IBM i
Version 7.2

Programming
IBM Rational Development Studio for i
ILE RPG Programmer's Guide

SC09-2507-09

Note

Before using this information and the product it supports, read the information in "Notices" on page 485.

This edition applies to IBM® Rational® Development Studio for i (product number 5770-WDS) and to all subsequent releases and modifications until otherwise indicated in new editions. This version does not run on all reduced instruction set computer (RISC) models nor does it run on CISC models.

Contents

Part 1. ILE RPG Programmer's Guide....................1

Chapter 1. About ILE RPG Programmer's Guide.......... 3
- Who Should Use This Guide.......... 3
- Prerequisite and Related Information.......... 3
- How to Send Your Comments.......... 3

Chapter 2. What's New.......... 5
- What's New since 7.2?.......... 5
- What's New in 7.2?.......... 9
- What's New in 7.1?..........17
- What's New in 6.1?..........21
- What's New in V5R4?.......... 25
- What's New in V5R3?.......... 29
- What's New in V5R2?.......... 33
- What's New in V5R1?.......... 36
- What's New in V4R4?.......... 41
- What's New in V4R2?.......... 45
- What's New in V3R7?.......... 49
- What's New in V3R6/V3R2?..........52

Chapter 3. ILE RPG Introduction..........57
- Overview of the RPG IV Programming Language.......... 57
  - RPG IV Specifications.......... 57
  - Cycle Programming..........58
  - Indicators..........59
  - Operation Codes.......... 59
  - Example of an ILE RPG Program.......... 60
  - Using IBM i.......... 65
  - Rational Development Studio for i.......... 66
- RPG Programming in ILE..........66
  - Program Creation..........67
  - Program Management.......... 68
  - Program Call..........69
  - Source Debugging..........69
  - Bindable APIs.......... 69
  - Multithreaded Applications.......... 70
- Program Creation Strategies..........71
  - Strategy 1: OPM-Compatible Application.......... 71
  - Strategy 2: ILE Program Using CRTBNDRPG.......... 73
  - Strategy 3: ILE Application Using CRTRPGMOD..........75
  - A Strategy to Avoid.......... 78
- Creating an Application Using Multiple Procedures..........79
  - A Multiple Procedures Module — Overview.......... 79
  - Example of Module with Multiple Procedures.......... 82
  - Coding Considerations..........89
  - For Further Information.......... 91

Chapter 4. Creating and Running an ILE RPG Application..........93
- Using Source Files.......... 93
  - Using Source Physical Files.......... 93

Using IFS Source Files........ 98
Creating a Program with the CRTBNDRPG Command........ 101
Using the CRTBNDRPG Command........ 102
Using a Compiler Listing........ 107
Accessing the RETURNCODE Data Area........ 113
Creating a Program with the CRTRPGMOD and CRTPGM Commands........ 115
Creating a Module Object........ 115
Binding Modules into a Program........ 122
Using a Binder Listing........ 127
Changing a Module or Program........ 127
Creating a Service Program........ 129
Service Program Overview........ 129
Strategies for Creating Service Programs........ 130
Creating a Service Program Using CRTSRVPGM........ 131
Sample Service Program........ 132
Running a Program........ 138
Running a Program Using the CL CALL Command........ 139
Running a Program From a Menu-Driven Application........ 141
Running a Program Using a User-Created Command........ 142
Replying to Run-Time Inquiry Messages........ 143
Ending an ILE Program........ 143
Managing Activation Groups........ 144
Managing Dynamically-Allocated Storage........ 146
Calling Programs and Procedures........ 159
Program/Procedure Call Overview........ 160
Using a Prototyped Call........ 165
Passing Prototyped Parameters........ 167
Interlanguage Calling Considerations........ 180
Using the Fixed-Form Call Operations........ 180
Returning from a Called Program or Procedure........ 183
Using Bindable APIs........ 186
Calling a Graphics Routine........ 187
Calling Special Routines........ 187
Storage Model........ 187
Multithreading Considerations........ 188
RPG and the eBusiness World........ 204
RPG and XML........ 204
RPG and other data formats such as JSON........ 210
RPG and MQSeries........ 210
RPG and Java........ 211

Chapter 5. Debugging and Exception Handling........ 241
Debugging Programs........ 241
The ILE Source 242
Preparing a Program for Debugging........ 244
Starting the ILE Source 247
Adding/Removing Programs from a Debug Session........ 249
Viewing the Program Source........ 250
Setting and Removing Breakpoints........ 253
Setting and Removing Watch Conditions........ 262
Example of Setting a Watch Condition........ 265
Stepping Through the Program Object........ 266
Displaying Data and Expressions........ 271
Changing the Value of Fields........ 282
Displaying Attributes of a Field........ 284
Equating a Name with a Field, Expression, or Command........ 285
Source Debug National Language Support for ILE RPG........ 286
Sample Source for Debug Examples........ 286

Handling Exceptions.. 290
Exception Handling Overview... 291
Using Exception Handlers... 295
Using RPG-Specific Handlers... 298
ILE Condition Handlers...307
Using Cancel Handlers... 313
Problems when ILE CL Monitors for Notify and Status Messages... 316
Obtaining a Dump..317
Obtaining an ILE RPG Formatted Dump...317
Using the DUMP Operation Code.. 318
Example of a Formatted Dump... 318

Chapter 6. Working with Files and Devices...325
Defining Files..325
Associating Files with Input/Output Devices.. 325
Naming Files.. 327
Types of File Descriptions.. 327
Defining Externally Described Files... 329
Defining Program-Described Files... 334
Data Management Operations and ILE RPG I/O Operations...334
General File Considerations..335
Overriding and Redirecting File Input and Output... 336
File Locking... 337
Record Locking.. 338
Sharing an Open Data Path..339
Spooling.. 340
SRTSEQ/ALTSEQ in an RPG Program versus a DDS File..341
Accessing Database Files.. 341
Database Files.. 342
Using Externally Described Disk Files...342
Using Program-Described Disk Files...347
Methods for Processing Disk Files.. 350
Valid File Operations...361
Using Commitment Control..363
Unexpected Results Using Keyed Files..368
DDM Files.. 368
Accessing Externally Attached Devices... 369
Types of Device Files...369
Accessing Printer Devices.. 370
Accessing Tape Devices... 377
Accessing Display Devices.. 377
Using Sequential Files.. 378
Using SPECIAL Files... 379
Using WORKSTN Files...382
Intersystem Communications Function...382
Using Externally Described WORKSTN Files.. 382
Using Program-Described WORKSTN Files.. 388
Valid WORKSTN File Operations... 390
Multiple-Device Files...391
Example of an Interactive Application... 393
Database Physical File...393
Main Menu Inquiry..394
File Maintenance.. 397
Search by Zip Code...406
Search and Inquiry by Name...412

Chapter 7. Appendixes.. 421
Appendix A. Behavioral Differences Between OPM RPG/400 and ILE RPG for AS/400.................421

Compiling......421
Running......421
Debugging and Exception Handling......422
I/O......423
DBCS Data in Character Fields......425
Appendix B. Using the RPG III to RPG IV Conversion Aid......425
Conversion Overview......425
Converting Your Source......428
Example of Source Conversion......435
Analyzing Your Conversion......438
Resolving Conversion Problems......442
Appendix C. The Create Commands......448
Using CL Commands......448
CRTBNDRPG Command......449
CRTRPGMOD Command......464
Appendix D. Compiler Listings......466
Reading a Compiler Listing......468
Appendix E. Information for Preprocessor Providers......480

Chapter 8. Bibliography......483

Notices......485
Programming interface information......486
Trademarks......486
Terms and conditions......487

Index......489

Part 1. ILE RPG Programmer's Guide

This guide provides information that shows how to use the ILE RPG compiler (ILE RPG) in the Integrated Language Environment®. ILE RPG is an implementation of the RPG IV language on the IBM i with the IBM i (IBM i) operating system. Use this guide to create and run ILE applications from RPG IV source.

Note: There are several screen captures in this guide, which might contain obsolete references to iSeries and other terms from prior releases.

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

Chapter 1. About ILE RPG Programmer's Guide

Read this section for an overview of the guide.

Who Should Use This Guide

This guide is for programmers who are familiar with the RPG 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 compiler on IBM i.

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 *IBM Rational Development Studio for i: ILE RPG Reference, SC09-2508-08*.

Before using this guide, you should:

- Know how to use applicable IBM i 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 *ILE Concepts, SC41-5606-09*.

Prerequisite and Related Information

Use the IBM i Information Center as your starting point for looking up IBM i technical information. You can access the Information Center from the following Web site:

```
http://www.ibm.com/systems/i/infocenter/
```

The Information Center contains new and updated system information, such as software installation, Linux, WebSphere®, Java™, high availability, database, logical partitions, CL commands, and system application programming interfaces (APIs). In addition, it provides advisors and finders to assist in planning, troubleshooting, and configuring your system hardware and software.

