     A          STATE          2A  O 15 35DSPATR(HI)
     A                                 15 41'Zip Code'
     A          ZIP            5S 00 15 50DSPATR(HI)
     A                                 17 25'A/R Balance'
     A          ARBAL         10Y 20 17 42DSPATR(HI)
     A                                     EDTCDE(J)
```

Figure 144 (Part 3 of 3). DDS for display device file SNAMMENU

The DDS for the SNAMMENU display device file contains seven record formats: HEAD, FOOT1, FOOT2, PROMPT, SUBFILE, SUBCTL, and CUSDSP.

The PROMPT record format requests the user to enter a zip code and search name. If no entry is made, the display starts at the beginning of the file. The user can press F3, which sets on indicator 03, to end the program.

The SUBFILE record format must be defined immediately preceding the subfile-control record format SUBCTL. The subfile-record format defined with the keyword SFL, describes each field in the record, and specifies the location where the first record is to appear on the display (here, on line 9).

The subfile-control record format SUBCTL contains the following unique keywords:

- SFLCTL identifies this format as the control record format and names the associated subfile record format.
- SFLCLR describes when the subfile is to be cleared of existing records (when indicator 55 is on). This keyword is needed for additional displays.
- SFLDSPCTL indicates when to display the subfile-control record format (when indicator 55 is off).
- SFLDSP indicates when to display the subfile (when indicator 55 is off).
- SFLSIZ specifies the total size of the subfile. In this example, the subfile size is 13 records that are displayed on lines 9 through 21.

- SFLPAG defines the number of records on a page. In this example, the page size is the same as the subfile size.
- ROLLUP indicates that indicator 95 is set on in the program when the roll up function is used.

The OVERLAY keyword defines this subfile-control record format as an overlay format. This record format can be written without the OS/400 system erasing the screen first. F4 is valid for repeating the search with the same name. (This use of F4 allows a form of roll down.)

The CUSDSP record format displays information for the selected customers.

SCHNAM: RPG Source

```
      *****************************************************************
      *  PROGRAM NAME:  SCHNAM                                        *
      * RELATED FILES:  CUSMSTL3 (LOGICAL FILE)                       *
      *                 SNAMMENU (WORKSTN FILE)                       *
      *   DESCRIPTION:  THIS PROGRAM SHOWS A CUSTOMER MASTER SEARCH   *
      *                 PROGRAM USING WORKSTN SUBFILE PROCESSING.     *
      *                 THIS PROGRAM PROMPTS THE USER FOR THE CUSTOMER*
      *                 NAME AND USES IT TO POSITION THE CUSMSTL3     *
      *                 FILE BY THE SETLL OPERATION. THEN IT DISPLAYS *
      *                 THE RECORDS USING SUBFILES.                   *
      *                 TO FILL ANOTHER PAGE, PRESS THE ROLL UP KEY.  *
      *                 TO DISPLAY CUSTOMER DETAIL, ENTER 'X' BESIDE  *
      *                 THAT CUSTOMER AND PRESS ENTER.                *
      *                 TO QUIT THE PROGRAM, PRESS PF3.               *
      *****************************************************************

     FCUSMSTL3  IF   E           K DISK
     FSNAMMENU  CF   E             WORKSTN SFILE(SUBFILE:RECNUM)

     C     CSTKEY        KLIST
     C                   KFLD                    SRCNAM
     C     ZIPKEY        KLIST
     C                   KFLD                    NAME
```

Figure 145 (Part 1 of 3). Source for module SCHNAM

```
 *******************************************************************
 *    MAINLINE                                                     *
 *******************************************************************

C                   WRITE     FOOT1
C                   WRITE     HEAD
C                   EXFMT     PROMPT
C                   DOW       NOT *IN03
C     CSTKEY        SETLL     CUSREC
C                   EXSR      SFLPRC
C                   EXSR      SFLCHG
C                   IF        (NOT *IN03) AND (NOT *IN04)
C                   WRITE     FOOT1
C                   WRITE     HEAD
C                   EXFMT     PROMPT
C                   ENDIF
C                   ENDDO
C*
C                   SETON                                        LR

 *******************************************************************
 *    SUBROUTINE - SFLPRC                                          *
 *    PURPOSE    - PROCESS SUBFILE AND DISPLAY                     *
 *******************************************************************

C     SFLPRC        BEGSR
C     NXTPAG        TAG
C                   EXSR      SFLCLR
C                   EXSR      SFLFIL
C     SAMPAG        TAG
C                   WRITE     FOOT2
C                   WRITE     HEAD
C                   EXFMT     SUBCTL
C                   IF        *IN95
C                   IF        NOT *IN71
C                   GOTO      NXTPAG
C                   ELSE
C                   GOTO      SAMPAG
C                   ENDIF
C                   ENDIF
C                   ENDSR
```

Figure 145 (Part 2 of 3). Source for module SCHNAM

```
 *****************************************************************
 *    SUBROUTINE - SFLFIL                                        *
 *    PURPOSE    - FILL SUBFILE                                  *
 *****************************************************************

C     SFLFIL        BEGSR
C                   DOW       NOT *IN21
C                   READ      CUSREC                            71
C                   IF        *IN71
C                   MOVE      *ON           *IN21
C                   ELSE
C                   ADD       1             RECNUM
C                   MOVE      *BLANK        SEL
C                   WRITE     SUBFILE                           21
C                   ENDIF
C                   ENDDO
C                   ENDSR

 *****************************************************************
 *    SUBROUTINE - SFLCLR                                        *
 *    PURPOSE    - CLEAR SUBFILE RECORDS                         *
 *****************************************************************

C     SFLCLR        BEGSR
C                   MOVE      *ON           *IN55
C                   WRITE     SUBCTL
C                   MOVE      *OFF          *IN55
C                   MOVE      *OFF          *IN21
C                   Z-ADD     *ZERO         RECNUM          5 0
C                   ENDSR

 *****************************************************************
 *    SUBROUTINE - SFLCHG                                        *
 *    PURPOSE    - CUSTOMER RECORD SELECTED                      *
 *****************************************************************

C     SFLCHG        BEGSR
C                   READC     SUBFILE                           98
C                   IF        NOT *IN98
C     ZIPKEY        CHAIN     CUSREC                        71
C                   EXFMT     CUSDSP
C                   ENDIF
C                   ENDSR
```

Figure 145 (Part 3 of 3). Source for module SCHNAM

The file description specifications identify the disk file to be searched and the display device file to be used (SNAMMENU). The SFILE keyword for the WORKSTN file identifies the record format (SUBFILE) to be used as a subfile. The relative-record-number field (RECNUM) specifies which record within the subfile is being accessed.

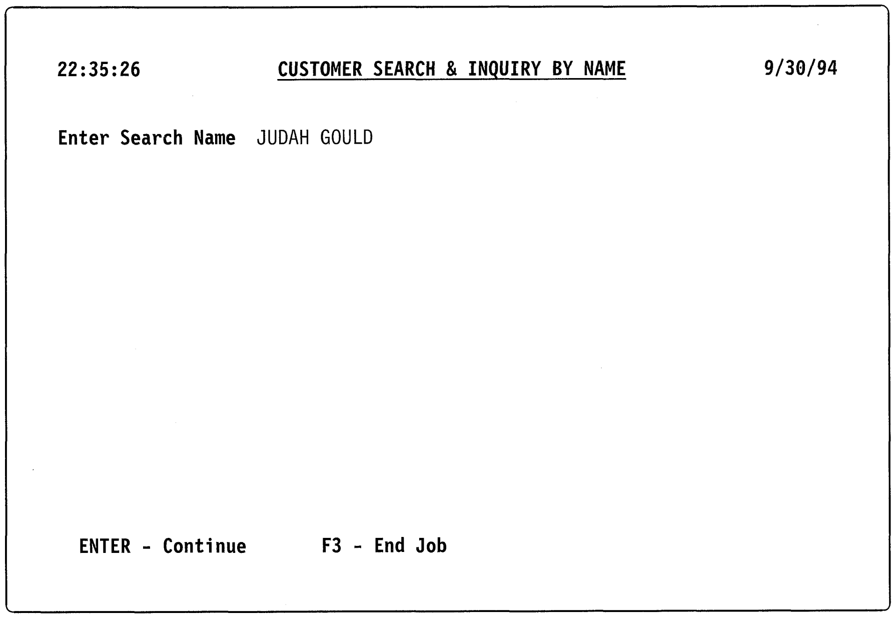
The program displays the PROMPT record format and waits for the workstation user's response. F3 sets on indicator 03, which controls the end of the program. The name (NAME) is used as the key to position the CUSMSTL3 file by the SETLL operation. Notice that the record format name CUSREC is used in the SETLL operation instead of the file name CUSMSTL3.

The SFLPRC subroutine handles the processing for the subfile: clearing, filling, and displaying. The subfile is prepared for additional requests in subroutine

SFLCLR. If indicator 55 is on, no action occurs on the display, but the main storage area for the subfile records is cleared. The SFLFIL routine fills the subfile with records. A record is read from the CUSMSTL3 file, the record count (RECNUM) is incremented, and the record is written to the subfile. This subroutine is repeated until either the subfile is full (indicator 21 on the WRITE operation) or end of file occurs on the CUSMSTL3 file (indicator 71 on the READ operation). When the subfile is full or end of file occurs, the subfile is written to the display by the EXFMT operation by the subfile-control record control format. The user reviews the display and decides:

- To end the program by pressing F3.
- To restart the subfile by pressing F4. The PROMPT record format is not displayed, and the subfile is displayed starting over with the same name.
- To fill another page by pressing the ROLL UP keys. If end of file has occurred on the CUSMSTL3 file, the current page is displayed again; otherwise, the subfile is cleared, and the next page is displayed.
- To display customer detail by entering X, and pressing ENTER. The user can then return to the PROMPT screen by pressing ENTER, display the subfile again by pressing F4, or end the program by pressing F3.

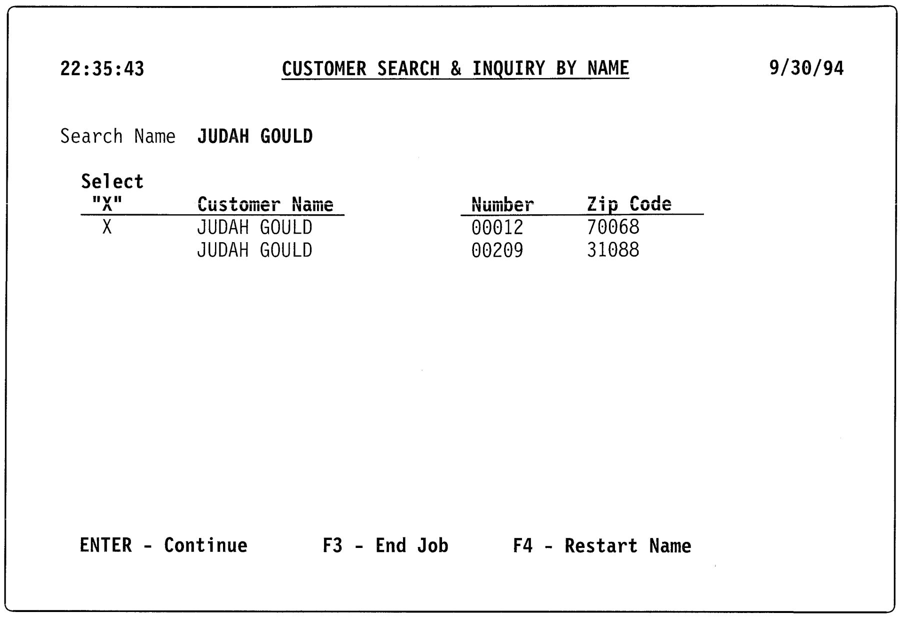
In Figure 146, the user responds to the initial prompt by entering a customer name.



```
22:35:26          CUSTOMER SEARCH & INQUIRY BY NAME          9/30/94

Enter Search Name  JUDAH GOULD













  ENTER - Continue       F3 - End Job
```

Figure 146. 'Customer Search and Inquiry by Name' prompt screen

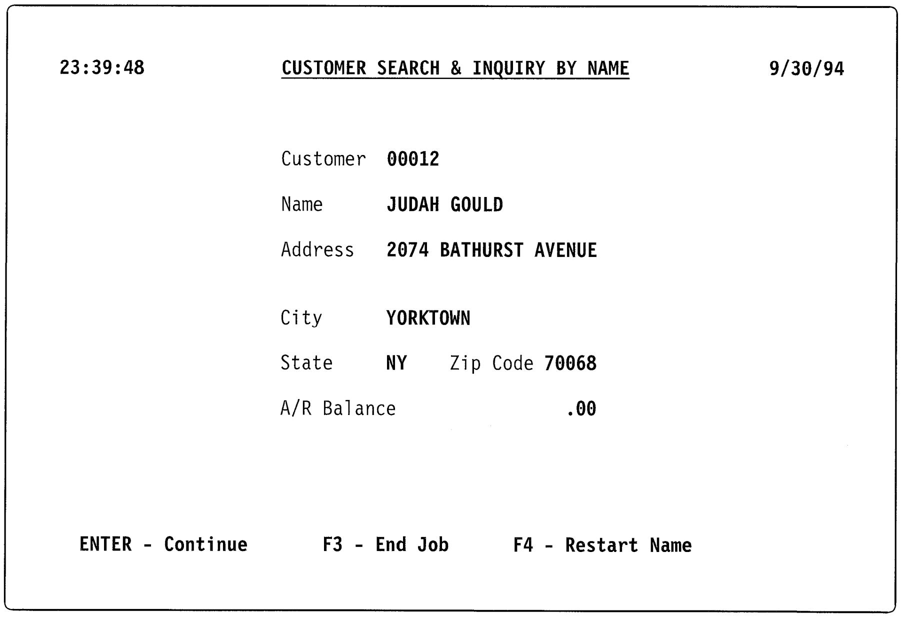
The user requests more information by entering an X as shown in Figure 147 on page 298.



```
 22:35:43             CUSTOMER SEARCH & INQUIRY BY NAME              9/30/94

 Search Name  JUDAH GOULD

   Select
    "X"       Customer Name              Number     Zip Code
     X        JUDAH GOULD                00012      70068
              JUDAH GOULD                00209      31088




   ENTER - Continue       F3 - End Job      F4 - Restart Name
```

Figure 147. 'Customer Search and Inquiry by Name' information screen

The detailed information for the customer selected is shown in Figure 148. At this point the user selects the appropriate function key to continue or end the inquiry.



```
 23:39:48             CUSTOMER SEARCH & INQUIRY BY NAME              9/30/94


                      Customer  00012

                      Name      JUDAH GOULD

                      Address   2074 BATHURST AVENUE


                      City      YORKTOWN

                      State     NY    Zip Code 70068

                      A/R Balance              .00


   ENTER - Continue       F3 - End Job      F4 - Restart Name
```

Figure 148. 'Customer Search and Inquiry by Name' detailed information screen

Appendixes

Appendix A. Behavioral Differences Between OPM RPG/400 and ILE RPG/400

The following lists note differences in the behavior of the OPM RPG/400 compiler and ILE RPG/400.

For information on the differences in the language definitions, see the appropriate appendix in *ILE RPG/400 Reference*.

Compiling

1. In RPG IV there is no dependency between DATEDIT and DECEDIT in the Control specification.
2. Regarding the ILE RPG/400 create commands (CRTBNDRPG and CRTRPGMOD):
   - Specifying CVTOPT(*NONE) in ILE RPG/400 means that all externally-described fields which are of a type or with attributes not supported by RPG (variable-length character and variable-length graphic) will be ignored. This also means that fields which are of type graphic, date, time, and timestamp are brought into the program with the same type as specified in the external-description.
   - The IGNDECERR parameter on the CRTRPGPGM command has been replaced by the FIXNBR parameter on the ILE RPG/400 create commands.
   - There is a new parameter TRUNCNBR for controlling whether numeric overflow is allowed.
   - There is no auto report feature or commands in RPG IV.
   - You cannot request an MI listing from the compiler.
3. In a compiler listing, line numbers start at 1 and increment by 1 for each line of source or generated specifications. Source ids are numeric, that is, there are no more AA000100 line numbers for /COPY members or expanded DDS.
4. RPG IV requires that all compiler directives appear *before* compile-time data, including /TITLE. When RPG IV encounters a /TITLE directive, it will treat it as data. (RPG III treats /TITLE specifications as compiler directives anywhere in the source.)

   The Conversion Aid will remove any /TITLE specifications it encounters in compile-time data.
5. ILE RPG/400 is more rigorous in detecting field overlap in data structures. In some cases, ILE RPG/400 will issue a message when the OPM compiler would not have.

Running

1. The FREE operation is not supported by RPG IV.
2. Certain MCH messages may appear in the job log which do not under OPM, for example, MCH1202. The appearance of these messages does not indicate a change in the behavior of the program.

3. If you use the non-bindable API QMHSNDPM to send messages from your program, you may need to add 1 to the stack offset parameter to allow for the presence of the program-entry procedure in the stack. This will only be the case if the ILE procedure is the user-entry procedure, and you were using the special value of '*' for the call message queue, and a value of greater than 0 for the stack offset.

4. ILE RPG/400 does not interpret return codes which are not 0 or 1 for non-RPG programs or procedures which end without an exception.

5. When recursion is detected, OPM RPG/400 displays inquiry message RPG8888. ILE RPG signals escape message RNX8888; no inquiry message is displayed for this condition.

Debugging and Exception Handling

1. The DEBUG operation is not supported in RPG IV.

2. You cannot use RPG tags, subroutine names, or points in the cycle such as *GETIN and *DETC, for setting breakpoints when using the ILE source debugger.

3. Function checks are normally written to the job log by both OPM RPG/400 and ILE RPG/400. However, in ILE RPG/400, if you have an error indicator or *PSSR error routine coded, then the function check will not appear.

   You should remove any code that deletes function checks, since the presence of the *PSSR will prevent function checks from occurring.

4. Call performance for LR-on will be greatly improved by having no PSDS, or a PSDS no longer than 80 bytes, since some of the information to fill the PSDS after 80 bytes is costly to obtain. However, a formatted dump will not contain information for the system time and the date the program began running in the PSDS if the PSDS is not coded, or the length of the PSDS does not include those fields.

5. Refer to "Differences between OPM and ILE RPG/400 Exception Handling" on page 156 for comparison of ILE RPG/400 and OPM RPG/400 exception handling.

I/O

1. In ILE RPG/400 you can read a record in a file opened for update, and created or overriden with SHARE(*YES), and then update this locked record in another program that has opened the same file for update.

2. You cannot modify the MR indicator using the MOVE or SETON operations. (RPG III only prevents using SETON with MR.)

3. The File Type entry on the File specification no longer dictates the type of I/O operations which must be present in the Calculation specifications.

   For example, in RPG III, if you define a file as an update file, then you must have an UPDAT operation later in the program. This is no longer true in RPG IV. However, your file definition still must be consistent with the I/O operations present in the program. So if you have an UPDATE operation in your source, the file must be defined as an update file.

4. ILE RPG/400 will allow record blocking even if the COMMIT keyword is specified on the File Description specification.

5. In RPG IV, a file opened for update will also be opened as delete capable. You do not need any DELETE operations to make it delete capable.
6. In RPG IV, you do not have to code an actual number for the number of devices which will be used by a multiple-device file. If you specify MAXDEV(*FILE) on a File Description specification, then the number of save areas created for SAVEDS and SAVEIND is based on the number of devices which your file can handle. (The SAVEDS, SAVEIND and MAXDEV keywords on an RPG IV File Description specification correspond to the SAVDS, IND and NUM options on a RPG III File Description specification continuation line respectively.)

   In ILE RPG/400, you cannot specify as the number of program devices that can be acquired by the program as value which is different than than the maximum number of devices defined in the device file. OPM RPG/400 allowed this through the NUM option.
7. In ILE RPG/400, the ACQ and REL operation codes can be used with single device files.
8. In ILE RPG/400, the relative record number and key fields in the database-specific feedback section of the INFDS are updated on each input operation when doing blocked reads.
9. When a referential constraint error occurs in OPM RPG/400, the status code is set to "01299" (I/O error). In ILE RPG/400, the status code is set to "01022", "01222", or "01299", depending on the type referential constraint error that occurred:
   - If data management in not able to allocate a record due to a referential constraint error, a CPF502D notify message is issued. ILE RPG/400 will set the status code to "01222" and OPM RPG/400 will set the status code to "01299".

     If you have no error indicator or INFSR error subroutine, ILE RPG/400 will issue the RNQ1222 inquiry message, and OPM RPG/400 will issue the RPG1299 inquiry message. The main difference between these two messages is that RNQ1222 allows you to retry the operation.
   - If data management detects a referential constraint error which causes it to issue either a CPF503A or CPF502D notify message, ILE RPG/400 will set the status code to "01022" and OPM RPG/400 will set the status code to "01299".

     If you have no error indicator or INFSR error subroutine, ILE RPG/400 will issue the RNQ1022 inquiry message, and OPM RPG will issue the RPG1299 inquiry message.
   - All referential constraint errors detected by data management that cause data management to issue an escape message will cause both OPM and ILE RPG/400 to set the status code to "01299".
10. ILE RPG/400 relies more on data management error handling than does OPM RPG/400. This means that in some cases you will find certain error messages in the job log of an ILE RPG/400 program, but not an OPM RPG/400 program. Some differences you will notice in error handling are:
    - When doing an UPDATE on a record in a database file that has not been locked by a previous input operation, both ILE RPG/400 and OPM RPG/400 set the status code to "01211". ILE RPG/400 detects this situ-

ation when data management issues a CPF501B notify message and places it in the job log.

- When handling WORKSTN files, and you try to do I/O to a device that has not been acquired or defined, both ILE and OPM RPG/400 will set the status to "01281". ILE RPG/400 detects this situation when data management issues a CPF5068 escape message and places it in the job log.

11. When doing READE, REDPE (READPE in ILE), SETLL on a database file, or when doing sequential-within-limits processing by a record-address-file, OPM RPG/400 does key comparisons using the *HEX collating sequence. This may give different results than expected when the contents of the fields on which the access path is built differ from the actual contents of the fields.

    ILE RPG/400 does key comparisons using *HEX collating sequence only for pre-V3R1 DDM files. See "Using Pre-V3R1 DDM Files" on page 237. for more information.

DBCS Data in Character Fields

1. In OPM RPG/400, position 57 (Transparency Check) of the Control specification allows you to specify whether the RPG/400 compiler should scan character literals and constants for DBCS characters. If you specify that the compiler should scan for transparent literals, and if a character literal which starts with an apostrophe followed by a shift-out fails the transparency check, the literal is reparsed as a literal that is not transparent.

   In ILE RPG/400, there is no option on the Control specification to specify whether the compiler should perform transparency check on character literals. If a character literal contains a shift-out control character, regardless of the position of the shift-out character within the character literal, the shift-out character signifies the beginning of DBCS data. The compiler will check for the following:

   - There is a matching shift-in for the shift-out (that is, the shift-out and shift-in control characters are balanced)
   - There is an even number (minimally two) between the shift-in and the shift-out.
   - There is no embedded shift-out in the DBCS data.

   If the above conditions are not met, the compiler will issue a diagnostic message, and the literal will not be reparsed. As a result, if there are character literals in your OPM RPG programs which fail the transparency check performed by the OPM RPG compiler, such programs will get compilation errors in ILE RPG/400.

2. In OPM RPG/400, if there are two consecutive apostrophes enclosed within shift-out and shift-in control characters inside a character literal, the two consecutive apostrophes are considered as one single apostrophe if the character literal is not a transparent literal. The fact that the character literal is not a transparent literal can be due to:

   - the character literal does not start with an apostrophe followed by a shift-out
   - the character literal fails the transparency check performed by the compiler
   - the user has not specified that transparency check should be performed by the compiler

In ILE RPG/400, if there are two consecutive apostrophes enclosed within shift-out and shift-in control characters inside a character literal, the apostrophes will not be considered as a single apostrophe. A pair of apostrophes inside a character literal will only be considered as a single apostrophe if they are not enclosed within shift-out and shift-in.

If you have a character literal that contains a shift-out character that you do not want to be interpreted as a shift-out character, then you should specify the literal as a hexadecimal literal. For example, if you have a literal 'AoB' where 'o' represents shift-out, you should code this literal as X'C10EC2'.

Appendix B. Using the RPG III to RPG IV Conversion Aid

The RPG IV source specification layouts differ significantly from the System/38 environment RPG III and the OPM RPG/400 layouts. For example, the positions of entries on the specifications have changed and the types of specifications available have also changed. The RPG IV specification layouts are not compatible with the previous layouts. To take advantage of RPG IV features, you must convert RPG III and RPG/400 source members in your applications to the RPG IV source format.

Note: The valid types of source members you can convert are RPG, RPT, RPG38, RPT38, SQLRPG, and blank. The Conversion Aid does not support conversion of RPG36, RPT36, and other non-RPG source member types.

If you are in a hurry and want to get started, go to "Converting Your Source" on page 310 and follow the general directions.

Conversion Overview

You convert source programs to the RPG IV source format by calling the Conversion Aid through the CL command Convert RPG Source (CVTRPGSRC). The Conversion Aid converts:

- a single member
- all members in a source physical file
- all members with a common member-name prefix in the same file.

In order to minimize the likelihood of there being conversion problems, you can optionally have the /COPY members included in the converted source code. For convenience in reading the code, you can optionally have specification templates included in the converted source code.

The Conversion Aid converts each source member on a line-by-line basis. After each member conversion, it updates a log file with the status of the conversion if you have specified a log file on the command. You can also obtain a conversion report which includes information such as conversion errors, /COPY statements, CALL operations, and conversion status.

The Conversion Aid assumes that your source code is free of any compilation errors. If this is the case, then it will successfully convert most of your source code. However, in some cases, there may be a small amount of code which you may have to convert manually. Some of these cases are identified by the Conversion Aid. Others are not detected until you attempt to compile the converted source. To see which ones the Conversion Aid can identify, you can run the Conversion Aid using the unconverted member as input, and specify a conversion report but no output member. For information on the types of coding which cannot be converted see "Resolving Conversion Problems" on page 326.

File Considerations

The Conversion Aid operates on file members. This section presents information on different aspects of files which must be taken into consideration when using the Conversion Aid.

Source Member Types

Table 19 lists the various source member types, indicates whether the member type can be converted, and indicates the output source member type.

Table 19. Source Member Types and their Conversion Status

Source Member Type	Convert?	Converted Member Type
RPG	Yes	RPGLE
RPG38	Yes	RPGLE
RPT	Yes	RPGLE
RPT38	Yes	RPGLE
'blank'	Yes	RPGLE
RPG36	No	N/A
RPT36	No	N/A
SQLRPG	Yes	SQLRPGLE
Any other type	No	N/A

If the source member type is 'blank', then the Conversion Aid will assume it has a member type of RPG. If the source member type is blank for an auto report source member, then you should assign the correct source member type (RPT or RPT38) to the member before converting it. If you do, then the Conversion Aid will automatically expand the auto report source member so that it can be converted properly. The expansion is necessary since ILE RPG/400 does not support auto report source members.

For more information on converting auto report source members, see “Converting Auto Report Source Members” on page 318.

File Record Length

The recommended record length for the converted source physical file is 112 characters. This record length takes into account the RPG IV structure as shown in Figure 149. The recommended record length of 112 characters also corresponds to the maximum amount of information that fits on a line of a compiler listing.

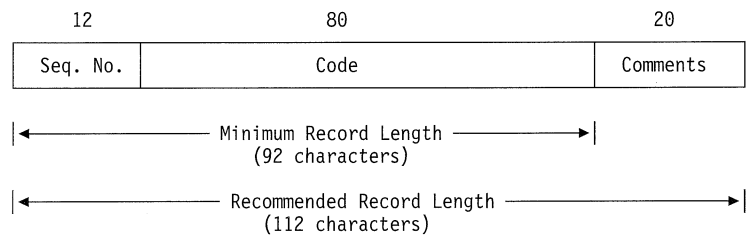


Figure 149. RPG IV Record Length Breakdown

If the converted source file has a record length less than 92 characters then an error message will be issued and the conversion will stop. This is because the record length is not long enough to contain the 80 characters allowed for source code and so some code is likely to be lost.

File and Member Names

The unconverted member and the member for the converted output can only have the same name if they are in different files or libraries.

The name of the converted source member(s) depends on whether you are converting one or several members. If you are converting one member, the default is to give the converted source member the same name as the unconverted member. You can, of course, specify a different name for the output member. If you are converting all source members in a file, or a group of them using a generic name, then the members will automatically be given the same name as the unconverted source members.

Note that specifying the file, library and member name for the converted output is optional. If you do not specify any of these names, the converted output will be placed in the file QRPGLESRC and have a member name the same as the unconverted member name. (The library list will be searched for the file QRPGLESRC.)

The Log File

The Conversion Aid uses a log file to provide audit trails on the status of each source member conversion. By browsing the log file, you can determine the status of previous conversions. You can access the log file with a user-written program for further processing, for example, compiling and binding programs.

If you specify that a log file is to be updated, then its record format must match the format of the IBM-suppled "model" database file QARNCVTLG in library QRPGLE. Figure 156 on page 325 shows the DDS for this file. Use the following CRTDUPOBJ command to create a copy of this model in your own library, referred to here as MYLIB. You may want to name your log file QRNCVTLG, as this is the default log file name for the Conversion Aid.

```
CRTDUPOBJ OBJ(QARNCVTLG) FROMLIB(QRPGLE) OBJTYPE(*FILE)
          TOLIB(MYLIB) NEWOBJ(QRNCVTLG)
```

You must have object management, operational and add authority to the log file which is accessed by the Conversion Aid.

For information on using the log file see "Using the Log File" on page 324.

Conversion Aid Tool Requirements

To use the Conversion Aid, you need the following authority:

- *USE authority for the CVTRPGSRC command
- *USE authority to the library which contains the source file and source members
- *CHANGE authority to the new library which will contain the source file and converted source members.
- object management, operational and add authority to the log file used by the Conversion Aid.

In addition to object authority requirements, there may be additional storage requirements. Each converted source program is, on average, about 25% larger than the size of the program before conversion. To use the Conversion Aid you need sufficient storage to store the converted source files.

What the Conversion Aid Won't Do

- The Conversion Aid does not support converting from the RPG IV format back to the RPG III or RPG/400 format.
- The RPG IV compiler does not support automatic conversion of RPG III or RPG/400 source members to the RPG IV source format *at compile time.*
- The Conversion Aid does not support converting RPG II source programs to the RPG IV source format. However, you can use the **RPG II to RPG III Conversion Aid** first and then the RPG III to RPG IV Conversion Aid.
- The Conversion Aid does not re-engineer source code, except where required (for example, the number of conditioning indicators.)
- The Conversion Aid does not create files. The log file and the output file must exist prior to running it.

Converting Your Source

This section discusses how to convert source programs to the RPG IV format. It explains the command CVTRPGSRC, which starts the Conversion Aid, and how to use it.

To convert your source code to the RPG IV format, follow these general steps:

1. If you use a data area as a Control specification, you must create a new one in the RPG IV format. Refer to the Control specifications section in *ILE RPG/400 Reference* for more information.
2. Create a log file, if necessary.

   Unless you specify LOGFILE(*NONE), there must be a log file for the Conversion Aid to access. If you do not have one, then you can create one by using the CRTDUPOBJ command. For more information, see “The Log File” on page 309 and “Using the Log File” on page 324.
3. Create the file for the converted source members.

   The Conversion Aid will not create any files. You must create the output file for the converted source prior to running the CVTRPGSRC command. The recommended name and record length for the output file is QRPGLESRC and 112 characters respectively. For additional file information see “File Considerations” on page 308.
4. Convert your source using the CVTRPGSRC command.

   You need to enter the name of the file and member to be converted. If you accept the defaults you will get a converted member in the file QRPGLESRC. The name of the member will correspond to the name of the unconverted source member. /COPY members will not be expanded in the converted source member, unless it is of type RPT or RPT38. A conversion report will be generated.

   See “The CVTRPGSRC Command” on page 311 for more information.
5. Check the log file or the error report for any errors. For information see “Analyzing Your Conversion” on page 321.

6. If there are errors, correct them and go to step 4.
7. If no errors, create your program. For information on creating ILE RPG/400 programs, see Chapter 5, "Creating a Program with the CRTBNDRPG Command" on page 37.
8. If your converted source member still has compilation problems they are most likely caused because your primary source member contains /COPY compiler directives. You have two choices to correct this situation:
   a. reconvert your source member specifying EXPCPY(*YES) to expand copy members into your converted source member
   b. manually correct any remaining errors using the compiler listing as a guide.

   Refer to "Resolving Conversion Problems" on page 326 for further discussion.
9. Once your converted source member has compiled successfully, retest the program before putting it back into production.

The CVTRPGSRC Command

To convert your RPG III or RPG/400 source to the new RPG IV format, you use the CVTRPGSRC command to start the Conversion Aid. Table 20 shows the parameters of the command based on their function.

Table 20. CVTRPGSRC Parameters and Their Default Values Grouped by Function

Program Identification	
FROMFILE	Identifies library and file name of RPG source to be converted.
FROMMBR	Identifies which source members are to be converted.
TOFILE(*LIBL/QRPGLESRC)	Identifies library and file name of converted output.
TOMBR(*FROMMBR)	Identifies file member names of converted source.
Conversion Processing	
TOMBR	If specified *NONE, then no file members are saved.
EXPCPY(*NO)	Determines if /COPY statements are included in converted output.
INSRTPL(*NO)	Indicate if specification templates are to be included in converted output.
Conversion Feedback	
CVTRPT(*YES)	Determine whether to produce conversion report.
SECLVL(*NO)	Determine whether to include second-level message text.
LOGFILE(*LIBL/QRNCVTLG)	Identifies log file for audit report.
LOGMBR(*FIRST)	Identifies which member of the log file to use for audit report.

The syntax for the CVTRPGSRC command is shown below.

Job: B,I Pgm: B,I REXX: B,I Exec

```
                           ┌─*LIBL/──────┐                                                  ┌─source-file-member-name─┐
►►──CVTRPGSRC──FROMFILE──(─┼─────────────┼──────source-file-name────)──FROMMBR──(─┼─*ALL────────────────────┼─)─────►
                           ├─*CURLIB/────┤                                                  └─generic*-member-name────┘
                           └─library-name/┘

►──┬──────────────────────────────────────────────────────────────────────────┬─┬──────────────────────────────────────────────┬─(P)──►
   └─TOFILE──(─┬─┬─*LIBL/──────────┬─┬─QRPGLESRC──────────┬─┬─)─┘ └─TOMBR──(─┬─*FROMMBR────────────────┬─)─┘
               │ ├─*CURLIB/────────┤ └─source-file-name───┘ │                 └─source-file-member-name─┘
               │ └─library-name/───┘                         │
               └─*NONE───────────────────────────────────────┘

►──┬──────────────────────────┬─┬──────────────────────────┬─┬──────────────────────────┬─┬───────────────────────────┬──►
   └─EXPCPY──(─┬─*NO──┬─)─┘ └─CVTRPT──(─┬─*YES─┬─)─┘ └─SECLVL──(─┬─*NO──┬─)─┘ └─INSRTPL──(─┬─*NO──┬─)─┘
               └─*YES─┘                 └─*NO──┘                 └─*YES─┘                  └─*YES─┘