The manuals that are most relevant to the ILE RPG compiler are listed in the Chapter 8, “Bibliography,” on page 483.

How to Send Your Comments

Your feedback is important in helping to provide the most accurate and high-quality information. IBM welcomes any comments about this book or any other IBM i documentation.

- If you prefer to send comments by mail, use the the following address:

  IBM Canada Ltd. Laboratory
  Information Development
  8200 Warden Avenue
  Markham, Ontario, Canada L6G 1C7

  If you are mailing a readers' comment form from a country other than the United States, you can give the form to the local IBM branch office or IBM representative for postage-paid mailing.
- If you prefer to send comments by fax , use 1–845–491–7727, attention: RCF Coordinator.
- If you prefer to send comments electronically, use one of these e-mail addresses:
  - Comments on books:

RCHCLERK@us.ibm.com

  – Comments on the IBM i Information Center:

    RCHINFOC@us.ibm.com

Be sure to include the following:

- The name of the book.
- The publication number of the book.
- The page number or topic to which your comment applies.

Chapter 2. What's New

New and changed features in each release of the ILE RPG compiler since V3R1

There have been several releases of RPG IV since the first V3R1 release. The following is a list of enhancements made for each release since V3R1 to the current release:

- “What's New since 7.2?” on page 5
- “What's New in 7.2?” on page 9
- “What's New in 7.1?” on page 17
- “What's New in 6.1?” on page 21
- “What's New in V5R4?” on page 25
- “What's New in V5R3?” on page 29
- “What's New in V5R2?” on page 33
- “What's New in V5R1?” on page 36
- “What's New in V4R4?” on page 41
- “What's New in V4R2?” on page 45
- “What's New in V3R7?” on page 49
- “What's New in V3R6/V3R2?” on page 52

You can use this section to link to and learn about new RPG IV functions.

Note: The information for this product is up-to-date with the 7.2 release of RPG IV. If you are using a previous release of the compiler, you will need to determine what functions are supported on your system. For example, if you are using a 6.1 system, the functions new to the 7.2 release will not be supported.

What's New since 7.2?

This section describes the enhancements made to ILE RPG after 7.2.

New operation code DATA-INTO

DATA-INTO reads data from a structured document, such as JSON, into an RPG variable. It requires a parser to parse the document. The DATA-INTO operation calls the parser, and the parser passes the information in the document back to the DATA-INTO operation, which places the information into the RPG variable.

```
DCL-DS product QUALIFIED;
   name VARCHAR(25);
   id CHAR(10);
   price PACKED(9 : 2);
END-DS product;

DATA-INTO product %DATA('ProductInfo.JSON' : 'doc=file')
                  %PARSER('MYLIB/MYJSONPARS');
```

Built-in function %PROC()

%PROC() returns the external name of the current procedure.

More places that complex qualified names can be used

Complex qualified names are now allowed in the following places:

- Built-in function %ELEM.
- Built-in function %SIZE.

- Operation code DEALLOC.
- Operation code RESET.

Support for PCML version 7.0
With PCML version 7.0, the restriction against varying-length arrays and subfields is removed. The value '7.0' is now supported for the QIBM_RPG_PCML_VERSION environment variable, and parameters *V6 and *V7 are supported for the Control specification PGMINFO keyword.

Nested data structure subfields
When a qualified data structure is defined using free-form syntax, a subfield can be directly defined as a nested data structure subfield. The *street* subfield in the following data structure is referred to as *product.manufacturer.address.street*.

```
DCL-DS product QUALIFIED;
   name VARCHAR(25);
   id CHAR(10);
   DCL-DS manufacturer;
      name VARCHAR(10);
      DCL-DS address;
         street VARCHAR(50);
         city VARCHAR(25);
      END-DS address;
      active IND;
   END-DS manufacturer;
   price PACKED(9 : 2);
END-DS product;
```

New built-in functions %MAX and %MIN
%MAX returns the maximum value of its operands. %MIN returns the minimum value of its operands. There must be at least two operands, but there is no upper limit on the number of operands when %MAX or %MIN are used in calculations. When %MAX or %MIN are used in Declaration statements, there must be exactly two numeric operands.

In the following example, after the assignment operations, *maxval* has the value 'Zorro' and *minval* has the value 'Batman'.

```
   DCL-S maxval VARCHAR(10);
   DCL-S minval VARCHAR(10);
   DCL-S name1 VARCHAR(20) INZ('Robin');
   DCL-S name2 VARCHAR(10) INZ('Batman');

   maxval = %MAX(name1 : name2 : 'Zorro');
   minval = %MIN(name1 : name2 : 'Zorro');
```

ALIGN(*FULL) to define the length of a data structure as a multiple of its alignment
When *FULL is specified for the ALIGN keyword, the length of the data structure is a multiple of the alignment size. Without *FULL, the length of the data structure is determined by highest end position of the subfields. Specify the *FULL parameter when defining an aligned data structure to be passed as a parameter to a function or API, or if you use %SIZE to determine the distance between the elements in an array of aligned data structures.

For example, the following two data structures have a float subfield followed by a character subfield with length 1. The alignment size for both data structures is 8. The size of data structure DS_ALIGN_FULL is 16, while the size of data structure DS_ALIGN is 9.

```
DCL-DS ds_align_full ALIGN(*FULL) QUALIFIED;
    sub1 FLOAT(8);
    sub2 CHAR(1);
END-DS;

DCL-DS ds_align ALIGN QUALIFIED;
    sub1 FLOAT(8);
    sub2 CHAR(1);
END-DS;
```

ON-EXIT section

The ON-EXIT section runs every time that a procedure ends, whether the procedure ends normally or abnormally.

In the following example, the procedure allocates heap storage. The code that follows the ON-EXIT operation is always run, so the heap storage is always deallocated. The ON-EXIT section is run whether the procedure ends suddenly due to an unhandled exception, or the procedure is canceled due to the job ending, or the procedure returns normally due to reaching the end of the main part of the procedure or due to reaching a RETURN operation.

```
dcl-proc myproc;
   p = %alloc(100);

   ...
on-exit;
   dealloc(n) p;
end-proc;
```

Support for fully free-form source

RPG source with the special directive **FREE in the first line contains only free-form code. The code can begin in column 1 and extend to the end of the line.

There is no practical limit on the length of a source line in fully free-form source.

Fixed-form code is not allowed in fully free-form source, but column-limited source that uses only columns 6-80 can be included by using the /COPY or /INCLUDE directive.

Extended ALIAS support for files

The ALIAS keyword can now be specified for any externally-described file.

If the ALIAS keyword is specified for a global file that is not qualified, the alternate names of the fields are available for use in the RPG program.

In the following example, the field *REQALC* in the file MYFILE has the alternate name *REQUIRED_ALLOCATION*. The ALIAS keyword indicates that the name for this field within the RPG program is REQUIRED_ALLOCATION.

```
     dcl-f myfile ALIAS;

     read myfile;
     if required_allocation <> 0
     and size > 0;
       ...
```

Relaxed rules for data structures for I/O operations

- An externally-described data structure or LIKEREC data structure defined with type *ALL can be used as the result data structure for any I/O operation.

```
dcl-f myfile usage(*input : *output : *update);
dcl-ds ds extname('MYFILE' : *ALL);

read myfile ds;
update myfmt ds;
write myfmt ds;
```

- When a data structure is defined for a record format of a DISK file using LIKEREC without the second parameter, and the output buffer layout is identical to the input buffer layout, the data structure can be used as the result data structure for any I/O operation.

```
dcl-f myfile usage(*input : *output : *update);
dcl-ds ds likerec(fmt);

read myfile ds;
update myfmt ds;
write myfmt ds;
```

PCML enhancements

- Specify the *DCLCASE parameter for the PGMINFO Control specification keyword to have the names in the program-interface information generated in the same case as the names are defined in the RPG source file.
- Specify PGMINFO(*YES) in the Procedure specification keywords for the procedure that should be included in the program-interface information when a module is being created, or specify PGMINFO(*NO) for the procedures that should not be included.

DCLOPT(*NOCHGDSLEN)

Specify DCLOPT(*NOCHGDSLEN) to prevent changing the length of a data structure on an Input, Output, or Calculation specification. Specifying DCLOPT(*NOCHGDSLEN) allows %SIZE(*data-structure*) to be used in more free-form declarations.

New parameter TGTCCSID for CRTBNDRPG and CRTRPGMOD to support compiling from Unicode source

The ILE RPG compiler normally reads the source files for a compile in the CCSID of the primary source file. Since the compiler supports reading the source only in an EBCDIC CCSID, this means that the compile fails when the primary source file has a Unicode CCSID such as UTF-8 or UTF-16.