►──┬─────────────────────────────────────────────────────────────────────┬─┬──────────────────────────────────────────┬──►◄
   └─LOGFILE──(─┬─┬─*LIBL/──────────┬─┬─QRNCVTLG───────┬─┬─)─┘ └─LOGMBR──(─┬─*FIRST───────────────┬─)─┘
                │ ├─*CURLIB/────────┤ └─log-file-name──┘ │                  ├─*LAST────────────────┤
                │ └─library-name/───┘                     │                  └─log-file-member-name─┘
                └─*NONE───────────────────────────────────┘
```

Note:
P All parameters preceding this point can be specified by position.

The parameters and their possible values follow the syntax diagram. If you need prompting, type CVTRPGSRC and press F4. The CVTRPGSRC screen appears, lists the parameters, and supplies default values. For a description of a parameter on the display, place your cursor on the parameter and press F1. Extended help for all of the parameters is available by pressing F1 on any parameter and then pressing F2.

FROMFILE
Specifies the name of the source file that contains the RPG III or RPG/400 source code to be converted and the library where the source file is stored. This is a required parameter; there is no default file name.

source-file-name
Enter the name of the source file that contains the source member(s) to be converted.

***LIBL**
The system searches the library list to find the library where the source file is stored.

***CURLIB**
The current library is used to find the source file. If you have not specified a current library, then the library QGPL is used.

library-name
Enter the name of the library where the source file is stored.

FROMMBR
Specifies the name(s) of the member(s) to be converted. This is a required parameter; there is no default member name.

The valid source member types of source members to be converted are RPG, RPT, RPG38, RPT38, SQLRPG and blank. The Convert RPG Source command does not support source member types RPG36, RPT36, and other non-RPG source member types (for example, CLP and TXT).

source-file-member-name
Enter the name of the source member to be converted.

**ALL*
The command converts all the members in the source file specified.

generic-member-name*
Enter the generic name of members having the same prefix in their names followed by a '*' (asterisk). The command converts all the members having the generic name in the source file specified. For example, specifying FROMMBR(PR*) will result in the conversion of all members whose names begin with 'PR'.

(See the CL Programmer's Guide for more information on the generic name.)

TOFILE
Specifies the name of the source file that contains converted source members and the library where the converted source file is stored. The converted source file must exist and should have a record length of 112 characters: 12 for the sequence number and date, 80 for the code and 20 for the comments.

QRPGLESRC
The default source file QRPGLESRC contains the converted source member(s).

***NONE**
No converted member is generated. The TOMBR parameter value is ignored. CVTRPT(*YES) must also be specified or the conversion will end immediately.

This feature allows you to find some potential problems without having to create the converted source member.

source-file-name
Enter the name of the converted source file that contains the converted source member(s).

The TOFILE source file name must be different from the FROMFILE source file name if the TOFILE library name is the same as the FROMFILE library.

***LIBL**
The system searches the library list to find the library where the converted source file is stored.

***CURLIB**
The current library is used to find the converted source file. If you have not specified a current library, then the library QGPL is used.

library-name
Enter the name of the library where the converted source file is stored.

TOMBR
Specifies the name(s) of the converted source member(s) in the converted source file. If the value specified on the FROMMBR parameter is *ALL or generic*, then TOMBR must be equal to *FROMMBR.

<u>*FROMMBR</u>
The member name specified in the FROMMBR parameter is used as the converted source member name. If FROMMBR(*ALL) is specified, then all the source members in the FROMFILE are converted. The converted source members have the same names as those of the original source members. If a generic name is specified in the FROMMBR parameter, then all the source members specified having the same prefix in their names are converted. The converted source members have the same names as those of the original generic source members.

source-file-member-name
Enter the name of the converted source member. If the member does not exist it will be created.

EXPCPY
Specifies whether or not /COPY member(s) is expanded into the converted source member. EXPCPY(*YES) should be specified only if you are having conversion problems pertaining to /COPY members.

Note: If the member is of type RPT or RPT38, EXPCPY(*YES) or EXPCPY(*NO) has no effect because the auto report program will always expand the /COPY members.

<u>*NO</u>
Do not expand the /COPY file member(s) into the converted source.

***YES**
Expands the /COPY file member(s) into the converted source.

CVTRPT
Specifies whether or not a conversion report is printed.

<u>*YES</u>
The conversion report is printed.

***NO**
The conversion report is not printed.

SECLVL

Specifies whether or not second-level text is printed in the conversion report in the message summary section.

***NO**

Second-level message text is not printed in the conversion report.

***YES**

Second-level message text is printed in the conversion report.

INSRTPL

Specifies if the ILE RPG/400 specification templates (H-, F-, D-, I-, C- and/or O-specification template), are inserted in the converted source member(s). The default value is *NO.

***NO**

A specification template is not inserted in the converted source member.

***YES**

A specification template is inserted in the converted source member. Each specification template is inserted at the beginning of the appropriate specification section.

LOGFILE

Specifies the name of the log file that is used to track the conversion information. Unless *NONE is specified, there must be a log file. The file must already exist, and it must be a physical data file. Create the log file by using the CRTDUPOBJ command with the "From object" file QARNCVTLG in library QRPGLE and the "New object" file QRNCVTLG in your library.

QRNCVTLG

The default log file QRNCVTLG is used to contain the conversion information.

***NONE**

Conversion information is not written to a log file.

log-file-name

Enter the name of the log file that is to be used to track the conversion information.

***LIBL**

The system searches the library list to find the library where the log file is stored.

library-name

Enter the name of the library where the log file is stored.

LOGMBR
Specifies the name of the log file member used to track conversion information. The new information is added to the existing data in the specified log file member.

If the log file contains no members, then a member having the same name as the log file is created.

***FIRST**
The command uses the first member in the specified log file.

***LAST**
The command uses the last member in the specified log file.

log-file-member-name
Enter the name of the log file member used to track conversion information.

Converting a Member Using the Defaults

You can take advantage of the default values supplied on the CVTRPGSRC command. Simply enter:

```
CVTRPGSRC FROMFILE(file name) FROMMBR(member name)
```

This will result in the conversion of the specified source member. The output will be placed in the file QRPGLESRC in whichever library in the library list contains this file. The /COPY members will not be expanded, no specification templates will be inserted, and the conversion report will be produced. The log file QRNCVTLG will be updated.

Note: The file QRPGLESRC and QRNCVTLG must already exist.

Converting All Members in a File

You can convert all of the members in a source physical file by specifying FROMMBR(*ALL) and TOMBR(*FROMMBR) on the CVTRPGSRC command. The Conversion Aid will attempt to convert all members in the file specified. If one member should fail to convert, the conversion process will still continue.

For example, if you want to convert all source members in the file QRPGSRC to the file QRPGLESRC, you would enter:

```
CVTRPGSRC  FROMFILE(OLDRPG/QRPGSRC)
           FROMMBR(*ALL)
           TOFILE(NEWRPG/QRPGLESRC)
           TOMBR(*FROMMBR)
```

This command converts all of the source members in library OLDRPG in the source physical file QRPGSRC. The new members are created in library NEWRPG in the source physical file QRPGLESRC.

If you prefer to keep all source (DDS source, RPG source, etc.) in the same file, you can still convert the RPG source members in one step, by specifying FROMMBR(*ALL). The Conversion Aid will only convert members with a valid RPG type (see Table 19 on page 308).

Converting Some Members in a File

If you need to convert only some members which are in a source physical file and they share a common prefix in the member name, then you can convert them by specifying the prefix followed by an * (asterisk).

For example, if you want to convert all members with a prefix of PAY, you would enter:

```
CVTRPGSRC  FROMFILE(OLDRPG/QRPGSRC)
           FROMMBR(PAY*)
           TOFILE(NEWRPG/QRPGLESRC)
           TOMBR(*FROMMBR)
```

This command converts all of the source members in library OLDRPG in the source physical file QRPGSRC. The new members are created in library NEWRPG in the source physical file QRPGLESRC.

Performing a Trial Conversion

You can do a trial run for any source member that you suspect you may have problems converting. You will then get a conversion report for the converted source member which may identify certain conversion errors.

For example, to perform a trial conversion on the source member PAYROLL, type:

```
CVTRPGSRC  FROMFILE(OLDRPG/QRPGSRC)
           FROMMBR(PAYROLL)
           TOFILE(*NONE)
```

The TOMBR parameter should be specified as *FROMMBR. However, since this is the default, you do not need to specify it unless the default value has been changed. The CVTRPT parameter should be specified as *YES — this is also the default. If it is not, then the conversion will stop immediately.

Using the TOFILE(*NONE) parameter stops the Conversion Aid from generating a converted member, but still allows it to produce a conversion report For more information on the conversion report, see “Analyzing Your Conversion” on page 321.

Obtaining Conversion Reports

The Conversion Aid normally produces a conversion report each time you issue the command. The name of the spooled file corresponds to the file name specified in the TOFILE parameter. If you try to convert a member that already exists or has an unsupported member type, then a message is printed in the job log indicating that these members have not been converted. The log file, if requested, is also updated to reflect this. However, no information regarding these members is placed in the report.

The conversion report includes the following information:

- CVTRPGSRC command options
- Source section which includes:
  conversion errors or warnings
  CALL operations
  /COPY directives
- Message summary
- Final summary

The conversion error messages provide you with suggestions on how to correct the error. In addition, any CALL operations and /COPY directives in the unconverted source are flagged to help you in identifying the various parts of the application you are converting. In general, you should convert all RPG components of an application at the same time.

If you do not want a conversion report, then specify CVTRPT(*NO).

Converting Auto Report Source Members

When an auto report source member (type RPT or RPT38) is detected in an RPG III or OPM RPG/400 source program, the Conversion Aid calls the CRTRPTPGM command to expand the source member and then converts it. (This is because auto report is not supported by ILE RPG/400.)

The auto report program produces a spooled file each time it is called by the Conversion Aid. You may want to check this file to see if any errors occurred on the auto report expansion since these errors will not be in the conversion report.

One particular error to check for in the auto report spooled file is one indicating that /COPY members were not found. The Conversion Aid will not know if these files are missing. However, without these files, it may not be able to successfully convert your source.

Note: If the source member type of the member to be converted is not RPT or RPT38 and the member *is* an auto report source member, you should assign the correct source member type (RPT or RPT38) to the member before converting it; otherwise conversion errors may occur.

Inserting Specification Templates

Because the source specifications for RPG IV are new, you may want to have specification templates inserted into the converted source. To have templates inserted specify INSRTPL(*YES) on the CVTRPGSRC command. The default is INSRTPL(*NO).

Converting Source from a Data File

The Conversion Aid will convert source from a data file. Because data files generally do not have sequence numbers, the minimum record length of the file for placing the converted output is 80 characters. (See Figure 149 on page 308.) The recommended record length is 100 characters for a data file.

Note: If your data file has sequence numbers, you should remove them prior to running the Conversion Aid.

Example of Source Conversion

The example shows a sample RPG III source member which is to be converted to RPG IV. Figure 150 shows the source of the RPG III version.

```
     H                                                                    TSTPGM
     FFILE1   IF  E                    DISK                               COMM1
     FQSYSPRT O   F     132      OF   LPRINTER
     LQSYSPRT  60FL 560L
     E                    ARR1    3   3  1               COMM2
     E                    ARR2    3   3  1
     IFORMAT1
     I              OLDNAME                         NAME
     I* DATA STRUCTURE COMMENT
     IDS1         DS
     I                                        1   3 FIELD1
     I* NAMED CONSTANT COMMENT
     I              'XYZ'                 C         CONST1                COMM3
     I                                        4   6 ARR1
     C           ARR1,3    DSPLY
     C                     READ FORMAT1                  01
     C           NAME      DSPLY
     C                     SETON                     LR
     C                     EXCPTOUTPUT
     OQSYSPRT E   01           OUTPUT
     O                         ARR2,3    10
**
123
**
456
```

Figure 150. RPG III Source for TEST1

To convert this source, enter:

```
CVTRPGSRC  FROMFILE(MYLIB/QRPGSRC) FROMMBR(TEST1)
           TOFILE(MYLIB/QRPGLESRC) INSRTPL(*YES)
```

The converted source is shown in Figure 151 on page 320.

```
 1 .....H*unctions+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++Comments+++++++++
 2      H DFTNAME(TSTPGM)
 3 .....F*ilename++IPEASFRlen+LKlen+AIDevice+.Functions++++++++++++++++++++++++++++Comments+++++++++
 4      FFILE1     IF   E             DISK                                         COMM1
 5      FQSYSPRT   O    F  132        PRINTER OFLIND(*INOF)
 6      F                                     FORMLEN(60)
 7      F                                     FORMOFL(56)
 8 .....D*ame+++++++++++ETDsFrom+++To/L+++IDc.Functions++++++++++++++++++++++++++++Comments+++++++++
 9      D ARR2            S              1    DIM(3) CTDATA PERRCD(3)
10      D* DATA STRUCTURE COMMENT
11      D DS1             DS
12      D  FIELD1                 1      3
13      D  ARR1                   4      6
14      D                                     DIM(3) CTDATA PERRCD(3)              COMM2
15      D* NAMED CONSTANT COMMENT
16      D CONST1          C                   CONST('XYZ')                         COMM3
17 .....I*ilename++SqNORiPos1+NCCPos2+NCCPos3+NCC..................................Comments+++++++++
18 .....I*.............Ext_field+Fmt+SPFrom+To+++DcField+++++++++L1M1FrP1MnZr......Comments+++++++++
19      IFORMAT1
20      I              OLDNAME                     NAME
21 .....C*0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....Comments+++++++++
22      C     ARR1(3)       DSPLY
23      C                   READ      FORMAT1                                01
24      C     NAME          DSPLY
25      C                   SETON                                        LR
26      C                   EXCEPT    OUTPUT
27      OQSYSPRT   E            OUTPUT           01
28      O                       ARR2(3)             10
29 **CTDATA ARR1
30 123
31 **CTDATA ARR2
32 456
```

Figure 151. Converted (RPG IV) Source for TEST1

Note the following about the converted source:

- The new specification types are H (Control), F (File), D (Definition), I (Input), C (Calculation) and O (Output); they must be entered in this order.

  The converted source contains specification templates for the new types since INSRTPL(*YES) was specified on CVTRPGSRC.

- The Control, File, and Definition specifications are keyword-oriented. See lines 2, 4 - 7, and 9 - 16.

- The ILE member has a new specification type, Definition. It is used to define stand-alone fields, arrays and named constants as well as data structures.

  In this example,

  – ARR2 is defined as a standalone array (Line 9)
  – Data structure DS1 is defined as a data structure with two subfields FIELD1 and ARR1 (Lines 11 - 14)
  – Constant CONST1 is defined as a constant (Line 16)

  The Input (I) specifications are now used only to define records and fields of a file. See Lines 19 - 20.

- The Extension (E) specifications have been eliminated. Arrays and tables are now defined using Definition specifications.

Record address file (RAF) entries on Extension specifications have been replaced by the keyword RAFDATA on the File Description specification.

- The Line Counter specifications have been eliminated. They have been replaced by the keywords FORMLEN and FORMOFL on the File Description specification. See Lines 6 and 7.
- All specification types have been expanded to allow for 10-character names for fields and files.
- In RPG IV, data structures (which are defined using Definition specifications) must precede the Input specifications.

  Note that in the converted source, the data structure DS1 (Line 11) has been moved to precede the specification containing the FORMAT1 information (Line 19).
- In RPG III, named constants can appear in the middle of a data structure. This is not allowed in RPG IV.

  In the converted source, CONST1 (Line 16) has been moved to follow data structure DS1 (Line 11).
- If a specification is moved, any comment which precedes it is also moved.

  In the converted source the comments above CONST1 and DS1 were moved with the following specifications.
- In RPG III, to define an array as a data structure subfield, you define both the array and a data structure subfield with the same name. This double definition is not allowed in RPG IV. Instead you specify the array attributes when you define the subfields using the new keyword syntax.

  In this example, ARR1 is defined twice in the OPM version, but has been merged into a single definition in converted source. See Lines 13 and 14.

  The merging of RPG III array specifications may result in the reordering of the array definitions. If the reordered arrays are compile-time arrays, then the loading of array data may be affected. To overcome this problem, RPG IV provides a keyword format for the ** records. Following ** you enter one of the keywords FTRANS, ALTSEQ or CTDATA. If the keyword is CTDATA, you enter the array or table name in positions 10 - 19.

  In this example the array ARR2 now precedes array ARR1, due to the merging of the two RPG III specifications for ARR2. The Conversion Aid has inserted the keywords and array names in the converted ** records which ensures the correct loading of the compile-time data. See Lines 29 and 31.
- Note that array syntax has changed. The notation ARR1,3 in RPG III is ARR1(3) in RPG IV. See line 28.

Analyzing Your Conversion

The Conversion Aid provides you with two ways to analyze your conversion results. They are:

- the conversion error report
- the log file.

Using the Conversion Report

The Conversion Aid generates a conversion report if you specify CVTRPT(*YES) parameter on the CVTRPGSRC command. The spooled file name is the same as the file name specified on the TOFILE parameter.

The conversion report consists of four parts:

1. CVTRPGSRC command options
2. source section
3. message summary
4. final summary

The first part of the listing includes a summary of the command options used by CVTRPGSRC. Figure 152 shows the command summary for a sample conversion.

```
5763RG1 V3R1M0  940909RN        IBM ILE RPG/400                                    AS400S01     09/30/94 20:41:35        Page       1

  Command . . . . . . . . . . . . :   CVTRPGSRC
    Issued by . . . . . . . . . . :     DAVE

  From file . . . . . . . . . . . :   QRPGSRC
    Library . . . . . . . . . . . :     MYLIB
  From member . . . . . . . . . . :   REPORT

  To file. . . . . . . . . . . . . :   QRPGLESRC
    Library . . . . . . . . . . . :     MYLIB
  To member . . . . . . . . . . . :   *FROMMBR

  Log file . . . . . . . . . . . . :   *NONE
    Library . . . . . . . . . . . :
  Log member . . . . . . . . . . . :   *FIRST

  Expand copy members. . . . . . . :   *NO
  Print conversion report . . . . :   *YES
  Include second level text. . . . :   *YES
  Insert specification template. . :   *YES
```

Figure 152. Command Summary of Sample Conversion Report

The source section includes lines which have informational, warning or error messages associated with them. These lines have an asterisk (*) in column 1 for ease of browsing in SEU. The message summary contains all three message types.

Two informational messages which may be of particular interest are:

- RNM0508 - flags /COPY statements
- RNM0511 - flags CALL operations

All /COPY members in an program must be converted in order for the corresponding ILE RPG/400 program to compile without errors. Similarly, you may want to convert all members related by CALL at the same time. Use this part of the report to assist you in identifying these members. Figure 153 on page 323 shows the source section for the sample conversion.

```
5763RG1 V3R1M0  940909RN        IBM ILE RPG/400                                      TORASD80     06/06/94 20:41:35        Page       2

  From file  . . . . . . . . . . . :   MYLIB/QRPGSRC(REPORT)
  To file. . . . . . . . . . . . . :   MYLIB/QRPGLESRC(REPORT)
  Log file . . . . . . . . . . . . :   *NONE

                     C o n v e r s i o n    R e p o r t

Sequence <----------------------- Source Specifications ---------------------------><-------------- Comments --------------> Page
Number   ....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8....+....9....+...10....+...11....+...12 Line

  000002 C                CALL      PROG1
*RNM0511 00 CALL operation code found.

  000003 C/COPY COPYCODE
*RNM0508 00 /COPY compiler directive found.

  000004 C                FREE      PROG2
*RNM0506 30 FREE operation code is not supported in RPG IV.

       * * * * *   E N D   O F   S O U R C E   * * * * *
```

Figure 153. Sample Source Section of Conversion Report

The message summary of the listing shows you the different messages which were issued. If you specify SECLVL(*YES), second-level messages will appear in the message summary. Figure 154 shows the messages section for the sample conversion, including second-level messages.

```
5763RG1 V3R1M0  940909RN        IBM ILE RPG/400                                      AS400S01     09/30/94 20:41:35        Page       2

                        M e s s a g e   S u m m a r y

 Msg id  Sv Number Message text

*RNM0508 00      1 /COPY compiler directive found.
                   Cause . . . . . :   In order for this RPG IV source to
                     compile correctly, ensure that all /COPY source members
                     included in this source member have also been converted to
                     RPG IV.
                   Recovery  . . . :   Ensure that all /COPY source
                     members are converted prior to compiling in RPG IV. In some
                     cases, problems may result when attempting to convert and
                     compile source members that make use of the /COPY compiler
                     directive.  If this situation results, specify *YES for the
                     EXPCPY parameter on the CVTRPGSRC command to expand the
                     /COPY member(s) into the converted source.  For further
                     information see the ILE RPG/400 Programmers Guide.

*RNM0511 00      1 CALL operation code found.
                   Cause . . . . . :   RPG specifications that contain CALL
                     operation codes have been identified because the user may
                     wish to:
                       -- change the CALL operation code to CALLB to take
                     advantage of static binding
                       -- convert all programs in an application to RPG IV.
                   Recovery  . . . :   Convert the CALL
                     operation code to a CALLB if you wish to take advantage of
                     static binding or convert the called program to RPG IV if
                     you wish to convert all programs in an application.

*RNM0506 30      1 FREE operation code is not supported in RPG IV.
                   Cause . . . . . :   The RPG III or RPG/400 program contains
                     the FREE operation code which is not supported in RPG IV.
                   Recovery  . . . :   Remove the FREE operation and replace
                     it with alternative code so that the programming logic is
                     not affected prior to compiling the converted source.