The TGTCCSID parameter for the CRTBNDRPG and CRTRPGMOD allows the RPG programmer to specify the CCSID with which the compiler reads the source files for the compile. Specify TGTCCSID(*JOB) or specify a specific EBCDIC CCSID such as TGTCCSID(37) or TGTCCSID(5035).

The default is TGTCCSID(*SRC).

See TGTCCSID parameter.

Table 1. Changed Language Elements Since 7.2: Control specification keywords

Element	Description
PGMINFO keyword	*DCLCASE parameter to generate the names in the program-interface information in the same case as the names are coded in the RPG source file.

Table 2. Changed Language Elements Since 7.2: File specification keywords

Element	Description
ALIAS keyword	Allowed for all externally-described files

Table 3. Changed Language Elements Since 7.2: Definitions

Element	Description
DCL-DS operation code	DCL-DS can be nested within a qualified data structure.
ALIGN(*FULL)	The ALIGN keyword can have parameter *FULL causing the length of a data structure to be a multiple of its alignment.

Table 4. New Language Elements Since 7.2: Directives

Element	Description
Special directive **FREE	**FREE indicates that the source is fully free-form, with RPG code from column 1 to the end of the source line.

Table 5. New Language Elements Since 7.2: Control specification keywords

Element	Description
DCLOPT (*NOCHGDSLEN) keyword	Disallow changing the size of a data structure on an Input, Output, or Calculation specification.

Table 6. New Language Elements Since 7.2: Built-in functions

Element	Description
%DATA	Specifies the document to be parsed by the DATA-INTO operation code
%MAX	Returns the maximum value of its operands
%MIN	Returns the minimum value of its operands
%PARSER	Specifies the parser for the DATA-INTO operation code
%PROC	Returns the external name of the current procedure

Table 7. New Language Elements Since 7.2: Control specification keywords

Element	Description
DCLOPT (*NOCHGDSLEN) keyword	Disallow changing the size of a data structure on an Input, Output, or Calculation specification.

Table 8. New Language Elements Since 7.2: Operation codes

Element	Description
DATA-INTO	Import data from a structured document into an RPG variable.
ON-EXIT	Begin the ON-EXIT section.

What's New in 7.2?

This section describes the enhancements made to ILE RPG in 7.2.

Free-form Control, File, Definition, and Procedure statements

- Free-form Control statements begin with CTL-OPT and end with a semicolon.

```
CTL-OPT OPTION(*SRCSTMT : *NODEBUGIO)
        ALWNULL(*USRCTL);
```

- Free-form File definition statements begin with DCL-F and end with a semicolon.

  The following statements define three files

  1. An externally-described DISK file opened for input and update.
  2. An externally-described WORKSTN file opened for input and output.
  3. A program-described PRINTER file with record-length 132.

```
DCL-F custFile usage(*update) extfile(custFilename);
DCL-F screen workstn;
DCL-F qprint printer(132) oflind(qprintOflow);
```

- Free-form data definition statements begin with DCL-C, DCL-DS, DCL-PI, DCL-PR, or DCL-S, and end with a semicolon.

  The following statements define several items

  1. A named constant *MAX_ELEMS*.
  2. A standalone varying length character field *fullName*.
  3. A qualified data structure with an integer subfield *num* and a UCS-2 subfield *address*.
  4. A prototype for the procedure 'Qp0lRenameUnlink'.

```
DCL-C MAX_ELEMS 1000;
DCL-S fullName VARCHAR(50)
               INZ('Unknown name');
DCL-DS ds1 QUALIFIED;
  num INT(10);
  address UCS2(100);
END-DS;
DCL-PR Qp0lRenameUnlink INT(10) EXTPROC(*DCLCASE);
   oldName POINTER VALUE OPTIONS(*STRING);
   newName POINTER VALUE OPTIONS(*STRING);
END-PR;
```

- Free-form Procedure definition statements begin with DCL-PROC and end with a semicolon. The END-PROC statement is used to end a procedure.

  The following example shows a free-form subprocedure definition.

```
DCL-PROC getCurrentUserName EXPORT;
   DCL-PI *n CHAR(10) END-PI;
   DCL-S curUser CHAR(10) INZ(*USER);

   RETURN curUser;
END-PROC;
```

- The /FREE and /END-FREE directives are no longer required. The compiler will ignore them.
- Free-form statements and fixed-form statements may be intermixed.

```
         IF endDate < beginDate;
C                   GOTO      internalError
         ENDIF;
         duration = %DIFF(endDate : beginDate : *days);
         . . .
C     internalError TAG
```

CCSID support for alphanumeric data

- The default alphanumeric CCSID for the module can be set to many more CCSIDs including UTF-8 and hexadecimal.
- Alphanumeric data can be defined with a CCSID. Supported CCSIDs include

  - Single-byte and mixed-byte EBCDIC CCSIDs
  - Single-byte and mixed-byte ASCII CCSIDs
  - UTF-8
  - Hexadecimal

CCSID of external alphanumeric subfields
Use CCSID(*EXACT) for an externally-described data structure to indicate that the alphanumeric subfields should have the same CCSID as the fields in the file.

CCSID conversion is not performed for hexadecimal data
CCSID conversion is not allowed for implicit or explicit conversion of hexadecimal data.

Hexadecimal data includes

- Hexadecimal literals
- Alphanumeric and graphic data defined with CCSID(*HEX)
- Alphanumeric and graphic data in buffers for externally-described files when the DATA keyword is in effect for the file and the CCSID of the field in the file is 65535
- Alphanumeric and graphic data in externally-described data structures defined with CCSID(*EXACT) when the CCSID of the field in the file is 65535

Implicit conversion for concatenation
The compiler will perform implicit conversion between alphanumeric, graphic, and UCS-2 data for concatenation expressions.

Open database files without conversion to the job CCSID
Use Control keyword OPENOPT(*NOCVTDATA) or File keyword DATA(*NOCVT) to specify that a database file will be opened so that alphanumeric and graphic data will not be converted to and from the job CCSID for input and output operations.

Temporarily change the default CCSIDs, date format, or time format
Use the /SET and /RESTORE directives to set default values for date formats, time formats, and CCSIDs.

Control the length returned by %SUBDT
An optional third parameter for %SUBDT allows you to specify the number of digits in the result. For example, you can return the value of the years as a four-digit value: %SUBDT(MyDate:*YEARS:4).

Increased precision for timestamp data
Timestamp data can have between 0 and 12 fractional seconds.

Open Access files
An Open Access file is a file which has all its operations handled by a user-written program or procedure, rather than by the operating system. This program or procedure is called an "Open Access Handler" or simply a "handler". The HANDLER keyword specifies the handler.

New XML-INTO options

- XML namespaces are supported by the "ns" and "nsprefix" options.
- XML names with characters that are not supported by RPG for subfield names are supported by the "case=convert" option.

Support for CCSID conversions that cause a loss of data when a source character does not exist in the target character set
Control-specification keyword CCSIDCVT(*EXCP : *LIST).

- Use CCSIDCVT(*EXCP) to get an exception if a CCSID conversion loses data due to the source character not having a match in the target character set.
- Use CCSIDCVT(*LIST) to get a listing of every CCSID conversion in the module, with a diagnostic message indicating whether the conversion has the potential of losing data.

VALIDATE(*NODATETIME) to allow the RPG compiler to skip the validation step when working with date, time and timestamp data

Use Control-specification keyword VALIDATE(*NODATETIME) to allow the RPG compiler to treat date, time, and timestamp data as character data, without performing the checks for validity.

This may improve the performance of some date, time, and timestamp operations.

Warning: Skipping the validation step can lead to serious data corruption problems. You should only use this feature when you are certain that your date, time, and timestamp data is always valid.

Table 9. Changed Language Elements In 7.2: Control specification keywords

Element	**Description**
CCSID keyword	• CCSID(*EXACT) instructs the compiler to be aware of the CCSID of all alphanumeric data in the module. – Alphanumeric and graphic literals have the CCSID of the source file – Alphanumeric data is always considered to have a CCSID When CCSID(*EXACT) is not specified, the RPG compiler may make incorrect assumptions about CCSIDs of data in literals, variables, or the input and output buffers of database files. • CCSID(*CHAR:ccsid) supports *HEX, *JOBRUNMIX, *UTF8, ASCII CCSIDs, and EBCDIC CCSIDs. • CCSID(*GRAPH:ccsid) supports *HEX, *JOBRUN. • CCSID(*UCS2:ccsid) supports *UTF16.
DFTACTGRP keyword	DFTACTGRP(*NO) is assumed if there are any free-form Control specifications, and at least one of the ACTGRP, BNDDIR, or STGMDL keywords is used.
OPENOPT keyword	OPENOPT(*{NO}CVT) controls the default for the DATA keyword for database files. • OPENOPT(*CVTDATA) indicates that DATA(*CVT) should be assumed for DISK and SEQ files if the DATA keyword is not specified for the file. • OPENOPT(*NOCVTDATA) indicates that DATA(*NOCVT) should be assumed for DISK and SEQ files if the DATA keyword is not specified for the file. .

Table 10. Changed Language Elements In 7.2: Directives

Element	**Description**
/FREE and /END-FREE directives	These directives are no longer necessary to indicate the beginning and ending of free-form code. They are ignored by the compiler.

Table 11. Changed Language Elements In 7.2: Definition-specification keywords

Element	**Description**
CCSID keyword	• Supported for alphanumeric data • Supported for externally-described data structures to control the CCSID of alphanumeric subfields • The parameter can be *HEX and *JOBRUN for graphic data • The parameter can be *UTF16 for UCS-2 data.

Table 11. Changed Language Elements In 7.2: Definition-specification keywords (continued)

Element	Description
DTAARA keyword	In a free-form definition: • *VAR is not used. If the name is specified without quotes, it is assumed to be the name of a variable or named constant. • For a data structure, *AUTO is used to specify that it is a Data Area Data Structure. *USRCTL is used to specify that the data area can be manipulated using IN, OUT and UNLOCK operations.
EXTFLD keyword	In a free-form subfield definition • The parameter is optional • If the parameter is specified without quotes, it is assumed to be the name of a previously-defined named constant.
EXTNAME keyword	In a free-form data structure definition • If the file-name or format-name parameter is specified without quotes, it is assumed to be the name of a previously-defined named constant.
EXPORT and IMPORT keywords	In a free-form definition • *DCLCASE may be specified for the external name indicating that the external name is identical to the way the stand-alone field or data structure is specified, with the same mixed case letters.
EXTPROC keyword	In a free-form prototype definition or a procedure-interface definition • *DCLCASE may be specified for the external procedure or method name indicating that the external name is identical to the way the prototype or procedure interface is specified, with the same mixed case letters. • If the procedure interface name is specified as *N, the external name is taken from the DCL-PROC statement.
LIKE keyword	In a free-form definition, the LIKE keyword has an optional second parameter specifying the length adjustment.
LEN keyword	In a free-form definition, the LEN keyword is allowed only for a data structure definition. For other free-form definitions, the length is specified as part of the data-type keyword.
CLASS, DATFMT, PROCPTR, TIMFMT, and VARYING keywords	These keywords are not used in a free-form definition. The information specified by these keywords is specified as part of the related data-type keywords.
FROMFILE, PACKEVEN, and TOFILE keywords	These keywords are not allowed in a free-form definition.
OVERLAY keyword	The parameter cannot be the name of the data structure for a free-form subfield definition. The POS keyword is used instead.

Table 12. Changed Language Elements In 7.2: Literals

Element	Description
Timestamp literals	Timestamp literals can have between 0 and 12 fractional seconds.

Table 13. Changed Language Elements In 7.2: Order of statements

Element	Description
File and Definition statements	File and Definition statements can be intermixed.

Table 14. Changed Language Elements In 7.2: Built-in functions

Element	Description
%CHAR	When the operand is a timestamp, the length of the returned value depends on the number of bytes in the timestamp. If the format is *ISO0, the number of bytes can be between 14 and 26. If the format is *ISO, the number of bytes can be 19, or between 21 and 32.
%DEC	When the operand is a timestamp, the number of digits can be between 14 and 26, depending on the number of fractional seconds in the timestamp.
%DIFF	When the operand is a timestamp, an optional fourth parameter specifies the number of fractional seconds to return.
%SECONDS	When %SECONDS is used to add seconds to a timestamp, the parameter can have decimal positions specifying the number of fractional seconds.
%SUBDT	• An optional third parameter specifies the number of digits in the result. • If the first operand is a timestamp, and the second operand is *SECONDS, an optional fourth operand indicates the number of fractional seconds in the result.
%TIMESTAMP	• The first parameter can be a timestamp. • The first parameter can be *SYS. • If the first parameter is date, timestamp, or *SYS, a second optional parameter can be a value between 0 and 12 indicating the number of fractional seconds. • If the first parameter is character or numeric, a third optional parameter can be a value between 0 and 12 indicating the number of fractional seconds.

Table 15. Changed Language Elements In 7.2: Fixed form Definition Specification

Element	Description
Length entry	The length entry for a timestamp can be 19, or a value between 21 and 32.
Decimal-positions entry	The decimal-positions entry for a timestamp can be a value between 0 and 12.

Table 16. New Language Elements In 7.2: Directives

Element	Description
/SET directive	Temporarily set a new value for the following Control statement keywords: • CCSID(*CHAR:ccsid) • CCSID(*GRAPH:ccsid) • CCSID(*UCS2:ccsid) • DATFMT(format) • TIMFMT(format) These values are used to supply a default value for definition statements when the value is not explicitly provided on the definition.
/RESTORE directive	Restore the previous setting to the value it had before the most recent /SET directive that set the value.: • CCSID(*CHAR) • CCSID(*GRAPH) • CCSID(*UCS2) • DATFMT • TIMFMT

Table 17. New Language Elements In 7.2: Free-form statements

Element	Description
CTL-OPT	Begins a free-form Control statement
DCL-F	Begins a free-form File definition
DCL-C	Begins a free-form Named Constant definition
DCL-DS	Begins a free-form Data Structure definition
DCL-SUBF	Begins a free-form Subfield definition. Specifying "DCL-SUBF" is optional unless the subfield name is the same as an operation code allowed in free-form calculations.
END-DS	Ends a free-form Data Structure definition. If there are no subfields, it can be specified after the last keyword of the DCL-DS statement.
DCL-PI	Begins a free-form Procedure Interface definition
DCL-PR	Begins a free-form Prototype definition
DCL-PARM	Begins a free-form Parameter definition. Specifying "DCL-PARM" is optional unless the parameter name is the same as an operation code allowed in free-form calculations .
END-PI	Ends a free-form Procedure Interface definition. If there are no parameters, it can be specified after the last keyword of the DCL-PI statement.
END-PR	Ends a free-form Prototype definition. If there are no parameters, it can be specified after the last keyword of the DCL-PR statement.
DCL-S	Begins a free-form Standalone Field definition
DCL-PROC	Begins a free-form Procedure definition
END-PROC	Ends a free-form Procedure definition

Table 18. New Language Elements in 7.2: Control specification keywords

Element	Description
CCSIDCVT(*EXCP | *LIST)	Allows you to control how the compiler handles conversions between data with different CCSIDs.
VALIDATE (*NODATETIME)	Specifies whether Date, Time and Timestamp data must be validated before it is used.

Table 19. New Language Elements In 7.2: File definition keywords

Element	Description
DATA(*{NO}CVT)	Controls whether a file is opened so that database performs CCSID conversion to and from the job CCSID for alphanumeric and graphic fields.
HANDLER(*handler* {:communication-area})	Specifies that the file is an Open Access file.
DISK{(*EXT | *record-length*)}	Device keywords to specify the device type of a free-form File definition. • The default device type is DISK. • The default parameter for each device-type keyword is *EXT, indicating that it is an externally-described file.
PRINTER{(*EXT | *record-length*)}	
SEQ{(*EXT | *record-length*)}	
SPECIAL{(*EXT | *record-length*)}	
WORKSTN{(*EXT | *record-length*)}	
USAGE(*INPUT *OUTPUT *UPDATE *DELETE)	Specifies the usage of the file in a free-form file definition.
KEYED{(*CHAR : *key-length*)}	Indicates that the file is keyed in a free-form file definition.

Table 20. New Language Elements In 7.2: Free-form data-type keywords

Element	Description
CHAR(*length*)	Fixed-length alphanumeric data type
VARCHAR(*length* {:*prefix-size*})	Varying-length alphanumeric data type
GRAPH(*length*)	Fixed-length Graphic data type
VARGRAPH(*length* {:*prefix-size*})	Varying-length Graphic data type
UCS2(*length*)	Fixed-length UCS-2 data type
VARUCS2(*length* {:*prefix-size*})	Varying-length UCS-2 data type
IND	Indicator data type

Table 20. New Language Elements In 7.2: Free-form data-type keywords (continued)

Element	Description
INT(*digits*)	Integer data type
UNS(*digits*)	Unsigned integer data type
PACKED(*digits* {:*decimals*})	Packed decimal data type
ZONED(*digits* {:*decimals*})	Zoned decimal data type
BINDEC(*digits* {:*decimals*})	Binary decimal data type
FLOAT(*size*)	Float data type
DATE{(format)}	Date data type
TIME{(format)}	Time data type
TIMESTAMP {(fractional seconds)}	Timestamp data type
POINTER{(*PROC)}	Pointer data type. The optional parameter *PROC indicates that it is a procedure pointer.
OBJECT{(*JAVA : *class-name*)}	Object data type. The parameters are optional if it is defining the return type of a Java constructor.