       * * * * *   E N D   O F   M E S S A G E   S U M M A R Y   * * * * *
```

Figure 154. Sample Message Summary of Conversion Report

The final summary of the listing provides message and record statistics. A final status message is also placed in the job log. Figure 155 on page 324 shows the messages section for the sample conversion.

```
                          F i n a l   S u m m a r y
Message Totals:
  Information  (00) . . . . . . . :       2
  Warning      (10) . . . . . . . :       0
  Severe Error (30+)  . . . . . . :       1
  --------------------------------  -------
  Total . . . . . . . . . . . . . :       3

Source Totals:
  Original Records Read . . . . . . :       3
  Converted Records Written . . . . :       4
  Highest Severity Message Issued . :      30

       * * * * *   E N D   O F   F I N A L   S U M M A R Y   * * * * *

       * * * * *   E N D   O F   C O N V E R S I O N   * * * * *
```

Figure 155. Sample Final Summary of Conversion Report

Using the Log File

By browsing the log file, you can see the results of your conversions. The log file is updated after each conversion operation. It tracks:

source members and their library names
converted source file names and their library names
highest severity error found.

For example, if no errors are found, the conversion status is set to 0. If severe errors are found, the status is set to 30.

If you try to convert a member with an unsupported member type or a member that already exists, then the conversion will not take place as this is a severe error (severity 40 or higher). A record will be added to the log file with conversion status set to 40. The TOFILE, TOMBR and TO LIBRARY will be set to blank to indicate that a TOMBR was not generated as the conversion did not take place.

The log file is an externally-described physical database file. A "model" of this file is provided in library QRPGLE in file QARNCVTLG. It has one record format called QRNCVTLG. All field names are six characters in length and follow the naming convention LGxxxx where xxxx describes the fields. Figure 156 on page 325 shows the DDS for this file.

```
     A          R QRNCVTFM
     A            LGCENT         1A          COLHDG('CVT' 'CENT')
     A                                       TEXT('Conversion Century: 0-20th 1-+
     A                                       21st')
     A            LGDATE         6A          COLHDG('CVT' 'DATE')
     A                                       TEXT('Conversion Date : format is Y+
     A                                       YMMDD')
     A            LGTIME         6A          COLHDG('CVT' 'TIME')
     A                                       TEXT('Conversion Time : format is H+
     A                                       HMMSS')
     A            LGSYST         8A          COLHDG('CVT' 'SYST')
     A                                       TEXT('Name of the system running co+
     A                                       nversion')
     A            LGUSER        10A          COLHDG('CVT' 'USER')
     A                                       TEXT('User Profile name of the user+
     A                                       running conversion')
     A            LGFRFL        10A          COLHDG('FROM' 'FILE')
     A                                       TEXT('From File')
     A            LGFRLB        10A          COLHDG('FROM' 'LIB')
     A                                       TEXT('From Library')
     A            LGFRMR        10A          COLHDG('FROM' 'MBR')
     A                                       TEXT('From Member')
     A            LGFRMT        10A          COLHDG('FMBR' 'TYPE')
     A                                       TEXT('From Member Type')
     A            LGTOFL        10A          COLHDG('TO' 'FILE')
     A                                       TEXT('To File')
     A            LGTOLB        10A          COLHDG('TO' 'LIB')
     A                                       TEXT('To Library')
     A            LGTOMR        10A          COLHDG('TO' 'MBR')
     A                                       TEXT('To Member')
     A            LGTOMT        10A          COLHDG('TMBR' 'TYPE')
     A                                       TEXT('To Member Type')
     A            LGLGFL        10A          COLHDG('LOG' 'FILE')
     A                                       TEXT('Log File')
     A            LGLGLB        10A          COLHDG('LOG' 'LIB')
     A                                       TEXT('Log Library')
     A            LGLGMR        10A          COLHDG('LOG' 'MBR')
     A                                       TEXT('Log Member')
     A            LGCEXP         1A          COLHDG('CPY' 'EXP')
     A                                       TEXT('Copy Member Expanded: Y=Yes, +
     A                                       N=No')
     A            LGERRL         1A          COLHDG('CVT' 'RPT')
     A                                       TEXT('Conversion Report Printed: Y=+
     A                                       Yes, N=No')
     A            LGSECL         1A          COLHDG('SEC' 'LVL')
     A                                       TEXT('Second Level Text Printed: Y=+
     A                                       Yes, N=No')
     A            LGINSR         1A          COLHDG('INSR' 'TPL')
     A                                       TEXT('Template Inserted: Y=Yes, N=N+
     A                                       o')
     A            LGSTAT         2A          COLHDG('CVT' 'STAT')
     A                                       TEXT('Conversion Status')
     A            LGMRDS        50A          COLHDG('MBR' 'DESC')
     A                                       TEXT('Member Description')
```

Figure 156. DDS for model log file QARNCVTLG in library QRPGLE

Resolving Conversion Problems

Conversion problems may arise for one or more of the following reasons:

- the RPG III source has compilation errors
- certain features of the RPG III language are not supported by RPG IV
- one or more /COPY compiler directives exists in the RPG III source
- use of externally-described data structures
- behavioral differences between the OPM and ILE run time

Each of these areas is discussed in the sections which follow.

Compilation Errors in Existing RPG III Code

The Conversion Aid assumes that you are attempting to convert a valid RPG III program, that is, a program with no compilation errors. If this is not the case, then unpredictable results may occur during conversion. If you believe your program contains compilation errors, compile it first using the RPG III compiler and correct any errors before performing the conversion.

Unsupported RPG III Features

A few features of the RPG III language are *not* supported in RPG IV. The most notable of these are:

- the auto report function
- the FREE operation code
- the DEBUG operation code

Since the auto report function is not supported, the Conversion Aid will automatically expand these programs (that is, call auto report) if the type is RPT or RPT38 prior to performing the conversion.

You must replace the FREE or DEBUG operation code with equivalent logic either before or after conversion.

If you specify the CVTRPT(*YES) option on the CVTRPGSRC command, you will receive a conversion report which will identify most of these types of problems. For further information on converting auto report members, see “Converting Auto Report Source Members” on page 318. For further information on differences between RPG III and RPG IV, see Appendix A, “Behavioral Differences Between OPM RPG/400 and ILE RPG/400” on page 301.

Use of the /COPY Compiler Directive

In some cases errors will not be found until you actually compile the converted RPG IV source. Conversion errors of this type are usually related to the use of the /COPY compiler directive. These fall into two categories: merging problems and context-sensitive problems. Following is a discussion of why these problems occur and how you might resolve them.

Merging Problems

Because of differences between the RPG III and RPG IV languages, the Conversion Aid must reorder certain source statements. An example of this reordering is shown in "Example of Source Conversion" on page 319 for the RPG III source member TEST1. If you compare the placement of the data structure DS1 in Figure 150 on page 319 and in Figure 151 on page 320, you can see that the data structure DS1 was moved above the record format FORMAT1.

Now suppose that the RPG III member TEST1 was split into two members TEST2 and COPYDS1 where the data structure DS1 and the named constant CONST1 are in a copy member COPYDS1 and that this copy member is included in source TEST2. Figure 157 and Figure 158 show the source for TEST2 and COPYDS1 respectively.

```
     H                                                                 TSTPGM
     FFILE1   IF  E                    DISK                            COMM1
     FQSYSPRT O   F     132     OF    LPRINTER
     LQSYSPRT  60FL 56OL
     E                 ARR1    3   3  1                COMM2
     E                 ARR2    3   3  1
     IFORMAT1
     I           OLDNAME                        NAME
      /COPY COPYDS1
     C          ARR1,3    DSPLY
     C                    READ FORMAT1                01
     C          NAME      DSPLY
     C                    SETON                     LR
     C                    EXCPTOUTPUT
     OQSYSPRT E   01          OUTPUT
     O                        ARR2,3    10
**
123
**
456
```

Figure 157. Source for TEST2

```
     I* DATA STRUCTURE COMMENT
     IDS1        DS
     I                                        1   3 FIELD1
     I* NAMED CONSTANT COMMENT
     I           'XYZ'                C         CONST1                 COMM3
     I                                        4   6 ARR1
```

Figure 158. Source for COPYDS1

In this situation the Conversion Aid would convert both member TEST2 and the copy member COPYDS1 correctly. However, when the copy member is included at compile time, it will be inserted below FORMAT1 because this is where the /COPY directive is. As a result, all source lines in the copy member COPYDS1 will get a "source record is out of sequence" error because in RPG IV, Definition specifications must precede Input specifications.

Note that the Conversion Aid could not move the /COPY directive above FORMAT1 because the contents of /COPY member are unknown.

You have two methods to correct this type of problem:

1. use the EXPCPY(*YES) option of the CVTRPGSRC command to include all /COPY members in the converted RPG IV source member

   This approach is easy and will work most of the time, however, including the /COPY members in each source member reduces the maintainability of your application.

2. manually correct the code after conversion using the information in the ILE RPG/400 compiler listing and the *ILE RPG/400 Reference*.

Other examples of this type of problem include:

- Line Specifications and Record Address Files

  In RPG III the Line Counter specification and the Record Address File of the Extension specification are changed to keywords (RAFDATA, FORMLEN and FORMOFL) on the File Description specification. If the content of a /COPY member contains only the Line Counter specification and/or the Record Address File of the Extension specification but not the corresponding File Description Specification, the Conversion Aid does not know where to insert the keywords.

- Extension Specification Arrays and Data Structure Subfields

  As mentioned in "Example of Source Conversion" on page 319, you are not allowed to define a stand-alone array and a data structure subfield with the same name in RPG IV. Therefore as shown in the example TEST1 (Figure 151 on page 320), the Conversion Aid must merge these two definitions. However, it the array and the data structure subfield are not in the same source member (that is, one or both is in a /COPY member) this merging cannot take place and a compile-time error will result.

- Merged compile time array and compile time data (**) records

  As shown in the example TEST1 (Figure 151 on page 320), if compile-time arrays are merged with data structure subfield definitions the loading of array data may be effected. To overcome this problem, compile-time array data are changed to the new **CTDATA format if at least one compile-time array is merged. However, if the arrays and the data do not reside in the same source file (that is, one or both is in a COPY member) the naming of compile-time data records using the **CTDATA format cannot proceed properly.

Context-Sensitive Problems

In RPG III, there are occasions when it is impossible to determine the type of specifications in a /COPY member without the context of the surrounding specifications of the primary source member. There are two instances of this problem:

- Data structure subfield or program-described file field

```
I* If the RPG III source member contains only the source
I* statements describing fields FIELD1 and FIELD2 below, the
I* Conversion Aid is unsure how to convert them.  These
I* statements may be data structure fields (which are converted
I* to Definition specifications) or program-described file
I* fields (which are converted to Input specifications).
I                                       1   3 FIELD1
I                                       4   6 FIELD2
```

Figure 159. RPG III Source

- Rename of an externally-described data structure field or a rename of an externally-described file field

```
I* If the RPG III source member contains only the source
I* statement describing field CHAR below, the Conversion
I* Aid is unsure how to convert it.  This statement may be
I* a rename of an externally-described data structure field
I* which is converted to a Definition specification) or
I* a rename of an externally-described file field)
I* (which is converted to an Input specification).
I              CHARACTER                        CHAR
```

Figure 160. RPG III Source

In the above two instances, a data structure is assumed and Definition specifications are produced. A block of comments containing the Input specification code is also produced. For example, the Conversion Aid will convert the source in Figure 159 to the code shown in Figure 161. If Input specification code is required, delete the Definition specifications and blank out the asterisks from the corresponding Input specifications.

```
D* If the RPG III source member contains only the source
D* statements describing fields FIELD1 and FIELD2 below, the
D* Conversion Aid is unsure how to convert them.  These
D* statements may be data structure fields (which are converted
D* to Definition specifications) or program-described file
D* fields (which are converted to Input specifications).
D FIELD1                1      3
D FIELD2                4      6
I*                               1    3  FIELD1
I*                               4    6  FIELD2
```

Figure 161. RPG IV Source

Remember, you have two choices to correct these types of problems. Either use the EXPCPY(*YES) option of the CVTRPGSRC command or manually correct the code after conversion.

Use of Externally-Described Data Structures

There are two problems that you may have to fix manually even though you specify the EXPCPY(*YES) option on the CVTRPGSRC command. They are related to the use of externally-described data structures

Because these problems will generate compile-time errors, you can use the information in the ILE RPG/400 compiler listing and the *ILE RPG/400 Reference* to correct them.

Merging an Array with an Externally-Described DS Subfield

As mentioned earlier, you are not allowed to define a stand-alone array and a data structure subfield with the same name in RPG IV. In general, the Conversion Aid will merge these two definitions. However, if the subfield is in an externally-described data structure this merging is not handled and you will be required to manually correct the converted source member.

For example, the field ARRAY in Figure 162 is included twice in Figure 163. It is included once as a stand-alone array and once in the externally-described data structure EXTREC. When converted the RPG IV source generated is shown in Figure 164. This code will not compile since ARRAY is defined twice. In order to correct this problem, delete the stand-alone array and add a subfield with the keywords to data structure DSONE as shown Figure 165.

```
A          R RECORD
A            CHARACTER     10
A            ARRAY         10
```

Figure 162. DDS

```
E                    ARRAY      10  1
IDSONE     E DSEXTREC
C           CHAR      DSPLY
C                     SETON                     LR
```

Figure 163. RPG III Source

```
D ARRAY           S              1   DIM(10)
D DSONE         E DS                 EXTNAME(EXTREC)
C     CHAR          DSPLY
C                   SETON                                        LR
```

Figure 164. RPG IV Source

```
D DSONE         E DS                 EXTNAME(EXTREC)
D  ARRAY        E                    DIM(10)
C     CHAR          DSPLY
C                   SETON                                        LR
```

Figure 165. Corrected RPG IV Source

Renaming and Initializing an Externally-Described DS Subfield

In RPG III, when both renaming and initializing a field in an externally described data structure you had to use two source lines as shown for the field CHAR in Figure 166. The converted source also contains two source lines, as shown in Figure 167. This will result in a compile-time error as the field CHAR is defined twice. To correct this code you must combine the keywords of the field CHAR into a single line as shown in Figure 168 where the key fields INZ and EXTFLD have been combined and only one instance on the field CHAR is shown.

```
IDSONE     E DSEXTREC
I              CHARACTER                      CHAR
I I           'XYZ'                           CHAR
C           CHAR      DSPLY
C                     SETON                    LR
```

Figure 166. RPG III Source

```
D DSONE         E DS                    EXTNAME(EXTREC)
D  CHAR         E                       EXTFLD(CHARACTER)
D  CHAR         E                       INZ('XYZ')
C     CHAR          DSPLY
C                   SETON                                        LR
```

Figure 167. RPG IV Source

```
D DSONE         E DS                    EXTNAME(EXTREC)
D  CHAR         E                       EXTFLD(CHARACTER) INZ('XYZ')
C     CHAR          DSPLY
C                   SETON                                        LR
```

Figure 168. Corrected RPG IV Source

Run-time Differences

If you have pre-run-time arrays which overlap in data structures, the order of loading these arrays at run time may be different in RPG III and RPG IV. This can cause the data in the overlapping section to differ. The order in which the arrays are loaded is the order they are encountered in the source, which may have changed because the arrays have been merged with the subfields during conversion.

In general, you should avoid situations where an application consists of OPM and ILE programs which are split across the OPM default activation group and a named activation group. When spilt across these two activation groups you are mixing OPM behavior with ILE behavior and your results may be hard to predict. Please see Chapter 3, "Program Creation Strategies" on page 19 or *ILE Concepts*.

Appendix C. The Create Commands

This section provides information on:

- Using CL commands
- Syntax diagram and description of CRTBNDRPG
- Syntax diagram and description of CRTRPGMOD

For information on the Create Program and Create Service Program commands, see *CL Reference*.

Using CL Commands

Control Language (CL) commands, **parameters**, and **keywords** can be entered in either uppercase or lowercase characters. In the syntax diagram they are shown in uppercase (for example, PARAMETER, PREDEFINED-VALUE). Variables appear in lowercase italic letters (for example, *user-defined-value*). Variables are user-defined names or values.

How to Interpret Syntax Diagrams

The syntax diagrams in this book use the following conventions:

```
►►──PARAMETER──(──┬──────────────────┬──user-defined-value──)──────────────────────►◄
                  └─PREDEFINED-VALUE─┘
```

Figure 169. Structure of a Syntax Diagram

Read the syntax diagram from left to right, and from top to bottom, following the path of the line.

The ►►── symbol indicates the beginning of the syntax diagram.

The ──►◄ symbol indicates the end of the syntax diagram.

The ───► symbol indicates that the statement syntax is continued on the next line.

The ►─── symbol indicates that a statement is continued from the previous line.

The ──(──)── symbol indicates that the parameter or value must be entered in parentheses.

Required parameters appear on the base line and must be entered. **Optional parameters** appear below the base line and do not need to be entered. In the following sample, you must enter REQUIRED-PARAMETER and a value for it, but you do not need to enter OPTIONAL-PARAMETER or a value for it.

```
►►──REQUIRED-PARAMETER──(──┬─PREDEFINED-VALUE───┬──)─────────────────────────────►
                           └─user-defined-value─┘
►──┬─────────────────────────────────────────────────────────┬──────────────────►◄
   └─OPTIONAL-PARAMETER──(──┬─PREDEFINED-VALUE───┬──)─┘
                            └─user-defined-value─┘
```

Default values appear above the base line and do not need to be entered. They are used when you do not specify a parameter. In the following sample, you can enter DEFAULT-VALUE, OTHER-PREDEFINED-VALUE, or nothing. If you enter nothing, DEFAULT-VALUE is assumed.

```
                  ┌─DEFAULT-VALUE──────────┐
►►──PARAMETER──(──┴─OTHER-PREDEFINED-VALUE─┴──)──────────────────────────────►◄
```

Optional values are indicated by a blank line. The blank line indicates that a value from the first group (OPTIONAL-VALUE1, OPTIONAL-VALUE2, *user-defined-value*) does not need to be entered. For example, based on the syntax below, you could enter KEYWORD(REQUIRED-VALUE).