Table 21. New Language Elements In 7.2: Free-form data definition keywords

Element	Description
EXT	Indicates that a data structure is externally described. This keyword is optional if the EXTNAME keyword is specified as the first keyword for a data structure definition.
POS(*subfield-start-position*)	Specifies the starting position of a subfield in the data structure.
PSDS	Specifies that the data structure is a Program Status Data Structure.

What's New in 7.1?

This section describes the enhancements made to ILE RPG in 7.1.

Sort and search data structure arrays

Data structure arrays can be sorted and searched using one of the subfields as a key.

```
      // Sort the custDs array by the amount_owing subfield
      SORTA custDs(*).amount_owing;

      // Search for an element in the custDs array where the
      // account_status subfield is "K"
      elem = %LOOKUP("K" : custDs(*).account_status);
```

Sort an array either ascending or descending

An array can be sorted ascending using SORTA(A) and descending using SORTA(D). The array cannot be a sequenced array (ASCEND or DESCEND keyword).

```
      // Sort the salary array in descending order
      SORTA(D) salary;
```

New built-in function %SCANRPL (scan and replace)

The %SCANRPL built-in function scans for all occurrences of a value within a string and replaces them with another value.

```
      // Replace NAME with 'Tom'
      string1 = 'See NAME. See NAME run. Run NAME run.';
      string2 = %ScanRpl('NAME' : 'Tom' : string1);
      // string2 = 'See Tom. See Tom run. Run Tom run.'
```

%LEN(varying : *MAX)

The %LEN builtin function can be used to obtain the maximum number of characters for a varying-length character, UCS-2 or Graphic field.

Use ALIAS names in externally-described data structures

Use the ALIAS keyword on a Definition specification to indicate that you want to use the alternate names for the subfields of externally-described data structures. Use the ALIAS keyword on a File specification to indicate that you want to use the alternate names for LIKEREC data structures defined from the records of the file.

```
      A          R CUSTREC
      A            CUSTNM        25A         ALIAS(CUSTOMER_NAME)
      A            CUSTAD        25A         ALIAS(CUSTOMER_ADDRESS)
      A            ID            10P 0

      D custDs        e ds                  ALIAS
      D                                     QUALIFIED EXTNAME(custFile)
       /free
        custDs.customer_name = 'John Smith';
        custDs.customer_address = '123 Mockingbird Lane';
        custDs.id = 12345;
```

Faster return values

A procedure defined with the RTNPARM keyword handles the return value as a hidden parameter. When a procedure is prototyped to return a very large value, especially a very large varying value, the performance for calling the procedure can be significantly improved by defining the procedure with the RTNPARM keyword.

```
      D getFileData     pr               a   varying len(1000000)
      D                                      rtnparm
      D   file                           a   const varying len(500)
      D data            S                a   varying len(1000)
       /free
         data = getFileData ('/home/mydir/myfile.txt');
```

%PARMNUM built-in function

The %PARMNUM(parameter_name) built-in function returns the ordinal number of the parameter within the parameter list. It is especially important to use this built-in function when a procedure is coded with the RTNPARM keyword.

```
      D                 pi
      D   name                      100a   const varying
      D   id                         10i 0 value
      D   errorInfo                        likeds(errs_t)
      D                                    options(*nopass)
       /free
        // Check if the "errorInfo" parameter was passed
        if %parms >= %parmnum(errorInfo);
```

Optional prototypes

If a program or procedure is not called by another RPG module, it is optional to specify the prototype. The prototype may be omitted for the following types of programs and procedures:

- A program that is only intended to be used as an exit program or as the command-processing program for a command
- A program that is only intended to be called from a different programming language
- A procedure that is not exported from the module
- A procedure that is exported from the module but only intended to be called from a different programming language

Pass any type of string parameter

Implicit conversion will be done for string parameters passed by value or by read-only reference. For example, a procedure can be prototyped to have a CONST UCS-2 parameter, and character expression can be passed as a parameter on a call to the procedure. This enables you to write a single procedure with the parameters and return value prototyped with the UCS-2 type. To call that procedure, you can pass any type of string parameter, and assign the return value to any type of string variable.

```
      // The makeTitle procedure upper-cases the value
      // and centers it within the provided length
      alphaTitle = makeTitle(alphaValue : 50);
      ucs2Title = makeTitle(ucs2Value : 50);
      dbcsTitle = makeTitle(dbcsValue : 50);
```

Two new options for XML-INTO

- The *datasubf* option allows you to name a subfield that will receive the text data for an XML element that also has attributes.
- The *countprefix* option reduces the need for you to specify the *allowmissing=yes* option. It specifies the prefix for the names of the additional subfields that receive the number of RPG array elements or non-array subfields set by the XML-INTO operation.

These options are also available through a PTF for 6.1.

Teraspace storage model

RPG modules and programs can be created to use the teraspace storage model or to inherit the storage model of their caller. With the teraspace storage model, the system limits regarding automatic storage are significantly higher than those for the single-level storage model. There are limits for the amount of automatic storage for a single procedure and for the total automatic storage of all the procedures on the call stack.

Use the storage model (STGMDL) parameter on the CRTRPGMOD or CRTBNDRPG command, or use the STGMDL keyword on the Control specification.

***TERASPACE**
The program or module uses the teraspace storage model.

***SNGLVL**
The program or module uses the single-level storage model.

***INHERIT**
The program or module inherits the storage model of its caller.

Change to the ACTGRP parameter of the CRTBNDRPG command and the ACTGRP keyword on the Control specification

The default value of the ACTGRP parameter and keyword is changed from QILE to *STGMDL.

ACTGRP(*STGMDL) specifies that the activation group depends on the storage model of the program. When the storage model is *TERASPACE, ACTGRP(*STGMDL) is the same as ACTGRP(QILETS). Otherwise, ACTGRP(*STGMDL) is the same as ACTGRP(QILE).

Note: The change to the ACTGRP parameter and keyword does not affect the default way the activation group is assigned to the program. The default value for the STGMDL parameter and keyword is *SNGLVL, so when the ACTGRP parameter or keyword is not specified, the activation group of the program will default to QILE as it did in prior releases.

Allocate teraspace storage

Use the ALLOC keyword on the Control specification to specify whether the RPG storage-management operations in the module will use teraspace storage or single-level storage. The maximum size of a teraspace storage allocation is significantly larger than the maximum size of a single-level storage allocation.

Encrypted listing debug view

When a module's listing debug view is encrypted, the listing view can only be viewed during a debug session when the person doing the debugging knows the encryption key. This enables you to send debuggable programs to your customers without enabling your customers to see your source code through the listing view. Use the DBGENCKEY parameter on the CRTRPGMOD, CRTBNDRPG, or CRTSQLRPGI command.

Table 22. Changed Language Elements Since 6.1

Language Unit	Element	Description
Control specification keywords	ACTGRP(*STGMDL)	*STGMDL is the new default for the ACTGRP keyword and command parameter. If the program uses the teraspace storage module, the activation group is QILETS. Otherwise it is QILE.
Built-in functions	%LEN(varying-field : *MAX)	Can now be used to obtain the maximum number of characters of a varying-length field.
Operation codes	SORTA(A | D)	The SORTA operation code now allows the A and D operation extenders indicating whether the array should be sorted ascending (A) or descending (D).

Table 23. New Language Elements Since 6.1

Language Unit	Element	Description
Control specification keywords	STGMDL(*INHERIT | *TERASPACE | *SNGLVL)	Controls the storage model of the module or program
	ALLOC(*STGMDL | *TERASPACE | *SNGLVL)	Controls the storage model for the storage-managent operations %ALLOC, %REALLOC, DEALLOC, ALLOC, REALLOC
File specification keywords	ALIAS	Use the alternate field names for the subfields of data structures defined with the LIKEREC keyword

Table 23. New Language Elements Since 6.1 (continued)

Language Unit	Element	Description
Definition specification keywords	ALIAS	Use the alternate field names for the subfields of the externally-described data structure
	RTNPARM	Specifies that the return value for the procedure should be handled as a hidden parameter
Built-in functions	%PARMNUM	Returns the ordinal number of the parameter in the parameter list
	%SCANRPL	Scans for all occurrences of a value within a string and replaces them with another value
XML-INTO options	datasubf	Name a subfield that will receive the text data for an XML element that also has attributes
	countprefix	Specifies the prefix for the names of the additional subfields that receive the number of RPG array elements or non-array subfields set by the XML-INTO operation

What's New in 6.1?

This section describes the enhancements made to ILE RPG in 6.1.