```
                  ┌─OPTIONAL-VALUE1────┐
►►──PARAMETER──(──┼────────────────────┼──REQUIRED-VALUE──)───────────────────►◄
                  ├─OPTIONAL-VALUE2────┤
                  └─user-defined-value─┘
```

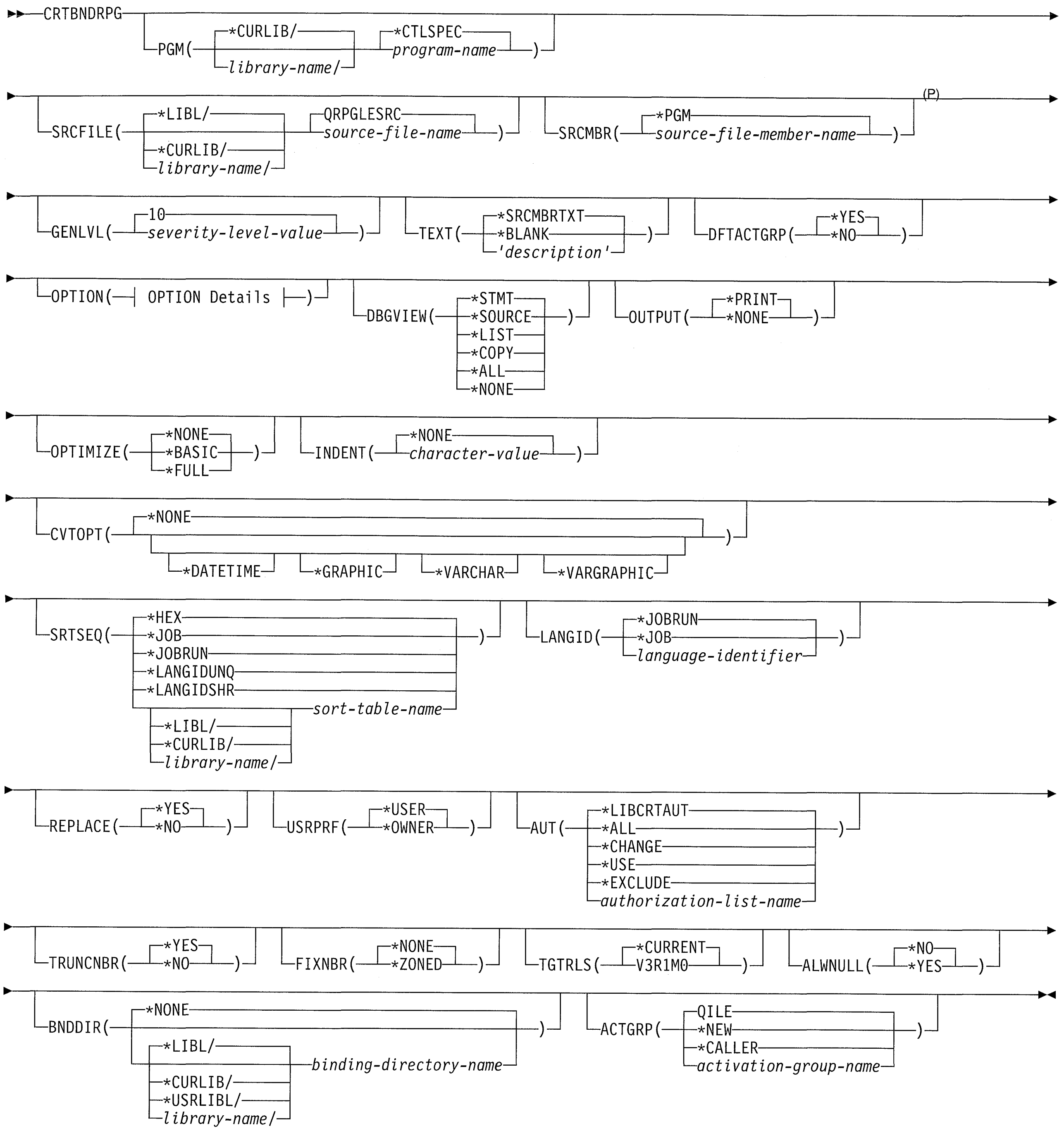
Repeated values can be specified for some parameters. The comma (,) in the following sample indicates that each *user-defined-value* must be separated by a comma.

```
                 ┌─,──────────────────┐
►►──KEYWORD──(───▼─user-defined-value─┴──)────────────────────────────────────►◄
```

CRTBNDRPG Command

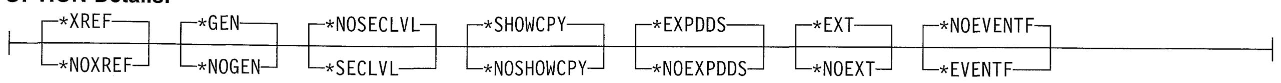
The Create Bound RPG (CRTBNDRPG) command performs the combined tasks of the Create RPG Module (CRTRPGMOD) and Create Program (CRTPGM) commands by creating a temporary module object from the source code, and then creating the program object. Once the program object is created, CRTBNDRPG deletes the module object it created. The entire syntax diagram for the CRTBNDRPG command is shown below.

Job: B,I Pgm: B,I REXX: B,I Exec


CRTBNDRPG
PGM(*CURLIB/ library-name/ *CTLSPEC program-name)
SRCFILE(*LIBL/ *CURLIB/ library-name/ QRPGLESRC source-file-name)
SRCMBR(*PGM source-file-member-name)
(P)
GENLVL(10 severity-level-value)
TEXT(*SRCMBRTXT *BLANK 'description')
DFTACTGRP(*YES *NO)
OPTION(OPTION Details)
DBGVIEW(*STMT *SOURCE *LIST *COPY *ALL *NONE)
OUTPUT(*PRINT *NONE)
OPTIMIZE(*NONE *BASIC *FULL)
INDENT(*NONE character-value)
CVTOPT(*NONE *DATETIME *GRAPHIC *VARCHAR *VARGRAPHIC)
SRTSEQ(*HEX *JOB *JOBRUN *LANGIDUNQ *LANGIDSHR *LIBL/ *CURLIB/ library-name/ sort-table-name)
LANGID(*JOBRUN *JOB language-identifier)
REPLACE(*YES *NO)
USRPRF(*USER *OWNER)
AUT(*LIBCRTAUT *ALL *CHANGE *USE *EXCLUDE authorization-list-name)
TRUNCNBR(*YES *NO)
FIXNBR(*NONE *ZONED)
TGTRLS(*CURRENT V3R1M0)
ALWNULL(*NO *YES)
BNDDIR(*NONE *LIBL/ *CURLIB/ *USRLIBL/ library-name/ binding-directory-name)
ACTGRP(QILE *NEW *CALLER activation-group-name)

Note:
P All parameters preceding this point can be specified by position.

OPTION Details:



Description of the CRTBNDRPG Command

The parameters, keywords, and variables of the CRTBNDRPG command are listed below. The same information is available online. Enter the command name on a command line, press PF4 (Prompt) and then press PF1 (Help) for any parameter you want information on.

PGM

Specifies the program name and library name for the program object (*PGM) you are creating. The program name and library name must conform to AS/400 naming conventions. If no library is specified, the created program is stored in the current library.

***CTLSPEC**

The name for the compiled program is taken from the name specified in the DFTNAME keyword of the control specification. If the program name is not specified on the control specification and the source member is from a database file, the member name, specified by the SRCMBR parameter, is used as the program name. If the source is not from a database file then the program name defaults to RPGPGM.

program-name

Enter the name of the program object.

***CURLIB**

The created program object is stored in the current library. If you have not specified a current library, QGPL is used.

library-name

Enter the name of the library where the created program object is to be stored.

SRCFILE

Specifies the name of the source file that contains the ILE RPG source member to be compiled and the library where the source file is located. The recommended source physical file length is 112 characters: 12 for the sequence number and date, 80 for the code and 20 for the comments. This is the maximum amount of source that is shown on the compiler listing.

QRPGLESRC

The default source file QRPGLESRC contains the ILE RPG source member to be compiled.

source-file-name

Enter the name of the source file that contains the ILE RPG source member to be compiled.

***LIBL**

The system searches the library list to find the library where the source file is stored. This is the default.

***CURLIB**
The current library is used to find the source file. If you have not specified a current library, QGPL is used.

library-name
Enter the name of the library where the source file is stored.

SRCMBR
Specifies the name of the member of the source file that contains the ILE RPG source program to be compiled.

***PGM**
Use the name specified by the PGM parameter as the source file member name. The compiled program object will have the same name as the source file member. If no program name is specified by the PGM parameter, the command uses the first member created in or added to the source file as the source member name.

source-file-member-name
Enter the name of the member that contains the ILE RPG source program.

GENLVL
Specifies whether or not a program object is generated, depending on the severity of the errors encountered.

A severity-level value corresponding to the severity level of the messages produced during compilation can be specified with this parameter. The value must be between 0 and 20 inclusive. If the severity-level value specified is greater than or equal to the maximum severity of any errors encountered, then the program object is generated. However, the program produced may contain errors that cause unpredictable results when run if the GENLVL option is set to a value greater than 10.

For errors greater than severity 20, the program object will not be generated.

10 A program object will not be generated if you have messages with a severity-level greater than 10.

severity-level-value
Enter a number, 0 through 20 inclusive. The generation severity level GENLVL controls the creation of the program object. The program object is created if all the compile messages have severity level below the generation severity level.

TEXT
Allows you to enter text that briefly describes the program and its function. The text appears whenever program information is displayed.

***SRCMBRTXT**
The text of the source member is used.

***BLANK**
No text appears.

'description'
Enter the text that briefly describes the function of the source specifications. The text can be a maximum of 50 characters and must be enclosed in apostrophes. The apostrophes are not part of the 50-character string. Apostrophes are not required if you are entering the text on the prompt screen.

DFTACTGRP
Specifies whether the created program is intended to always run in the default activation group.

<u>*YES</u>
When this program is called it will always run in the default activation group. The default activation group is the activation group where all original program model (OPM) programs are run.

Specifying DFTACTGRP(*YES) allows ILE RPG/400 programs to behave like OPM programs in the areas of override scoping, open scoping, and RCLRSC.

ILE static binding is not available when a program is created with DFTACTGRP(*YES). This means that your program cannot contain a CALLB operation, nor can you use the BNDDIR or ACTGRP parameters when creating this program.

DFTACTGRP(*YES) is useful when attempting to move an application on a program-by-program basis to ILE RPG/400.

***NO**
The program is associated with the activation group specified by the ACTGRP parameter. Static binding is allowed when *NO is specified.

If ACTGRP(*CALLER) is specified and this program is called by a program running in the default activation group, then this program will behave according to ILE semantics in the areas of file sharing, file scoping and RCLRSC.

DFTACTGRP(*NO) is useful when you intend to take advantage of ILE concepts, for example, running in a named activation group or binding to a service program.

OPTION
Specifies the options to use when the source member is compiled. You can specify any or all of the options in any order. Separate the options with one or more blank spaces. If an option is specified more than once, the last one is used.

<u>*XREF</u>
Produces a cross-reference listing (when appropriate) for the source member.

***NOXREF**
A cross-reference listing is not produced.

***GEN**
Create a program object if the highest severity level returned by the compiler does not exceed the severity specified in the GENLVL option.

***NOGEN**
Do not create a program object.

***NOSECLVL**
Do not print second-level message text on the line following the first-level message text.

***SECLVL**
Print second-level message text on the line following the first-level message text in the Message Summary section.

***SHOWCPY**
Show source records of members included by the /COPY compiler directive.

***NOSHOWCPY**
Do not show source records of members included by the /COPY compiler directive.

***EXPDDS**
Show the expansion of externally-described files in the listing.

***NOEXPDDS**
Do not show the expansion of externally described files in the listing.

***EXT**
Show the list of external procedures and fields referenced during the compile on the listing.

***NOEXT**
Do not show the list of external procedures and fields referenced during the compilation on the listing.

***NOEVENTF**
Do not create an Event File for use by CoOperative Development Environment/400 (CODE/400). CODE/400 uses this file to provide error feedback integrated with the CODE/400 editor. An Event File is normally created when you create a module or program from within CODE/400.

***EVENTF**
Create an Event File for use by CoOperative Development Environment/400 (CODE/400). The Event File is created as a member in file EVFEVENT in the library where the created module or program object is to be stored. If the file EVFEVENT does not exist it is automatically created. The Event File member name is the same as the name of the object being created.

CODE/400 uses this file to provide error feedback integrated with the CODE/400 editor. An Event File is normally created when you create a module or program from within CODE/400.

DBGVIEW

Specifies which level of debugging is available for the compiled program object, and which source views are available for source-level debugging.

***STMT**

Allows the program object to be debugged using the Line Numbers shown on the left-most column of the source section of the compiler listing.

***SOURCE**

Generates the source view for debugging the compiled program object. This view is not available if the root source member is a DDM file. Also, if changes are made to any source members after the compile and before attempting to debug the program, the views for those source members may not be usable.

***LIST**

Generates the listing view for debugging the compiled program object. The information contained in the listing view is dependent on whether *SHOWCPY and *EXPDDS are specified for the OPTION parameter.

Note: The listing view will not show any indentation which you may have requested using the Indent option.

***COPY**

Generates the source and copy views for debugging the compiled program object. The source view for this option is the same source view generated for the *SOURCE option. The copy view is a debug view which has all the /COPY source members included. These views will not be available if the root source member is a DDM file. Also, if changes are made to any source members after the compile and before attempting to debug the program, the views for those source members may not be usable.

***ALL**

Generates the listing, source and copy views for debugging the compiled program object. The information contained in the listing view is dependent on whether *SHOWCPY and *EXPDDS are specified for the OPTION parameter.

***NONE**

Disables all of the debug options for debugging the compiled program object.

OUTPUT

Specifies if a compiler listing is generated.

***PRINT**

Produces a compiler listing, consisting of the ILE RPG/400 program source and all compile-time messages. The information contained in the listing is dependent on whether *XREF, *SECLVL, *SHOWCPY, *EXPDDS and *EXT are specified for the OPTION parameter.

***NONE**
Do not generate the compiler listing.

OPTIMIZE
Specifies the level of optimization, if any, of the program.

***NONE** (default)
Generated code is not optimized. This is the fastest in terms of translation time. It allows variables to be displayed and modified while in debug mode.

***BASIC**
Some optimization is performed on the generated code. This allows user variables to be displayed but not modified while in debug.

***FULL**
Optimization which generates the most efficient code. Translation time is the longest. User variables may not be modified but may be displayed although the presented values may not be current values.

INDENT
Specifies whether structured operations should be indented in the source listing for enhanced readability. Also specifies the characters that are used to mark the structured operation clauses.

Note: Any indentation which you request here will not be reflected in the listing debug view which is created when you specify DBGVIEW(*LIST).

***NONE**
Structured operations will not be indented in the source listing.

character-value
The source listing is indented for structured operation clauses. Alignment of statements and clauses are marked using the characters you choose. You can choose any character string up to 2 characters in length. If you want to use a blank in your character string, you must enclose it in single quotation marks.

Note: The indentation may not appear as expected if there are errors in the program.

CVTOPT
Specifies how the ILE RPG compiler handles date, time, timestamp, graphic data types, and variable-length data types which are retrieved from externally-described database files.

***NONE**
Ignores variable-length database data types and use the native RPG date, time, timestamp and graphic data types.

***DATETIME**
Specifies that date, time, and timestamp database data types are to be declared as fixed-length character fields.

***GRAPHIC**
Specifies that double-byte character set (DBCS) graphic data types are to be declared as fixed-length character fields.

***VARCHAR**
Specifies that variable-length character data types are to be declared as fixed-length character fields.

***VARGRAPHIC**
Specifies that variable-length double-byte character set (DBCS) graphic data types are to be declared as fixed-length character fields.

SRTSEQ
Specifies the sort sequence table that is to be used in the ILE RPG source program.

<u>*HEX</u>
No sort sequence table is used.

***JOB**
Use the SRTSEQ value for the job when the *PGM is created.

***JOBRUN**
Use the SRTSEQ value for the job when the *PGM is run.

***LANGIDUNQ**
Use a unique weighted table. This special value is used in conjunction with the LANGID parameter to determine the proper sort sequence table.

***LANGIDSHR**
Use a shared weighted table. This special value is used in conjunction with the LANGID parameter to determine the proper sort sequence table.

sort-table-name
Enter the qualified name of the sort sequence table to be used with the program.

***LIBL**
The system searches the library list to find the library where the sort sequence table is stored.

***CURLIB**
The current library is used to find the sort sequence table. If you have not specified a current library, QGPL is used.

library-name
Enter the name of the library where the sort sequence table is stored.

LANGID

Specifies the language identifier to be used when the sort sequence is *LANGIDUNQ and *LANGIDSHR. The LANGID parameter is used in conjunction with the SRTSEQ parameter to select the sort sequence table.

<u>*JOBRUN</u>
Use the LANGID value associated with the job when the RPG program is executed.

***JOB**
Use the LANGID value associated with the job when the RPG program is created.

language-identifier
Use the language identifier specified. (For example, FRA for French and DEU for German.)

REPLACE

Specifies if a new program is created when a program of the same name already exists in the specified (or implied) library. The intermediate module created during the processing of the CRTBNDRPG command are not subject to the REPLACE specifications, and have an implied REPLACE(*NO) against the QTEMP library. The intermediate modules is deleted once the CRTBNDRPG command has completed processing.

<u>*YES</u>
A new program is created in the specified library. The existing program of the same name in the specified library is moved to library QRPLOBJ.

***NO**
A new program is not created if a program of the same name already exists in the specified library. The existing program is not replaced, a message is displayed, and compilation stops.

USRPRF

Specifies the user profile that will run the created program object. The profile of the program owner or the program user is used to run the program and control which objects can be used by the program (including the authority the program has for each object). This parameter is not updated if the program already exists. To change the value of USRPRF, delete the program and recompile using the correct value (or, if the constituent *MODULE objects exist, you may choose to invoke the CRTPGM command).

<u>*USER</u>
The program runs under the user profile of the program's user.

***OWNER**
The program runs under the user profile of both the program's user and owner. The collective set of object authority in both user profiles are used to find and access objects while the program is running. Any object created during the program are owned by the program's user.

AUT

Specifies the authority given to users who do not have specific authority to the object, who are not on the authorization list, and whose user group has no specific authority to the object. The authority can be altered for all or for specified users after the program is created with the CL commands Grant Object Authority (GRTOBJAUT) or Revoke Object Authority (RVKOBJAUT). For further information on these commands, see the *CL Reference.*

***LIBCRTAUT**

The public authority for the object is taken from the CRTAUT keyword of the target library (the library that contains the object). The value is determined when the object is created. If the CRTAUT value for the library changes after the create, the new value will not affect any existing objects.

***ALL**

Authority for all operations on the program object except those limited to the owner or controlled by authorization list management authority. The user can control the program object's existence, specify this security for it, change it, and perform basic functions on it, but cannot transfer its ownership.

***CHANGE**

Provides all data authority and the authority to perform all operations on the program object except those limited to the owner or controlled by object authority and object management authority. The user can change the object and perform basic functions on it.

***USE**

Provides object operational authority and read authority; authority for basic operations on the program object. The user is prevented from changing the object.

***EXCLUDE**

The user is prevented from accessing the object.

authorization-list name

Enter the name of an authorization list of users and authorities to which the program is added. The program object will be secured by this authorization list, and the public authority for the program object will be set to *AUTL. The authorization list must exist on the system when the CRTBNDRPG command is issued.

Note: Use the AUT parameter to reflect the security requirements of your system. The security facilities available are described in detail in the *Security Reference manual.*

TRUNCNBR

Specifies if the truncated value is moved to the result field or an error is generated when numeric overflow occurs while running the program.

***YES**
Ignore numeric overflow and move the truncated value to the result field.

***NO**
When numeric overflow is detected, a run time error is generated with error code RNX0103.

FIXNBR
Specifies whether zoned decimal data which are not valid are fixed by the compiler upon conversion to packed data.

***NONE**
Zoned decimal data will not be fixed by the compiler on the conversion to packed data and will result in decimal errors during runtime if used.

***ZONED**
Zoned decimal data which is not valid will be fixed by the compiler on the conversion to packed data. Blanks in numeric fields will be treated as zeroes. Each decimal digit will be checked for validity. If a decimal digit is not valid, it is replaced with zero. If a sign is not valid, the sign will be forced to a positive sign code of hex 'F'. If the sign is valid, it will be changed to either a positive sign hex 'F' or a negative sign hex 'D' as appropriate. If the resulting packed data is not valid, it will not be fixed.

TGTRLS
Specifies the release level of the OS/400 system on which you intend to run the object being generated.

***CURRENT**
The current release of the compiler is used to compile the source program and to generate an object to be run on the current release of the OS/400 system.

V3R1M0
The target release level is V3R1M0.

ALWNULL
Specifies whether an ILE RPG program will accept records with null-capable fields in an externally-described input-only file. A severity 0, compile-time message is issued when a record format contains null-capable fields.

***NO**
Specifies that the ILE RPG program will not process records with null-value fields from externally-described files. If you attempt to retrieve a record containing null values, no data in the record is accessible to the ILE RPG program and a data-mapping error occurs.

***YES**
Specifies that an ILE RPG program will accept records with null-value fields for an externally-described input-only file. When a record containing null

values is retrieved from an externally-described input-only file, no data mapping errors occur and the database default values are placed into any fields which contain null values.

BNDDIR
Specifies the list of binding directories that are used in symbol resolution.

***NONE**
No binding directory is specified.

binding-directory-name
Specify the name of the binding directory used in symbol resolution.

The directory name can be qualified with one of the following library values:

***LIBL**
The system searches the library list to find the library where the binding directory is stored.

***CURLIB**
The current library for the job is searched. If no library is specified as the current library for the job, library QGPL is used.

***USRLIBL**
Only the libraries in the user portion of the job's library list are searched.

library-name
Specify the name of the library to be searched.

ACTGRP
Specifies the activation group this program is associated with when it is called.

QILE
When this program is called, it is activated into the named activation group QILE.

***NEW**
When this program is called, it is activated into a new activation group.

***CALLER**
When this program is called, it is activated into the caller's activation group.

activation-group-name
Specify the name of the activation group to be used when this program is called.

CRTRPGMOD Command

The Create RPG Module (CRTRPGMOD) command compiles ILE RPG/400 source code to create a module object (*MODULE). The entire syntax diagram for the CRTRPGMOD command is shown below.

Job: B,I Pgm: B,I REXX: B,I Exec

```
►►──CRTRPGMOD──┬──────────────────────────────────────────────────┬──────────────────────────────────────────────►
               └─MODULE(──┬─*CURLIB/───────┬──┬─*CTLSPEC────┬──)─┘
                          └─library-name/──┘  └─module-name─┘

                                                                                                      (P)
►──┬───────────────────────────────────────────────────────────────────┬──┬──────────────────────────────────────────────┬──────►
   └─SRCFILE(──┬─*LIBL/─────────┬──┬─QRPGLESRC─────────┬──)─┘  └─SRCMBR(──┬─*MODULE──────────────────┬──)─┘
               ├─*CURLIB/───────┤  └─source-file-name──┘                  └─source-file-member-name──┘
               └─library-name/──┘

►──┬──────────────────────────────────────────┬──┬──────────────────────────────┬──┬─────────────────────────────────┬──►
   └─GENLVL(──┬─10───────────────────┬──)─┘  └─TEXT(──┬─*SRCMBRTXT────┬──)─┘  └─OPTION(──┤ OPTION Details ├──)─┘
              └─severity-level-value─┘                ├─*BLANK────────┤
                                                      └─'description'─┘

►──┬──────────────────────────┬──┬───────────────────────────┬──┬──────────────────────────────┬──────────────────────►
   └─DBGVIEW(──┬─*STMT───┬──)─┘  └─OUTPUT(──┬─*PRINT─┬──)─┘  └─OPTIMIZE(──┬─*NONE──┬──)─┘
               ├─*SOURCE─┤                  └─*NONE──┘                     ├─*BASIC─┤
               ├─*LIST───┤                                                 └─*FULL──┘
               ├─*COPY───┤
               ├─*ALL────┤
               └─*NONE───┘

►──┬──────────────────────────────────────┬──────────────────────────────────────────────────────────────────────────►
   └─INDENT(──┬─*NONE───────────┬──)─┘
              └─character-value─┘

►──┬────────────────────────────────────────────────────────────────────────────────────┬──────────────────────────►
   └─CVTOPT(──┬─*NONE─────────────────────────────────────────────────────────┬──)─┘
              └──┬───────────────┬──┬──────────────┬──┬──────────────┬──┬────────────────┬──┘
                 └─*DATETIME─┘  └─*GRAPHIC─┘  └─*VARCHAR─┘  └─*VARGRAPHIC─┘

►──┬──────────────────────────────────────────────────────────┬──┬───────────────────────────────────────────┬──────►
   └─SRTSEQ(──┬─*HEX────────────────────────────────┬──)─┘  └─LANGID(──┬─*JOBRUN──────────────┬──)─┘
              ├─*JOB────────────────────────────────┤                  ├─*JOB─────────────────┤
              ├─*JOBRUN─────────────────────────────┤                  └─language-identifier──┘
              ├─*LANGIDUNQ──────────────────────────┤
              ├─*LANGIDSHR──────────────────────────┤
              └──┬─────────────────┬─sort-table-name─┘
                 ├─*LIBL/──────────┤
                 ├─*CURLIB/────────┤
                 └─library-name/───┘

►──┬──────────────────────────┬──┬────────────────────────────────────────┬──┬─────────────────────────────┬────────►
   └─REPLACE(──┬─*YES─┬──)─┘  └─AUT(──┬─*LIBCRTAUT────────────────┬──)─┘  └─TRUNCNBR(──┬─*YES─┬──)─┘
               └─*NO──┘               ├─*ALL──────────────────────┤                      └─*NO──┘
                                      ├─*CHANGE───────────────────┤
                                      ├─*USE──────────────────────┤
                                      ├─*EXCLUDE──────────────────┤
                                      └─authorization-list-name───┘

►──┬────────────────────────────┬──┬─────────────────────────────┬──┬──────────────────────────┬─────────────────►◄
   └─FIXNBR(──┬─*NONE──┬──)─┘  └─TGTRLS(──┬─*CURRENT─┬──)─┘  └─ALWNULL(──┬─*NO──┬──)─┘
              └─*ZONED─┘                  └─V3R1M0───┘                    └─*YES─┘
```

Note:
P All parameters preceding this point can be specified by position.

OPTION Details:

```
├──┬─*XREF───┬──┬─*GEN───┬──┬─*NOSECLVL─┬──┬─*SHOWCPY───┬──┬─*EXPDDS───┬──┬─*EXT───┬──┬─*NOEVENTF─┬──────────────┤
   └─*NOXREF─┘  └─*NOGEN─┘  └─*SECLVL───┘  └─*NOSHOWCPY─┘  └─*NOEXPDDS─┘  └─*NOEXT─┘  └─*EVENTF───┘
```

Description of the CRTRPGMOD command

The parameters, options and variables for the CRTRPGMOD command are listed below. The same information is available online. Enter the command name on a command line, press PF4 (Prompt) and then press PF1 (Help) for any parameter you want information on.

MODULE
Specifies the library name and module name for the module object you are creating. The module name and library name must conform to AS/400 naming conventions. If no library is specified, the created module is stored in the current library.

<u>*CTLSPEC</u>
The name for the compiled module is taken from the name specified in the DFTNAME keyword of the control specification. If the module name is not specified on the control specification, and the source member is from a database file, the member name, specified by the SRCMBR parameter, is used as the module name. If the source is not from a database file then the module name defaults to RPGMOD.

module-name
Enter the name of the module object.

<u>*CURLIB</u>
The compiled module object is stored in the current library. If you have not specified a current library, QGPL is used.

library-name
Enter the name of the library where the compiled module object is to be stored.

SRCFILE
Specifies the name of the source file that contains the ILE RPG source member to be compiled and the library where the source file is stored. The recommended source physical file length is 112 characters: 12 for the sequence number and date, 80 for the code and 20 for the comments. This is the maximum amount of source that is shown on the compiler listing.

<u>QRPGLESRC</u>
The default source file QRPGLESRC contains the ILE RPG source member to be compiled.

source-file-name
Enter the name of the source file that contains the ILE RPG source member to be compiled.

<u>*LIBL</u>
The system searches the library list to find the library where the source file is stored. This is the default.

***CURLIB**
The current library is used to find the source file. If you have not specified a current library, QGPL is used.

library-name
Enter the name of the library where the source file is stored.

SRCMBR
Specifies the name of the member of the source file that contains the ILE RPG source specifications to be compiled.

***MODULE**
Use the name specified by the MODULE parameter as the source file member name. The compiled module object will have the same name as the source file member. If no module name is specified for the MODULE parameter, the command uses the first member created in or added to the source file as the source member name.

source-file-member-name
Enter the name of the member that contains the ILE RPG source specifications.

GENLVL
Specifies whether or not a module object is generated, depending on the severity of the errors encountered.

A severity-level value corresponding to the severity level of the messages produced during compilation can be specified with this parameter. The value must be between 0 and 20 inclusive. If the severity-level value specified is greater than or equal to the maximum severity of any errors encountered, then the module object is generated. However, the module produced may contain errors that cause unpredictable results when bound and run if the GENLVL option is set to a value greater than 10.

For errors greater than severity 20, the module object will not be generated.

10 A module object will not be generated if you have messages with a severity-level greater than 10.

severity-level-value
Enter a number, 0 through 20 inclusive. The generation severity level GENLVL controls the creation of the program object. The module object is created if all the compile messages have severity level below the generation severity level.

TEXT
Allows you to enter text that briefly describes the module and its function. The text appears whenever module information is displayed.

<u>*SRCMBRTXT</u>
The text of the source member is used.

***BLANK**
No text appears.

'description'
Enter the text that briefly describes the function of the source specifications. The text can be a maximum of 50 characters and must be enclosed in apostrophes. The apostrophes are not part of the 50-character string. Apostrophes are not required if you are entering the text on the prompt screen.

OPTION
Specifies the options to use when the source member is compiled. You can specify any or all of the options in any order. Separate the options with one or more blank spaces. If an option is specified more than once, the last one is used.

<u>*XREF</u>
Produces a cross-reference listing (when appropriate) for the source member.

***NOXREF**
Does not produce a cross-reference listing.

<u>*GEN</u>
Creates a module object that can be bound using the CRTPGM command if the highest severity level returned by the compiler does not exceed the severity specified in the GENLVL option.

***NOGEN**
A module object will not be created.

<u>*NOSECLVL</u>
Do not print second-level message text on the line following the first-level message text.

***SECLVL**
Print second-level message text on the line following the first-level message text in the Message Summary section.

<u>*SHOWCPY</u>
Show source records of members included by the /COPY compiler directive.

***NOSHOWCPY**
Do not show source records of members included by the /COPY compiler directive.

<u>*EXPDDS</u>
Show the expansion of externally-described files in the listing.

***NOEXPDDS**
Do not show the expansion of externally-described files in the listing.

***EXT**
Show the list of external procedures and fields referenced during the compile on the listing.

***NOEXT**
Do not show the list of external procedures and fields referenced during compilation on the listing.

***NOEVENTF**
Do not create an Event File for use by CoOperative Development Environment/400 (CODE/400). CODE/400 uses this file to provide error feedback integrated with the CODE/400 editor. An Event File is normally created when you create a module or program from within CODE/400.

***EVENTF**
Create an Event File for use by CoOperative Development Environment/400 (CODE/400). The Event File is created as a member in file EVFEVENT in the library where the created module or program object is to be stored. If the file EVFEVENT does not exist it is automatically created. The Event File member name is the same as the name of the object being created.

CODE/400 uses this file to provide error feedback integrated with the CODE/400 editor. An Event File is normally created when you create a module or program from within CODE/400.

DBGVIEW
Specifies which level of debugging is available for the compiled module object, and which source views are available for source-level debugging.

***STMT**
Allows the module object to be debugged using the Line Numbers shown on the left-most column of the source section of the compiler listing.

***SOURCE**
Generates the source view for debugging the compiled module object. This view is not available if the root source member is a DDM file. Also, if changes are made to any source members after the compile and before attempting to debug the module, the views for those source members may not be usable.

***LIST**
Generates the listing view for debugging the compiled module object. The information contained in the listing view is dependent on whether *SHOWCPY and *EXPDDS are specified for the OPTION parameter.

Note: The listing view will not show any indentation which you may have requested using the Indent option.

***COPY**
Generates the source and copy views for debugging the compiled module object. The source view for this option is the same source view generated for the *SOURCE option. The copy view is a debug view which has all the /COPY source members included. These views are not available if the root source member is a DDM file. Also, if changes are made to any source

members after the compile and before attempting to debug the module, the views for those source members may not be usable.

***ALL**
Generates the listing, source and copy views for debugging the compiled module object. The information contained in the listing view is dependent on whether *SHOWCPY and *EXPDDS are specified for the OPTION parameter.

***NONE**
Disables all of the debug options for debugging the compiled module object.

OUTPUT
Specifies if a compiler listing is generated.

<u>***PRINT**</u>
Produces a compiler listing, consisting of the ILE RPG/400 module source and all compile-time messages. The information contained in the listing is dependent on whether *XREF, *SECLVL, *SHOWCPY, *EXPDDS and *EXT are specified for the OPTION parameter.

***NONE**
Do not generate the compiler listing.

OPTIMIZE
Specifies the level of optimization, if any, of the module.

<u>***NONE**</u>
Generated code is not optimized. This is the fastest in terms of translation time. It allows variables to be displayed and modified while in debug mode.

***BASIC**
Some optimization is performed on the generated code. This allows user variables to be displayed but not modified while in debug mode.

***FULL**
Optimization which generates the most efficient code. Translation time is the longest. User variables may not be modified but may be displayed, although the presented values may not be current values.

INDENT
Specifies whether structured operations should be indented in the source listing for enhanced readability. Also specifies the characters that are used to mark the structured operation clauses.

Note: Any indentation which you request here will not be reflected in the listing debug view which is created when you specify DBGVIEW(*LIST).

<u>*NONE</u>
Structured operations will not be indented in the source listing.

character-value
The source listing is indented for structured operation clauses. Alignment of statements and clauses are marked using the characters you choose. You can choose any character string up to 2 characters in length. If you want to use a blank in your character string, you must enclose it in single quotation marks.

Note: The indentation may not appear as expected if there are errors in the module.

CVTOPT
Specifies how the ILE RPG compiler handles date, time, timestamp, graphic data types, and variable-length data types which are retrieved from externally-described database files.

<u>*NONE</u>
Ignores variable-length database data types and uses the native RPG date, time, timestamp and graphic data types.

***DATETIME**
Specifies that date, time, and timestamp database data types are to be declared as fixed-length character fields.

***GRAPHIC**
Specifies that double-byte character set (DBCS) graphic data types are to be declared as fixed-length character fields.

***VARCHAR**
Specifies that variable-length character data types are to be declared as fixed-length character fields.

***VARGRAPHIC**
Specifies that variable-length double-byte character set (DBCS) graphic data types are to be declared as fixed-length character fields.

SRTSEQ
Specifies the sort sequence table that is to be used in the ILE RPG source program.

<u>*HEX</u>
No sort sequence table is used.

***JOB**
Use the SRTSEQ value from the job when the module is created.

***JOBRUN**
Use the SRTSEQ value from the job when the module is run (after being bound).

***LANGIDUNQ**
Use a unique weighted table. This special value is used in conjunction with the LANGID parameter to select the proper sort sequence table.

***LANGIDSHR**
Use a shared weighted table. This special value is used in conjunction with the LANGID parameter to select the proper sort sequence table.

sort-table-name
Enter the name of the sort sequence table.

***LIBL**
The system searches the library list to find the library where the sort sequence table is stored.

***CURLIB**
The current library is used to find the sort sequence table. If you have not specified a current library, QGPL is used.

library-name
Enter the name of the library where the sort sequence table is stored.

LANGID
Specifies the language identifier to be used when the sort sequence is *LANGIDUNQ or *LANGIDSHR. The LANGID parameter is used in conjunction with the SRTSEQ parameter to select the sort sequence table.

<u>*JOBRUN</u>
Use the LANGID value associated with the job when the RPG module is run (after being bound).

***JOB**
Use the LANGID value associated with the job when the RPG module is created.

language-identifier
Use the language identifier specified. (For example, FRA for French and DEU for German).

REPLACE
Specifies whether a new module object is created if a module of the same name already exists in the specified library.

<u>*YES</u>
A new module object is created in the specified library. The existing module object of the same name in the specified library is moved to library QRPLOBJ.

***NO**
A new module object is not created if a module object of the same name already exists in the specified library.

AUT

Specifies the authority given to users who do not have specific authority to the object, who are not on the authorization list, and whose user group has no specific authority to the object. The authority can be altered for all or for specified users after the module is created with the CL commands Grant Object Authority (GRTOBJAUT) or Revoke Object Authority (RVKOBJAUT). For further information on these commands, see the *CL Reference.*

***LIBCRTAUT**

The public authority for the object is taken from the CRTAUT keyword of the target library (the library that contains the object). The value is determined when the object is created. If the CRTAUT value for the library changes after the create, the new value will not affect any existing objects.

***ALL**

Authority for all operations on the module object except those limited to the owner or controlled by authorization list management authority. The user can control the module object's existence, specify this security for it, change it, and perform basic functions on it, but cannot transfer its ownership.

***CHANGE**

Provides all data authority and the authority to perform all operations on the module object except those limited to the owner or controlled by object authority and object management authority. The user can change the object and perform basic functions on it.

***USE**

Provides object operational authority and read authority; authority for basic operations on the module object such as binding it into a program. The user is prevented from changing the object.

***EXCLUDE**

The user is prevented from accessing the object.

authorization-list name

Enter the name of an authorization list of users and authorities to which the module is added. The module object will be secured by this authorization list, and the public authority for the module object will be set to *AUTL. The authorization list must exist on the system when the CRTRPGMOD command is issued.

Note: Use the AUT parameter to reflect the security requirements of your system. The security facilities available are described in detail in the *Security Reference manual.*

TRUNCNBR

Specifies if the truncated value is moved to the result field or an error generated when numeric overflow occurs while running the program.

<u>*YES</u>
Ignore numeric overflow and move the truncated value to the result field.

***NO**
When numeric overflow is detected, a run time error is generated with error code RNX0103.

FIXNBR
Specifies whether zoned decimal data which are not valid are fixed by the compiler upon conversion to packed data.

<u>*NONE</u>
Zoned decimal data will not be fixed by the compiler on the conversion to packed data and will result in decimal errors during runtime if used.

***ZONED**
Zoned decimal data which is not valid will be fixed by the compiler on the conversion to packed data. Blanks in numeric fields will be treated as zeroes. Each decimal digit will be checked for validity. If a decimal digit is not valid, it is replaced with zero. If a sign is not valid, the sign will be forced to a positive sign code of hex 'F'. If the sign is valid, it will be changed to either a positive sign hex 'F' or a negative sign hex 'D' as appropriate. If the resulting packed data is not valid, it will not be fixed.

TGTRLS
Specifies the release level of the OS/400 system on which you intend to use the object being generated.

<u>*CURRENT</u>
The current release of the compiler is used to compile the source specifications and to generate an object to be used on the current release of the OS/400 system.

V3R1M0
The target release level is V3R1M0.

ALWNULL
Specifies whether an ILE RPG module will accept null values from null-capable fields in an externally described input-only file. A severity 0, compile-time message is issued when a record format contains null-capable fields.

<u>*NO</u>
Specifies that the ILE RPG module will not process null value fields from externally-described files. If you attempt to retrieve a record containing null values, no data in the record is accessible to the ILE RPG module and a data-mapping error occurs.

***YES**
Specifies that an ILE RPG module will accept null value fields for an externally-described input-only file. When a record containing null values is

retrieved from an externally-described input-only file, no data mapping errors occur and the database default values are placed into any fields which contain null values.

Appendix D. Compiler Listings

Compiler listings provide you with information regarding the correctness of your code with respect to the syntax and semantics of the RPG IV language. The listings are designed to help you to correct any errors through a source editor, as well as assist you while you are debugging a module. This section tells you how to interpret an ILE RPG/400 compiler listing. See “Using a Compiler Listing” on page 41 for information on how to use a listing.

To obtain a compiler listing specify OUTPUT(*PRINT) on either the CRTRPGMOD command or the CRTBNDRPG command. (This is their default setting.) The specification OUTPUT(*NONE) will suppress a listing.

Table 21 summarizes the keyword specifications and their associated compiler listing information.

Table 21 (Page 1 of 2). Sections of the Compiler Listing

Listing Section	**OPTION**1	**Description**
Prologue		Command option summary
Source listing		Source specifications
In-line diagnostic messages		Errors contained within one line of source
/COPY members	*SHOWCPY	/COPY member source records
Externally described files	*EXPDDS	Generated specifications
Matching field table		Lengths that are matched based on matching fields
Additional diagnostic messages		Errors spanning more than one line of source
Field Positions in Output Buffer		Start and end positions of programmed-described output fields
/COPY member table		List of /COPY members and their external names
Compile-time data		Compilation source records
Alternate collating sequences		ALTSEQ records and table or NLSS information and table
File translation		File translation records
Arrays		Array records
Tables		Table records
Key field information	*EXPDDS	Key field attributes
Cross reference	*XREF	File and record, and field and indicator references
External references	*EXT	List of external procedures and fields referenced during compilation
Message summary		List of messages and number of times they occurred
Second-level text	*SECLVL	Second-level text of messages
Final summary		Message and source record totals, and final compilation message
Code generation errors2		Errors (if any) which occur during code generation phase.

Table 21 (Page 2 of 2). Sections of the Compiler Listing

Listing Section	OPTION[1]	Description
Binding section[2]		Errors (if any) which occur during binding phase for CRTBNDRPG command

1 The OPTION column indicates what value to specify on the OPTION parameter to obtain this information. A blank entry means that the information will always appear if OUTPUT(*PRINT) is specified.

2 The sections containing the code generation errors and binding errors appear only if there are errors. There is no option to suppress these sections.

Reading a Compiler Listing

The following text contains a brief discussion and an example of each section of the compiler listing. The sections are presented in the order in which they appear in a listing.

Prologue

The prologue section summarizes the command parameters and their values as they were processed by the CL command analyzer. If *CURLIB or *LIBL was specified, the actual library name is listed. Also indicated in the prologue is the effect of overrides. Figure 170 illustrates how to interpret the Prologue section of the listing for the program MYSRC, which was compiled using the CRTBNDRPG command.