THREAD(*CONCURRENT)

When THREAD(*CONCURRENT) is specified on the Control specification of a module, it provides ability to run concurrently in multiple threads:

- Multiple threads can run in the module at the same time.
- By default, static variables will be defined so that each thread will have its own copy of the static variable.
- Individual variables can be defined to be shared by all threads using STATIC(*ALLTHREAD).
- Individual procedures can be serialized so that only one thread can run them at one time, by specifying SERIALIZE on the Procedure-Begin specification.

Ability to define a main procedure which does not use the RPG cycle

Using the MAIN keyword on the Control specification, a subprocedure can be identified as the program entry procedure. This allows an RPG application to be developed where none of the modules uses the RPG cycle.

Files defined in subprocedures

Files can be defined locally in subprocedures. I/O to local files can only be done with data structures; I and O specifications are not allowed in subprocedures, and the compiler does not generate I and O specifications for externally described files. By default, the storage associated with local files is automatic; the file is closed when the subprocedure returns. The STATIC keyword can be used to indicate that the storage associated with the file is static, so that all invocations of the subprocedure will use the same file, and if the file is open when the subprocedure returns, it will remain open for the next call to the subprocedure.

Qualified record formats

When a file is defined with the QUALIFIED keyword, the record formats must be qualified by the file name, MYFILE.MYFMT. Qualified files do not have I and O specifications generated by the compiler; I/O can only be done through data structures.

Files defined like other files

Using the LIKEFILE keyword, a file can be defined to use the same settings as another File specification, which is important when passing a file as a parameter. If the file is externally-described, the QUALIFIED keyword is implied. I/O to the new file can only be done through data structures.

Files passed as parameters

A prototyped parameter can be defined as a File parameter using the LIKEFILE keyword. Any file related through the same LIKEFILE definition may be passed as a parameter to the procedure. Within the called procedure or program, all supported operations can be done on the file; I/O can only be done through data structures.

EXTDESC keyword and EXTFILE(*EXTDESC)

The EXTDESC keyword identifies the file to be used by the compiler at compile time to obtain the external decription of the file; the filename is specified as a literal in one of the forms 'LIBNAME/FILENAME' or 'FILENAME'. This removes the need to provide a compile-time override for the file.

The EXTFILE keyword is enhanced to allow the special value *EXTDESC, indicating that the file specified by EXTDESC is also to be used at runtime.

EXTNAME to specify the library for the externally-described data structure

The EXTNAME keyword is enhanced to allow a literal to specify the library for the external file. EXTNAME('LIBNAME/FILENAME') or EXTNAME('FILENAME') are supported. This removes the need to provide a compile-time override for the file.

EXFMT allows a result data structure

The EXFMT operation is enhanced to allow a data structure to be specified in the result field. The data structure must be defined with usage type *ALL, either as an externally-described data structure for the record format (EXTNAME(file:fmt:*ALL), or using LIKEREC of the record format (LIKEREC(fmt:*ALL).

Larger limits for data structures, and character, UCS-2 and graphic variables

- Data structures can have a size up to 16,773,104.
- Character definitions can have a length up to 16,773,104. (The limit is 4 less for variable length character definitions.)
- UCS-2 definitions can have a length up to 8,386,552 UCS-2 characters. (The limit is 2 less for variable length UCS-2 definitions.)
- Graphic definitions can have a length up to 8,386,552 DBCS characters. (The limit is 2 less for variable length graphic definitions.)
- The VARYING keyword allows a parameter of either 2 or 4 indicating the number of bytes used to hold the length prefix.

%ADDR(varying : *DATA)

The %ADDR built-in function is enhanced to allow *DATA as the second parameter to obtain the address of the data part of a variable length field.

Larger limit for DIM and OCCURS

An array or multiple-occurrence data structure can have up to 16,773,104 elements, provided that the total size is not greater than 16,773,104.

Larger limits for character, UCS-2 and DBCS literals

- Character literals can now have a length up to 16380 characters.

- UCS-2 literals can now have a length up to 8190 UCS-2 characters.
- Graphic literals can now have a length up to 16379 DBCS characters.

TEMPLATE keyword for files and definitions

The TEMPLATE keyword can be coded for file and variable definitions to indicate that the name will only be used with the LIKEFILE, LIKE, or LIKEDS keyword to define other files or variables. Template definitions are useful when defining types for prototyped calls, since the compiler only uses them at compile time to help define other files and variables, and does not generate any code related to them.

Template data structures can have the INZ keyword coded for the data structure and its subfields, which will ease the use of INZ(*LIKEDS).

Relaxation of some UCS-2 rules

The compiler will perform some implicit conversion between character, UCS-2 and graphic values, making it unnecessary to code %CHAR, %UCS2 or %GRAPH in many cases. This enhancement is also available through PTFs for V5R3 and V5R4. Implicit conversion is now supported for

- Assignment using EVAL and EVALR.
- Comparison operations in expressions.
- Comparison using fixed form operations IFxx, DOUxx, DOWxx, WHxx, CASxx, CABxx, COMP.
- Note that implicit conversion was already supported for the conversion operations MOVE and MOVEL.

UCS-2 variables can now be initialized with character or graphic literals without using the %UCS2 built-in function.

Eliminate unused variables from the compiled object

New values *UNREF and *NOUNREF are added to the OPTION keyword for the CRTBNDRPG and CRTRPGMOD commands, and for the OPTION keyword on the Control specification. The default is *UNREF. *NOUNREF indicates that unreferenced variables should not be generated into the RPG module. This can reduce program size, and if imported variables are not referenced, it can reduce the time taken to bind a module to a program or service program.

PCML can now be stored in the module

Program Call Markup Language (PCML) can now be stored in the module as well as in a stream file. By using combinations of the PGMINFO command parameter and/or the new PGMINFO keyword for the Control specification, the RPG programmer can choose where the PCML information should go. If the PCML information is placed in the module, it can later be retrieved using the QBNRPII API. This enhancement is also available through PTFs for V5R4, but only through the Control specification keyword.

Table 24. Changed Language Elements Since V5R4

Language Unit	Element	Description
Control specification keywords	OPTION(*UNREF | *NOUNREF)	Specifies that unused variables should not be generated into the module.
	THREAD(*CONCURRENT)	New parameter *CONCURRENT allows running concurrently in multiple threads.
File specification keywords	EXTFILE(*EXTDESC)	Specifies that the value of the EXTDESC keyword is also to be used for the EXTFILE keyword.
Built-in functions	%ADDR(varying-field : *DATA)	Can now be used to obtain the address of the data portion of a varying-length variable.

Table 24. Changed Language Elements Since V5R4 (continued)

Language Unit	Element	Description
Definition specification keywords	DIM(16773104)	An array can have up to 16773104 elements.
	EXTNAME('LIB/FILE')	Allows a literal for the file name. The literal can include the library for the file.
	OCCURS(16773104)	A multiple-occurrence data structure can have up to 16773104 elements.
	VARYING{(2|4)}	Can now take a parameter indicating the number of bytes for the length prefix.
Definition specifications	Length entry	Can be up to 9999999 for Data Structures, and definitions of type A, C or G. (To define a longer item, the LEN keyword must be used.)
Input specifications	Length entry	Can be up to 99999 for alphanumeric fields, and up to 99998 for UCS-2 and Graphic fields.
Calculation specifications	Length entry	Can be up to 99999 for alphanumeric fields.
Operation codes	EXFMT format { result-ds }	Can have a data structure in the result entry.

Table 25. New Language Elements Since V5R4

Language Unit	Element	Description
Control specification keywords	MAIN(subprocedure-name)	Specifies the program-entry procedure for the program.
	PGMINFO(*NO | *PCML { : *MODULE })	Indicates whether Program Information is to be placed directly in the module.

Table 25. New Language Elements Since V5R4 (continued)

Language Unit	Element	Description
File specification keywords	STATIC	Indicates that a local file retains its program state across calls to a subprocedure.
	QUALIFIED	Indicates that the record format names of the file are qualified by the file name, FILE.FMT.
	LIKEFILE(filename)	Indicates that the file is defined the same as another file.
	TEMPLATE	Indicates that the file is only to be used for later LIKEFILE definitions.
	EXTDESC(constant-filename)	Specifies the external file used at compile time for the external definitions.
Definition specification keywords	STATIC(*ALLTHREAD)	Indicates that the same instance of the static variable is used by all threads running in the module.
	LIKEFILE(filename)	Indicates that the parameter is a file.
	TEMPLATE	Indicates that the definition is only to be used for LIKE or LIKEDS definitions.
	LEN(length)	Specifies the length of a data structure, or a definition of type A, C or G.
Procedure specification keywords	SERIALIZE	Indicates that the procedure can be run by only one thread at a time.

What's New in V5R4?