```
5763RG1 V3R1M0  940909RN        IBM ILE RPG/400        MYLIB/MYSRC  1         AS400S01    09/30/94 09:43:20       Page    1

  Command . . . . . . . . . . . . :   CRTBNDRPG
    Issued by . . . . . . . . . . :     MYUSERID

  Program . . . . . . . . . . . . :   MYSRC   2
    Library . . . . . . . . . . . :     MYLIB
  Text 'description' . . . . . . . :   *SRCMBRTXT

  Source Member . . . . . . . . . :   MYSRC   3
  Source File . . . . . . . . . . :   QRPGLESRC  4
    Library . . . . . . . . . . . :     MYLIB
    CCSID . . . . . . . . . . . . :     37
  Text 'description' . . . . . . . :
  Last Change . . . . . . . . . . :   09/30/94  11:54:44

  Generation severity level  . . . :   10
  Default activation group . . . . :   *YES
  Compiler options . . . . . . . . :   *XREF      *GEN       *NOSECLVL  *SHOWCPY      5
                                       *EXPDDS    *EXT       *NOEVENT
  Debugging views  . . . . . . . . :   *NONE
  Output . . . . . . . . . . . . . :   *PRINT
  Optimization level . . . . . . . :   *NONE
  Source listing indentation . . . :   '| '  6
  Type conversion options  . . . . :   *NONE
  Sort sequence  . . . . . . . . . :   *HEX
  Language identifier  . . . . . . :   *JOBRUN
  Replace program  . . . . . . . . :   *NO
  User profile . . . . . . . . . . :   *USER
  Authority  . . . . . . . . . . . :   *LIBCRTAUT
  Truncate numeric . . . . . . . . :   *YES
  Fix numeric  . . . . . . . . . . :   *NONE
  Target release . . . . . . . . . :   V3R1M0
  Allow null values  . . . . . . . :   *NO
```

Figure 170. Sample Prologue for CRTRPGMOD

1 Page Heading
The page heading information includes the product information line and the text supplied by a /TITLE directive. "Customizing a Compiler Listing" on page 42 describes how you can customize the page heading and spacing in a compiler listing.

2 Module or Program
The name of the created module object (if using CRTRPGMOD) or the name of the created program object (if using CRTBNDRPG)

3 Source member
The name of the source member from which the source records were retrieved (this can be different from 2 if you used command overrides).

4 Source
The name of the file actually used to supply the source records. If the file is overridden, the name of the overriding source is used.

5 Compiler options
The compiler options in effect at the time of compilation, as specified on either the CRTRPGMOD command or the CRTBNDRPG command.

6 Indentation Mark
The character used to mark structured operations in the source section of the listing.

Source Section

The source section shows records that comprise the ILE RPG/400 source specifications. The root source member records are always shown. If OPTION(*EXPDDS) is also specified, then the source section shows records generated from externally-described files, and marks them with a '=' in the column beside the line number. These records are not shown if *NOEXPDDS is specified. If OPTION(*SHOWCPY) is specified, then it also shows the records from /COPY members specified in the source, and marks them with a '+' in the column beside the line number. These records are not shown if *NOSHOWCPY is specified.

The source section identifies any syntax errors in the source, and includes a match-field table, when appropriate.

Note that the line numbers in the listing reflect the compiled source. For example, Program MYSRC has a /COPY statement in line 26. In the root source member, the next line is a DOWEQ operation. In the listing, however, the DOWEQ operation is on line 30. The three intervening lines shown in the listing are from the /COPY source member.

Figure 171 on page 362 shows the source section for MYSRC.

```
5763RG1 V3R1M0  940909RN        IBM ILE RPG/400        MYLIB/MYSRC            AS400S01     09/30/94 10:03:55        Page       2
1
Line   <---------------------- Source Specifications ----------------------------><---- Comments ----> Do  Page  Change  Src  Seq
Number ....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8....+....9....+...10 Num Line  Date    Id   Number
                             S o u r c e   L i s t i n g
     1 H ALTSEQ(*SRC)                                                                                             940909        000100

     2 FInFile    IF   E             DISK                                                                         940909        000200
        *---------------------------------------------------------------------------------------------* 2
        *                                RPG name         External name                                *
        * File name. . . . . . . . . :   INFILE           MYLIB/INFILE                                 *
        * Record format(s) . . . . . :   INREC            INREC                                        *
        *---------------------------------------------------------------------------------------------*
     3 FKEYL6     IF   E           K DISK                                                                         940909        000300
        *---------------------------------------------------------------------------------------------*
        *                                RPG name         External name                                *
        * File name. . . . . . . . . :   KEYL6            MYLIB/KEYL6                                  *
        * Record format(s) . . . . . :   REC1             REC1                                         *
        *                                REC2             REC2                                         *
        *---------------------------------------------------------------------------------------------*
     4 FOutFile   O    E             DISK                                                                         940909        000400
        *---------------------------------------------------------------------------------------------*
        *                                RPG name         External name                                *
        * File name. . . . . . . . . :   OUTFILE          MYLIB/OUTFILE                                *
        * Record format(s) . . . . . :   OUTREC           OUTREC                                       *
        *---------------------------------------------------------------------------------------------*

     5 D Blue            S             4    DIM(5) CTDATA PERRCD(1)                                               940908        000500
     6 D Green           S             2    DIM(5) ALT(Blue)                                                      940908        000600
     7 D Red             S             4    DIM(2) CTDATA PERRCD(1)                                               940908        000700
     8 D DSEXT1        E DS          100    PREFIX(BI_)                                                           940909        000800
        *---------------------------------------------------------------------------------------------* 3                1
        * Data structure . . . . . . :  DSEXT1                                                        *                  1
        * Prefix . . . . . . . . . . :  BI_                                                           *                  1
        * External format  . . . . . :  REC1 : MYLIB/DSEXT1                                           *                  1
        * Format text  . . . . . . . :  BUFFER INFO                                                   *                  1
        *---------------------------------------------------------------------------------------------*                  1
                                                                                               4
     9=D BI_FLD1                      5A    EXTFLD (FLD1)                          FORMAT                                   1      1
    10=D BI_FLD2                     10A    EXTFLD (FLD2)                          MAPPING                                  1      2
    11=D BI_FLD3                     18A    EXTFLD (FLD3)                          TABLE ENTRY                              1      3

    12=IINREC                                                                                                               2      1
        *---------------------------------------------------------------------------------------------*                  2
        * RPG record format  . . . . :   INREC                                                         *                  2
        * External format  . . . . . :   INREC : MYLIB/INFILE                                          *                  2
        *---------------------------------------------------------------------------------------------*                  2
    13=I                              1    25  FLDA                                                                     2      2
    14=I                             26    90  FLDB                                                                     2      3
    15=IREC1                                                                                                            3      1
        *---------------------------------------------------------------------------------------------*                  3
        * RPG record format  . . . . :   REC1                                                          *                  3
        * External format  . . . . . :   REC1 : MYLIB/KEYL6                                            *                  3
        *---------------------------------------------------------------------------------------------*                  3
    16=I                      *ISO-D   1   10  FLD12                                                                    3      2
    17=I                              11   13  FLD13                                                                    3      3
    18=I                              14   17  FLD14                                                                    3      4
    19=I                              18   22  FLD15                                                                    3      5
    20=IREC2                                                                                                            4      1
        *---------------------------------------------------------------------------------------------*                  4
        * RPG record format  . . . . :   REC2                                                          *                  4
        * External format  . . . . . :   REC2 : MYLIB/KEYL6                                            *                  4
        *---------------------------------------------------------------------------------------------*                  4
    21=I                      *ISO-D   1   10  FLD22                                                                    4      2
    22=I                              11   13  FLD23                                                                    4      3
    23=I                              14   17  FLD24                                                                    4      4
    24=I                              18   22  FLD25                                                                    4      5
```

Figure 171 (Part 1 of 2). Sample Source Part of the Listing

```
Line   <---------------------- Source Specifications --------------------------------------------><---- Comments ----> Src  Seq
Number ....1....+....2....+<-------- 26 - 35 --------->....4....+....5....+....6....+....7....+....8....+....9....+...10 Id   Number
    25 C                   MOVE                     '123'        BI_FLD1                                                000900
    26  /COPY MYCPY                                                                                 940909              001000
        *---------------------------------------------------------------------------------------* 5
        * RPG member name  . . . . . :  MYCPY                                                   *              5
        * External name  . . . . . . :  MYLIB/QRPGLESRC(MYCPY)                                  *              5
        * Last change  . . . . . . . :  09/30/94  11:55:20                                      *              5
        * Text 'description' . . . . :                                                          *              5
        *---------------------------------------------------------------------------------------*
  6
    27+C      Blue(1)      DSPLY                                                                               5 000100
    28+C      Green(4)     DSPLY                                                                               5 000200
    29+C      Red(2)       DSPLY                                                                               5 000300
  7
    30 C      *in20        doweq                    *OFF                                                         001100
    31 C                   | READ                   InRec                      ----20                           001200
    32 C                   | if                     NOT *in20                                                    001300
    33 C      FLDA         | | DSPLY                                                                              001400
    34 C                   | endif                                                                              001500
    35 C                   enddo                                                                                001600
    36 C                   write                    outrec                                                       001700
                                                                               8
    37 C                   SETON                                               LR----                           001800

Line   <---------------------- Source Specifications ----------------------------><---- Comments ----> Do  Page  Change  Src  Seq
Number ....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8....+....9....+...10 Num Line  Date    Id   Number
    38=OOUTREC                                                                                                 6      1
        *---------------------------------------------------------------------------------------*              6
        * RPG record format  . . . . :  OUTREC                                                  *              6
        * External format  . . . . . :  OUTREC : MYLIB/OUTFILE                                  *              6
        *---------------------------------------------------------------------------------------*              6
    39=O                     FLDY              100  CHAR       00                                              6      2
    40=O                     FLDZ              132  CHAR       32                                              6      3
       * * * * *   E N D   O F   S O U R C E   * * * * *
```

Figure 171 (Part 2 of 2). Sample Source Part of the Listing

1 Source Heading

Line Number
Starts at 1 and increments by 1 for each source or generated record. Use this number when debugging using statement numbers.

Ruler Line
This line adjusts when indentation is specified.

Do Number
Identifies the level of the structured operations. This number will not appear if indentation is requested.

Page Line
Shows the first 5 columns of the source record.

Source Id
Identifies the source (either /COPY or DDS) of the record. For /COPY members, it can be used to obtain the external member name from the /COPY member table.

Sequence Number
Shows the SEU sequence number of the record from a member in a source physical file. Shows an incremental number for records from a member in a physical file or records generated from DDS.

2 File/Record Information
Identifies the externally-described file and the records it contains.

3 DDS Information
Identifies from which externally-described file the field information is extracted. Shows the prefix value, if specified. Shows the format record text if specified in the DDS.

4 Generated Specifications
Shows the specifications generated from the DDS, indicated by '=' beside the Line Number. Shows up to 50 characters of field text if it is specified in the DDS.

5 /COPY Member Information
Identifies which /COPY member is used. Shows the member text, if any. Shows the date and time of the last change to the member.

6 /COPY Member Records
Shows the records from the /COPY member, indicated by a '+' beside the Line Number.

7 Indentation
Shows how structured operations appear when you request that they be marked.

8 Indicator Usage
Shows position of unused indicators, when an indicator is used.

Additional Diagnostic Messages

The Additional Diagnostic Messages section lists compiler messages which indicate errors spanning more than one line. When possible, the messages indicate the line number of the source which is in error. Figure 172 shows an example.

```
          A d d i t i o n a l   D i a g n o s t i c   M e s s a g e s

 Msg id  Sv Number Message text
*RNF7086 00      2 RPG handles blocking for file INFILE. INFDS is updated only
                   when blocks of data are transferred.
*RNF7086 00      4 RPG handles blocking for file OUTFILE. INFDS is updated
                   only when blocks of data are transferred.
*RNF7066 00      0 Record-Format REC1 not used for input or output.
*RNF7066 00      0 Record-Format REC2 not used for input or output.

 * * * * *   E N D   O F   A D D I T I O N A L   D I A G N O S T I C   M E S S A G E S   * * * * *
```

Figure 172. Sample Additional Diagnostic Messages

Field Positions in Output Buffer

The Field Positions in Output Buffer table is included in the listing whenever the source contains programmed-described Output specifications. For each variable or literal that is output, the table contains the line number of output field specification and its start and end positions within the output buffer. Literals that are too long for the table are truncated and suffixed with '...' with no ending apostrophe (for example, 'Extremely-long-litera...'.

/COPY Member Table

The /COPY member table identifies any /COPY members specified in the source and lists their external names. You can find the name and location of a member using the Source ID number. The table is also useful as a record of what members are used by the module/program. Figure 173 shows an example.

```
                           / C o p y   M e m b e r s

Line   Src  RPG name   <-------- External name -------> CCSID  <- Last change ->
Number Id              Library    File       Member            Date     Time
    26    5 MYCPY      MYLIB      QRPGLESRC  MYCPY        37  09/30/94 11:55:20

         * * * * *   E N D   O F   / C O P Y   M E M B E R S   * * * * *
```

Figure 173. Sample /COPY Member Table

Compile-Time Data

The Compile-Time Data section includes information on ALTSEQ or NLSS tables, and on tables and arrays. In this example, there is an alternate collating sequence and two arrays, as shown in Figure 174 on page 366.

```
                   C o m p i l e   T i m e   D a t a
   27 **                                                                                        940609       001800
      *-----------------------------------------------------------------*
      * Alternate Collating Sequence Table Data:                        *
      *-----------------------------------------------------------------*
   31 ALTSEQ    1122ACAB4B7C36F83A657D73                                                      940609       001900

      *-----------------------------------------------------------------*
      * Alternate Collating Sequence Table:                             *
      * Number of characters with an altered sequence . . . . . . :  6 1 *
      *     0_ 1_ 2_ 3_ 4_ 5_ 6_ 7_ 8_ 9_ A_ B_ C_ D_ E_ F_        2     *
      * _0  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .   _0         *
      * _1  .  22 3.  .  .  .  .  .  .  .  .  .  .  .  .  .  _1         *
      * _2  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .   _2         *
      * _3  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .   _3         *
      * _4  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .   _4         *
      * _5  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .   _5         *
      * _6  .  .  .  F8 .  .  .  .  .  .  .  .  .  .  .  .   _6         *
      * _7  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .   _7         *
      * _8  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .   _8         *
      * _9  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .   _9         *
      * _A  .  .  .  65 .  .  .  .  .  .  .  .  .  .  .  .   _A         *
      * _B  .  .  .  .  7C .  .  .  .  .  .  .  .  .  .  .   _B         *
      * _C  .  .  .  .  .  .  .  .  .  .  AB .  .  .  .  .   _C         *
      * _D  .  .  .  .  .  .  .  73 .  .  .  .  .  .  .  .   _D         *
      * _E  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .   _E         *
      * _F  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .   _F         *
      *     0_ 1_ 2_ 3_ 4_ 5_ 6_ 7_ 8_ 9_ A_ B_ C_ D_ E_ F_              *
      *-----------------------------------------------------------------*

   32 **                                                                                        000000       002000
      *-----------------------------------------------------------------*
      * Array . . . : BLUE          Alternating Array  . . . . : GREEN   *
      *-----------------------------------------------------------------*
   33 1234ZZ                                                                                    000000       002100
   34 ABCDYY                                                                                    000000       002200
   35 5432XX                                                                                    000000       002300
   36 EDCBWW                                                                                    000000       002400
*RNF8042 00      37 Not enough source records provided to load array or table.
   37 **                                                                                        940608       002500
      *-----------------------------------------------------------------*
      * Array . . . : RED                                               *
      *-----------------------------------------------------------------*
   38 3861                                                                                      940608       002600
   39 TKJL                                                                                      940608       002700

   * * * * *   E N D   O F   C O M P I L E   T I M E   D A T A   * * * * *
```

Figure 174. Sample Compile-Time Data Section

1 Total Number of Characters Altered
Shows the number of characters whose sort sequence has been altered.

2 Character to be Altered
The rows and columns of the table together identify the characters to be altered.

3 Alternate Sequence
The new hexadecimal sort value of the selected character.

4 Array/Table information
Identifies the name of the array or table for which the compiler is expecting data. The name of the alternate array is also shown, if it is defined.

Key Field Information

The Key Field Information section shows information about key fields for each keyed file. It also shows information on any keys that are common to multiple records (that is, common keys). Figure 175 shows an example.