The following list describes the enhancements made to ILE RPG in V5R4:

New operation code EVAL-CORR

```
EVAL-CORR{(EH)} ds1 = ds2
```

New operation code EVAL-CORR assigns data and null-indicators from the subfields of the source data structure to the subfields of the target data structure. The subfields that are assigned are the subfields that have the same name and compatible data type in both data structures.

For example, if data structure DS1 has character subfields A, B, and C, and data structure DS2 has character subfields B, C, and D, statement EVAL-CORR DS1 = DS2; will assign data from subfields DS2.B and DS2.C to DS1.B and DS1.C. Null-capable subfields in the target data structure that are affected by the EVAL-CORR operation will also have their null-indicators assigned from the null-indicators of the source data structure's subfields, or set to *OFF, if the source subfield is not null-capable.

```
    //  DS1 subfields          DS2 subfields
    //    s1  character          s1 packed
    //    s2  character          s2 character
    //    s3  numeric
    //    s4  date               s4 date
    //                           s5 character
    EVAL-CORR  ds1 = ds2;
    // This EVAL-CORR operation is equivalent to the following EVAL operations
    //    EVAL  ds1.s2 = ds2.s2
    //    EVAL  ds1.s4 = ds2.s4
    // Other subfields either appear in only one data structure (S3 and S5)
    // or have incompatible types (S1).
```

EVAL-CORR makes it easier to use result data structures for I/O operations to externally-described files and record formats, allowing the automatic transfer of data between the data structures of different record formats, when the record formats have differences in layout or minor differences in the types of the subfields.

New prototyped parameter option OPTIONS(*NULLIND)

When OPTIONS(*NULLIND) is specified for a parameter, the null-byte map is passed with the parameter, giving the called procedure direct access to the null-byte map of the caller's parameter.

New builtin function %XML

```
%XML (xmldocument { : options } )
```

The %XML builtin function describes an XML document and specifies options to control how the document should be parsed. The ***xmldocument*** parameter can be a character or UCS-2 expression, and the value may be an XML document or the name of an IFS file containing an XML document. If the value of the ***xmldocument*** parameter has the name of a file, the "doc=file" option must be specified.

New builtin function %HANDLER

```
%HANDLER (handlingProcedure : communicationArea  )
```

%HANDLER is used to identify a procedure to handle an event or a series of events. %HANDLER does not return a value, and it can only be specified as the first operand of XML-SAX and XML-INTO.

The first operand, *handlingProcedure*, specifies the prototype of the handling procedure. The return value and parameters specified by the prototype must match the parameters required for the handling procedure; the requirements are determined by the operation that %HANDLER is specified for.

The second operand, *communicationArea,* specifies a variable to be passed as a parameter on every call to the handling procedure. The operand must be an exact match for the first prototyped parameter of the handling procedure, according to the same rules that are used for checking prototyped parameters passed by reference. The communication-area parameter can be any type, including arrays and data structures.

New operation code XML-SAX

```
XML-SAX{ (e) } %HANDLER(eventHandler : commArea ) %XML(xmldoc { : options } );
```

XML-SAX initiates a SAX parse for the XML document specified by the %XML builtin function. The XML-SAX operation begins by calling an XML parser which begins to parse the document. When the parser discovers an event such as finding the start of an element, finding an attribute name, finding the end of an element etc., the parser calls the ***eventHandler*** with parameters describing the event. The *commArea* operand is a variable that is passed as a parameter to the ***eventHandler*** providing a way for the XML-SAX operation code to communicate with the handling procedure. When the

eventHandler returns, the parser continues to parse until it finds the next event and calls the ***eventHandler*** again.

New operation code XML-INTO

```
XML-INTO{ (EH) } variable   %XML(xmlDoc { : options });
XML-INTO{ (EH) } %HANDLER(handler : commArea ) %XML(xmlDoc { : options });
```

XML-INTO reads the data from an XML document in one of two ways:

- directly into a variable
- gradually into an array parameter that it passes to the procedure specified by %HANDLER.

Various options may be specified to control the operation.

The first operand specifies the target of the parsed data. It can contain a variable name or the % HANDLER built-in function.

The second operand contains the %XML builtin function specifying the source of the XML document and any options to control how the document is parsed. It can contain XML data or it can contain the location of the XML data. The doc option is used to indicate what this operand specifies.

```
// Data structure "copyInfo" has two subfields, "from"
// and "to".  Each of these subfields has two subfields
// "name" and "lib".
// File cpyA.xml contains the following XML document
// <copyinfo>
//    <from><name>MASTFILE</name><lib>CUSTLIB</lib></from>
//    <to><name>MYFILE</name><lib>*LIBL</lib>
// <copyinfo>
xml-into copyInfo %XML('cpyA.xml' : 'doc=file');
// After the XML-INTO operation, the following
// copyInfo.from  .name = 'MASTFILE  ' .lib = 'CUSTLIB   '
// copyInfo.to    .name = 'MYFILE    ' .lib = '*LIBL     '
```

Use the PREFIX keyword to remove characters from the beginning of field names

```
PREFIX('' : number_of_characters)
```

When an empty character literal (two single quotes specified with no intervening characters) is specified as the first parameter of the PREFIX keyword for File and Definition specifications, the specified number of characters is removed from the field names. For example if a file has fields XRNAME, XRIDNUM, and XRAMOUNT, specifying `PREFIX('':2)`on the File specification will cause the internal field names to be NAME, IDNUM, and AMOUNT.

If you have two files whose subfields have the same names other than a file-specific prefix, you can use this feature to remove the prefix from the names of the subfields of externally-described data structures defined from those files. This would enable you to use EVAL-CORR to assign the same-named subfields from one data structure to the other. For example, if file FILE1 has a field F1NAME and file FILE2 has a field F2NAME, and `PREFIX('':2)` is specified for externally-described data structures DS1 for FILE1 and DS2 for FILE2, then the subfields F1NAME and F2NAME will both become NAME. An EVAL-CORR operation between data structures DS1 and DS2 will assign the NAME subfield.

New values for the DEBUG keyword

```
DEBUG { ( *INPUT  *DUMP  *XMLSAX *NO *YES ) }
```

The DEBUG keyword determines what debugging aids are generated into the module. *NO and *YES are existing values. *INPUT, *DUMP and *XMLSAX provide more granularity than *YES.

***INPUT**
Fields that are in Input specifications but are not used anywhere else in the module are read into the program fields during input operations.

***DUMP**
DUMP operations without the (A) extender are performed.

***XMLSAX**
An array of SAX event names is generated into the module to be used while debugging a SAX event handler.

***NO**
Indicates that no debugging aids are to be generated into the module. Specifying DEBUG(*NO) is the same as omitting the DEBUG keyword.

***YES**
This value is kept for compatibility purposes. Specifying DEBUG(*YES) is the same as specifying DEBUG without parameters, or DEBUG(*INPUT : *DUMP).

Syntax-checking for free-form calculations

In SEU, free-form statements are now checked for correct syntax.

Improved debugging support for null-capable subfields of a qualified data structure
When debugging qualified data structures with null-capable subfields, the null-indicators are now organized as a similar data structure with an indicator subfield for every null-capable subfield. The name of the data structure is _QRNU_NULL_*data_structure_name*, for example _QRNU_NULL_MYDS. If a subfield of the data structure is itself a data structure with null-capable subfields, the null-indicator data structure will similarly have a data structure subfield with indicator subfields. For example, if data structure DS1 has null-capable subfields DS1.FLD1, DS1.FLD2, and DS1.SUB.FLD3, you can display all the null-indicators in the entire data structure using the debug instruction.

```
===> EVAL _QRNU_NULL_DS
> EVAL _QRNU_NULL_DS1
  _QRNU_NULL_DS1.FLD1 = '1'
  _QRNU_NULL_DS1.FLD2 = '0'
  _QRNU_NULL_DS1.SUB.FLD3 = '1'
===> EVAL _QRNU_NULL_DS.FLD2
  _QRNU_NULL_DS1.FLD2 = '0'
===> EVAL _QRNU_NULL_DS.FLD2 = '1'
===> EVAL DSARR(1).FLD2
  DSARR(1).FLD2 = 'abcde'

===> EVAL _QRNU_NULL_DSARR(1).FLD2

  _QRNU_NULL_DSARR(1).FLD2 = '0'
```

Change to end-of-file behaviour with shared files

If a module performs a keyed sequential input operation to a shared file and it results in an EOF condition, and a different module sets the file cursor using a positioning operation such as SETLL, a subsequent sequential input operation by the first module may be successfully done. Before this change, the first RPG module ignored the fact that the other module had repositioned the shared file.

This change in behaviour is available with PTFs for releases V5R2M0 (SI13932) and V5R3M0 (SI14185).