```
                 K e y   F i e l d   I n f o r m a t i o n

        File          Internal    External
          Record      field name  field name  Attributes
     2  KEYL6
          Common Keys:
                                              DATE *ISO- 10
                                              CHAR       3
          REC1
                      FLD12                   DATE *ISO- 10
                      FLD13                   CHAR       3
                      FLD15                   CHAR       5
          REC2
                      FLD22                   DATE *ISO- 10
                      FLD23                   CHAR       3
                      FLD24                   CHAR       4

  * * * * *   E N D   O F   K E Y   F I E L D   I N F O R M A T I O N   * * * * *
```

Figure 175. Sample Key Field Information

Cross-Reference Table

The Cross-Reference table contains three lists:

- files and records
- fields
- indicators

Use it to check where files, fields and indicators are used within the module/program.

Note that the informational messages RNF7031, which are issued when an identifier is not referenced, will only appear in the cross-reference section of the listing and in the message summary. They do not appear in the source section of the listing. Figure 176 on page 368 shows an example.

```
                       C r o s s   R e f e r e n c e
      File and Record References:

         File            Device            References (D=Defined)
           Record
         INFILE          DISK                 2D
           INREC                             12       31
         KEYL6           DISK                 3D
           REC1                              15
           REC2                              20
         OUTFILE         DISK                 4D
           OUTREC                            36       38


      Field References:

         Field           Attributes        References (D=Defined M=Modified)
         *IN20           A(1)                30       32
         BI_FLD1         A(5)                25M
         BLUE      (5)   A(4)                 5D       6       27
*RNF7031 DSEXT1          S(100)               8D
         FLDA            A(25)               13D      33
*RNF7031 FLDB            A(65)               14D
         FLDY            A(100)              39
         FLDZ            A(32)               40
*RNF7031 FLD12           D(10*ISO-)          16D
*RNF7031 FLD13           A(3)                17D
*RNF7031 FLD14           A(4)                18D
*RNF7031 FLD15           A(5)                19D
*RNF7031 FLD22           D(10*ISO-)          21D
*RNF7031 FLD23           A(3)                22D
*RNF7031 FLD24           A(4)                23D
*RNF7031 FLD25           A(5)                24D
         GREEN     (5)   A(2)                 6D      28
         RED       (2)   A(4)                 7D      29


      Indicator References:

         Indicator                         References (D=Defined M=Modified)
         20                                  31D
         LR                                  37D

      * * * * *   E N D   O F   C R O S S   R E F E R E N C E   * * * * *
```

Figure 176. Sample Cross-Reference Table

External References List

The External References section lists the external procedures and fields which are required from or available to other modules at bind time. This section is shown whenever the source contains statically-bound procedures, imported Fields, or exported fields.

The statically-bound procedures portion contains the procedure name, and the references to the name on a CALLB operation or %PADDR built-in function.

The imported fields and exported fields portions contain the field name, the dimension if it is an array, the field attribute and its definition reference.

Message Summary

The message summary contains totals by severity of the errors that occurred. If OPTION(*SECLVL) is specified, it also provides second-level message text. Figure 177 on page 369 shows an example.

```
                       M e s s a g e   S u m m a r y

 Msg id  Sv Number Message text
*RNF7031 00     12 The name or indicator is not referenced.
*RNF7066 00      2 Record-Format name of Externally-Described file is not
                   used.
*RNF8042 00      1 Not enough source records provided to load array or table.


      * * * * *   E N D   O F   M E S S A G E   S U M M A R Y   * * * * *
```

Figure 177. Sample Message Summary

Final Summary

The final summary section provides final message statistics and source statistics. It also specifies the status of the compilation. Figure 178 shows an example.

```
                       F i n a l   S u m m a r y

  Message Totals:

    Information  (00) . . . . . . . :       17
    Warning      (10) . . . . . . . :        0
    Error        (20) . . . . . . . :        0
    Severe Error (30+)  . . . . . . :        0
    ---------------------------------  -------
    Total . . . . . . . . . . . . . :       17


  Source Totals:

    Records . . . . . . . . . . . . :       50
    Specifications  . . . . . . . . :       39
    Data records  . . . . . . . . . :        7
    Comments  . . . . . . . . . . . :        0


       * * * * *   E N D   O F   F I N A L   S U M M A R Y   * * * * *

Program MYSRC placed in library MYLIB. 00 highest severity. Created on 94/06/09 at 17:44:55.

        * * * * *   E N D   O F   C O M P I L A T I O N * * * * *
```

Figure 178. Sample Final Summary

Code Generation and Binding Errors

Following the final summary section, you may find a section with code generation errors and/or binding errors.

The code generation error section will appear only if errors occur while the compiler is generating code for the module object. Generally, this section will not appear. The binding errors section will appear whenever there are messages arising during the binding phase of the CRTBNDRPG command. A common error is the failure to specify the location of *all* the external procedures and fields referenced in the source at the time the CRTBNDRPG command was issued.

Bibliography

For additional information about topics related to ILE RPG/400 programming on the AS/400 system, refer to the following IBM AS/400 publications:

- *ADTS/400: Application Development Manager/400 User's Guide*, SC09-1808, describes creating and managing projects defined for the Application Development Manager/400 feature as well as using the program to develop applications.
- *ADTS/400: Programming Development Manager*, SC09-1771, provides information about using the Application Development ToolSet/400 programming development manager (PDM) to work with lists of libraries, objects, members, and user-defined options to easily do such operations as copy, delete, and rename. Contains activities and reference material to help the user learn PDM. The most commonly used operations and function keys are explained in detail using examples.
- *ADTS/400: Source Entry Utility*, SC09-1774, provides information about using the Application Development ToolSet/400 source entry utility (SEU) to create and edit source members. The manual explains how to start and end an SEU session and how to use the many features of this full-screen text editor. The manual contains examples to help both new and experienced users accomplish various editing tasks, from the simplest line commands to using pre-defined prompts for high-level languages and data formats.
- *Application Display Programming*, SC41-3715, provides information about:
  - Using DDS to create and maintain displays for applications;
  - Creating and working with display files on the system;
  - Creating online help information;
  - Using UIM to define panels and dialogs for an application;
  - Using panel groups, records, or documents
- *Backup and Recovery – Advanced*, SC41-3305, provides information about setting up and managing the following:
  - Journaling, access path protection, and commitment control
  - User auxiliary storage pools (ASPs)
  - Disk protection (device parity, mirrored, and checksum)

  Provides performance information about backup media and save/restore operations. Also includes advanced backup and recovery topics, such as using save-while-active support, saving and restoring to a different release, and programming tips and techniques.
- *CL Programming*, SC41-3721, provides a wide-ranging discussion of AS/400 programming topics including a general discussion on objects and libraries, CL programming, controlling flow and communicating between programs, working with objects in CL programs, and creating CL programs. Other topics include predefined and impromptu messages and message handling, defining and creating user-defined commands and menus, application testing, including debug mode, breakpoints, traces, and display functions.
- *CL Reference*, SC41-3722, provides a description of the AS/400 control language (CL) and its OS/400 commands. (Non-OS/400 commands are described in the respective licensed program publications.) Also provides an overview of *all* the CL commands for the AS/400 system, and it describes the syntax rules needed to code them.
- *Communications Management*, SC41-3406, provides information about work management in a communications environment, communications status, tracing and diagnosing communications problems, error handling and recovery, performance, and specific line speed and subsystem storage information.
- *Data Management*, SC41-3710, provides information about using files in application programs. Includes information on the following topics:
  - Fundamental structure and concepts of data management support on the system
  - Overrides and file redirection (temporarily making changes to files when an application program is run)
  - Copying files by using system commands to copy data from one place to another
  - Tailoring a system using double-byte data
- *DB2/400 Database Programming*, SC41-3701, provides a detailed discussion of the AS/400 database organization, including information on how to create, describe, and update database files on the system. Also describes how to define files to the system using OS/400 data description specifications (DDS) keywords.
- *DB2/400 SQL Programming*, SC41-3611, provides information about how to use DB2/400 Query Manager and SQL Development kit licensed program. Shows how to access data in a database library and prepare, run, and test an application program that contains embedded SQL statements. Contains examples of SQL/400 statements and a description of the interactive SQL function.

Describes common concepts and rules for using SQL/400 statements in COBOL/400, ILE COBOL/400, PL/I, ILE C/400, FORTRAN/400, RPG/400, ILE RPG/400, and REXX.

- *DB2/400 SQL Reference*, SC41-3612, provides information about how to use Structured Query Language/400 DB2/400 statements and gives details about the proper use of the statements. Examples of statements include syntax diagrams, parameters, and definitions. A list of SQL limits and a description of the SQL communication area (SQLCA) and SQL descriptor area (SQLDA) are also provided.
- *DDS Reference*, SC41-3712, provides detailed descriptions for coding the data description specifications (DDS) for file that can be described externally. These files are physical, logical, display, print, and intersystem communication function (ICF) files.
- *Distributed Data Management*, SC41-3307, provides information about remote file processing. It describes how to define a remote file to OS/400 distributed data management (DDM), how to create a DDM file, what file utilities are supported through DDM, and the requirements of OS/400 DDM as related to other systems.
- *Experience RPG IV Multimedia Tutorial* is an interactive self-study program explaining the differences between RPG III and RPG IV and how to work within the new ILE environment. An accompanying workbook provides additional exercises and doubles as a reference upon completion of the tutorial. ILE RPG/400 code examples are shipped with the tutorial and run directly on the AS/400. Dial 1-800-IBM-CALL to order the tutorial.
- *GDDM Programming*, SC41-3717, provides information about using OS/400 graphical data display manager (GDDM) to write graphics application programs. Includes many example programs and information to help users understand how the product fits into data processing systems.
- *GDDM Reference*, SC41-3718, provides information about using OS/400 graphical data display manager (GDDM) to write graphics application programs. This manual provides detailed descriptions of all graphics routines available in GDDM. Also provides information about high-level language interfaces to GDDM.
- *ICF Programming*, SC41-3442, provides information needed to write application programs that use AS/400 communications and the OS/400 intersystem communications function (OS/400-ICF). Also contains information on data description specifications (DDS) keywords, system-supplied formats, return codes, file transfer support, and program examples.
- *IDDU Use*, SC41-3704, describes how to use the AS/400 interactive data definition utility (IDDU) to describe data dictionaries, files, and records to the system. Includes:
  - An introduction to computer file and data definition concepts
  - An introduction to the use of IDDU to describe the data used in queries and documents
  - Representative tasks related to creating, maintaining, and using data dictionaries, files, record formats, and fields
  - Advanced information about using IDDU to work with files created on other systems and information about error recovery and problem prevention.
- *ILE C/400 Programmer's Guide*, SC09-1820, provides information on how to develop applications using the ILE C/400 language. It includes information about creating, running and debugging programs. It also includes programming considerations for interlanguage program and procedure calls, locales, handling exceptions, database, externally described and device files. Some performance tips are also described. An appendix includes information on migrating source code from EPM C/400 or System C/400 to ILE C/400.
- *ILE COBOL/400 Programmer's Guide*, SC09-1522, provides information about how to write, compile, bind, run, debug, and maintain ILE COBOL/400 programs on the AS/400 system. It provides programming information on how to call other ILE COBOL/400 and non-ILE COBOL/400 programs, share data with other programs, use pointers, and handle exceptions. It also describes how to perform input/output operations on externally attached devices, database files, display files, and ICF files.
- *ILE Concepts*, SC41-3606, explains concepts and terminology pertaining to the Integrated Language Environment (ILE) architecture of the OS/400 licensed program. Topics covered include creating modules, binding, running programs, debugging programs, and handling exceptions.
- *ILE RPG/400 Reference*, SC09-1526, provides information about the ILE RPG/400 programming language. This manual describes, position by position and keyword by keyword, the valid entries for all RPG IV specifications, and provides a detailed description of all the operation codes and built-in functions. This manual also contains information on the RPG logic cycle, arrays and tables, editing functions, and indicators.
- *ILE RPG Reference Summary*, SX09-1261, provides information about the RPG III and RPG IV programming language. This manual contains tables and lists for all specifications and operations in both languages. A key is provided to map RPG

III specifications and operations to RPG IV specifications and operations.

- *Printer Device Programming*, SC41-3713, provides information to help you understand and control printing. Provides specific information on printing elements and concepts of the AS/400 system, printer file and print spooling support for printing operations, and printer connectivity.
  Includes considerations for using personal computers, other printing functions such as Business Graphics Utility (BGU), advanced function printing (AFP), and examples of working with the AS/400 system printing elements such as how to move spooled output files from one output queue to a different output queue. Also includes an appendix of control language (CL) commands used to manage printing workload.
  Fonts available for use with the AS/400 system are also provided. Font substitution tables provide a cross-reference of substituted fonts if attached printers do not support application-specified fonts.
- *Security – Basic*, SC41-3301, explains why security is necessary, defines major concepts, and provides information on planning, implementing, and monitoring basic security on the AS/400 system.
- *Security – Reference*, SC41-3302, tells how system security support can be used to protect the system and the data from being used by people who do not have the proper authorization, protect the data from intentional or unintentional damage or destruction, keep security information up-to-date, and set up security on the system.
- *Software Installation*, SC41-3120, provides step-by-step procedures for initial installation, installing licensed programs, program temporary fixes (PTFs), and secondary languages from IBM.
  This manual is also for users who already have an AS/400 system with an installed release and want to install a new release.
- *System API Reference*, SC41-3801, provides information for the experienced programmer on how to use the application programming interfaces (APIs) to such OS/400 functions as:
  - Dynamic Screen Manager
  - Files (database, spooled, hierarchical)
  - Message handling
  - National language support
  - Network management
  - Objects
  - Problem management
  - Registration facility
  - Security
  - Software products
  - Source debug
  - UNIX-type
  - User-defined communications
  - User interface
  - Work management

  Includes original program model (OPM), Integrated Language Environment (ILE), and UNIX-type APIs.
- *System Operation*, SC41-3203, provides information about handling messages, working with jobs and printer output, devices communications, working with support functions, cleaning up your system, and so on.
- *System Startup and Problem Handling*, SC41-3206, provides information about the system unit control panel, starting and stopping the system, using tapes and diskettes, working with program temporary fixes, as well as handling problems.
- *Tape and Diskette Device Programming*, SC41-3716, provides information to help users develop and support programs that use tape and diskette drives for I/O. Includes information on device files and descriptions for tape and diskette devices.

Index

Communicating Your Comments to IBM

AS/400
ILE RPG/400
Programmer's Guide
Version 3

Publication No. SC09-1525-00

If there is something you like—or dislike—about this book, please let us know. You can use one of the methods listed below to send your comments to IBM. If you want a reply, include your name, address, and telephone number. If you are communicating electronically, include the book title, publication number, page number, or topic you are commenting on.

The comments you send should only pertain to the information in this book and its presentation. To request additional publications or to ask questions or make comments about the functions of IBM products or systems, you should talk to your IBM representative or to your IBM authorized remarketer.

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

If you are mailing a readers' comment form (RCF) from a country other than the United States, you can give it to the local IBM branch office or IBM representative for postage-paid mailing.

- If you prefer to send comments by mail, use the RCF at the back of this book.
- If you prefer to send comments by FAX, use this number:
  - United States and Canada: 416-448-6161
  - Other countries: (+1)-416-448-6161
- If you prefer to send comments electronically, use the network ID listed below. Be sure to include your entire network address if you wish a reply.
  - Internet: torrcf@vnet.ibm.com
  - IBMLink: toribm(torrcf)
  - IBM/PROFS: torolab4(torrcf)
  - IBMMAIL: ibmmail(caibmwt9)

Readers' Comments — We'd Like to Hear from You

AS/400
ILE RPG/400
Programmer's Guide
Version 3

Publication No. SC09-1525-00

Overall, how satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Overall satisfaction	☐	☐	☐	☐	☐

How satisfied are you that the information in this book is:

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Accurate	☐	☐	☐	☐	☐
Complete	☐	☐	☐	☐	☐
Easy to find	☐	☐	☐	☐	☐
Easy to understand	☐	☐	☐	☐	☐
Well organized	☐	☐	☐	☐	☐
Applicable to your tasks	☐	☐	☐	☐	☐

Please tell us how we can improve this book:

Thank you for your responses. May we contact you? ☐ Yes ☐ No

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

Name	Address
Company or Organization	
Phone No.	

Readers' Comments — We'd Like to Hear from You
SC09-1525-00

Cut or
Along

Fold and Tape **Please do not staple** Fold and Tape

PLACE
POSTAGE
STAMP
HERE

IBM Canada Ltd. Laboratory
Information Development
2G/345/1150/TOR
1150 EGLINTON AVENUE EAST
NORTH YORK ONTARIO CANADA M3C 1H7

Fold and Tape **Please do not staple** Fold and Tape

SC09-1525-00

Cut or
Along

Readers' Comments — We'd Like to Hear from You

AS/400
ILE RPG/400
Programmer's Guide
Version 3

Publication No. SC09-1525-00

Overall, how satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Overall satisfaction	☐	☐	☐	☐	☐

How satisfied are you that the information in this book is:

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Accurate	☐	☐	☐	☐	☐
Complete	☐	☐	☐	☐	☐
Easy to find	☐	☐	☐	☐	☐
Easy to understand	☐	☐	☐	☐	☐
Well organized	☐	☐	☐	☐	☐
Applicable to your tasks	☐	☐	☐	☐	☐

Please tell us how we can improve this book:

Thank you for your responses. May we contact you? ☐ Yes ☐ No

Name	Address
Company or Organization	
Phone No.	

Readers' Comments — We'd Like to Hear from You
SC09-1525-00

IBM®

Cut o
Along

Fold and Tape **Please do not staple** Fold and Tape

PLACE
POSTAGE
STAMP
HERE

IBM Canada Ltd. Laboratory
Information Development
2G/345/1150/TOR
1150 EGLINTON AVENUE EAST
NORTH YORK ONTARIO CANADA M3C 1H7

Fold and Tape **Please do not staple** Fold and Tape

SC09-1525-00

Cut o
Along

Program Number: 5763-RG1

Printed in Denmark by
Scanprint a·s, Viby J.

SC09-1525-00