Table 26. Changed Language Elements Since V5R3

Language Unit	Element	Description
Control specification keywords	DEBUG(*INPUT|*DUMP *XMLSAX|*NO|*YES)	New parameters *INPUT, *DUMP and *XMLSAX give more options for debugging aids.
File specification keywords	`PREFIX('':2)`	An empty literal may be specified as the first parameter of the PREFIX keyword, allowing characters to be removed from the beginning of names.

Table 26. Changed Language Elements Since V5R3 (continued)

Language Unit	Element	Description
Definition specification keywords	OPTIONS(*NULLIND)	Indicates that the null indicator is passed with the parameter.
	`PREFIX('':2)`	An empty literal may be specified as the first parameter of the PREFIX keyword, allowing characters to be removed from the beginning of names.

Table 27. New Language Elements Since V5R3

Language Unit	Element	Description
Built-in functions	%HANDLER(prototype: parameter)	Specifies a handling procedure for an event.
	%XML(document{:options})	Specifies an XML document and options to control the way it is parsed.
Operation codes	EVAL-CORR	Assigns data and null-indicators from the subfields of the source data structure to the subfields of the target data structure.
	XML-INTO	Reads the data from an XML document directly into a program variable.
	XML-SAX	Initiates a SAX parse of an XML document.

What's New in V5R3?

The following list describes the enhancements made to ILE RPG in V5R3:

- **New builtin function %SUBARR:**

  New builtin function %SUBARR allows assignment to a sub-array or returning a sub-array as a value.

  Along with the existing %LOOKUP builtin function, this enhancements enables the implementation of dynamically sized arrays with a varying number of elements.

  %SUBARR(array : start) specifies array elements array(start) to the end of the array

  %SUBARR(array : start : num) specifies array elements array(start) to array(start + num - 1)

  Example:

```
// Copy part of an array to another array:
  resultArr = %subarr(array1:start:num);
  // Copy part of an array to part of another array:
  %subarr(Array1:x:y) = %subarr(Array2:m:n);
  // Sort part of an array
  sorta %subarr(Array3:x:y);

  // Sum part of an array
  sum = %xfoot(%subarr(Array4:x:y));
```

- **The SORTA operation code is enhanced to allow sorting of partial arrays.**

When %SUBARR is specified in factor 2, the sort only affects the partial array indicated by the %SUBARR builtin function.

- **Direct conversion of date/time/timestamp to numeric, using %DEC:**

  %DEC is enhanced to allow the first parameter to be a date, time or timestamp, and the optional second parameter to specify the format of the resulting numeric value.

  Example:

```
D numDdMmYy       s              6p 0
  D date            s               d    datfmt(*jul)
     date = D'2003-08-21';
     numDdMmYy = %dec(date : *dmy);     // now numDdMmYy = 210803
```

- **Control specification CCSID(*CHAR : *JOBRUN) for correct conversion of character data at runtime:**

  The Control specification CCSID keyword is enhanced to allow a first parameter of *CHAR. When the first parameter is *CHAR, the second parameter must be *JOBRUN. CCSID(*CHAR : *JOBRUN) controls the way character data is converted to UCS-2 at runtime. When CCSID(*CHAR:*JOBRUN) is specified, character data will be assumed to be in the job CCSID; when CCSID(*CHAR : *JOBRUN) is not specified, character data will be assumed to be in the mixed-byte CCSID related to the job CCSID.

- **Second parameter for %TRIM, %TRIMR and %TRIML indicating what characters to trim:**

  %TRIM is enhanced to allow an optional second parameter giving the list of characters to be trimmed.

  Example:

```
trimchars = '*-.';
  data = '***a-b-c-.'
  result = %trim(data : trimchars);
  // now result = 'a-b-c'.  All * - and . were trimmed from the ends of the data
```

- **New prototype option OPTIONS(*TRIM) to pass a trimmed parameter:**

  When OPTIONS(*TRIM) is specified on a prototyped parameter, the data that is passed be trimmed of leading and trailing blanks. OPTIONS(*TRIM) is valid for character, UCS-2 and graphic parameters defined with CONST or VALUE. It is also valid for pointer parameters defined with OPTIONS(*STRING). With OPTIONS(*STRING : *TRIM), the passed data will be trimmed even if a pointer is passed on the call.

  Example:

```
D proc            pr
D   parm1                        5a    const options(*trim)
D   parm2                        5a    const options(*trim : *rightadj)
D   parm3                        5a    const varying options(*trim)
D   parm4                         *    value options(*string : *trim)
D   parm5                         *    value options(*string : *trim)
D ptr             s               *
D data            s             10a
D fld1            s              5a

   /free
         data = ' rst ' + x'00';
         ptr = %addr(data);

         proc (' xyz ' : ' @#$ ' : ' 123 ' : ' abc ' : ptr);
         // the called procedure receives the following parameters
         //    parm1 = 'xyz  '
         //    parm2 = '  @#$'
         //    parm3 = '123'
         //    parm4 = a pointer to 'abc.' (where . is x'00')
         //    parm5 = a pointer to 'rst.' (where . is x'00')
```

- **Support for 63 digit packed and zoned decimal values**

  Packed and zoned data can be defined with up to 63 digits and 63 decimal positions. The previous limit was 31 digits.

- **Relaxation of the rules for using a result data structure for I/O to externally-described files and record formats**
  - The result data structure for I/O to a record format may be an externally-described data structure.
  - A data structure may be specified in the result field for I/O to an externally-described file name for operation codes CHAIN, READ, READE, READP and READPE.

  Examples:

  1. The following program writes to a record format using from an externally-described data structure.

```
Foutfile    o    e           k disk
D outrecDs      e ds                  extname(outfile) prefix(O_)
/free
      O_FLD1 = 'ABCDE';
      O_FLD2 = 7;
      write outrec outrecDs;
      *inlr = *on;
/end-free
```

  2. The following program reads from a multi-format logical file into data structure INPUT which contains two overlapping subfields holding the fields of the respective record formats.

```
Flog       if   e           k disk    infds(infds)
D infds           ds
D   recname              261    270
D input           ds                  qualified
D   rec1                              likerec(rec1) overlay(input)
D   rec2                              likerec(rec2) overlay(input)
   /free
      read log input;
      dow not %eof(log);
         dsply recname;
         if recname = 'REC1';
         // handle rec1
         elseif recname = 'REC2';
            // handle rec2
         endif;
         read log input;
      enddo;
      *inlr = *on;
   /end-free
```

- If a program/module performs a keyed sequential input operation to a shared file and it results in an EOF condition, a subsequent sequential input operation by the same program/module may be attempted. An input request is sent data base and if a record is available for input, the data is moved into the program/module and the EOF condition is set off.
- **Support for new environment variables for use with RPG programs calling Java methods**
  - **QIBM_RPG_JAVA_PROPERTIES** allows RPG users to explicitly set the Java properties used to start the JVM

    This environment variable must be set before any RPG program calls a Java method in a job.

    This environment variable has contains Java options, separated and terminated by some character that does not appear in any of the option strings. Semicolon is usually a good choice.

    Examples:

    1. **Specifying only one option:** If the system's default JDK is 1.3, and you want your RPG programs to use JDK 1.4, set environment variable QIBM_RPG_JAVA_PROPERTIES to

```
 '-Djava.version=1.4;'
```

    Note that even with just one option, a terminating character is required. This example uses the semicolon.

2. **Specifying more than one option:** If you also want to set the os400.stdout option to a different value than the default, you could set the environment variable to the following value:

```
'-Djava.version=1.4!-Dos400.stdout=file:mystdout.txt!'
```

This example uses the exclamation mark as the separator/terminator. Note: This support is also available in V5R1 and V5R2 with PTFs. V5R1: SI10069, V5R2: SI10101.

– **QIBM_RPG_JAVA_EXCP_TRACE** allows RPG users to get the exception trace when an RPG call to a Java method ends with an exception

This environment variable can be set, changed, or removed at any time.

If this environment variable contains the value 'Y', then when a Java exception occurs during a Java method call from RPG, or a called Java method throws an exception to its caller, the Java trace for the exception will be printed. By default, it will be printed to the screen, and may not be possible to read. To get it printed to a file, set the Java option os400.stderr. (This would have to be done in a new job; it could be done by setting the QIBM_RPG_JAVA_PROPERTIES environment variable to

```
'-Dos400.stderr=file:stderr.txt;'
```

- **An RPG preprocessor enabling the SQL preprocessor to handle conditional compilation and nested /COPY**

  When the RPG compiler is called with a value other than *NONE for parameter PPGENOPT, it will behave as an RPG preprocessor. It will generate a new source file rather than generating a program. The new source file will contain the original source lines that are accepted by the conditional compilation directives such as /DEFINE and /IF. It will also have the source lines from files included by /COPY statements, and optionally it will have the source lines included by /INCLUDE statements. The new source file will have the comments from the original source file if PPGENOPT(*DFT) or PPGENOPT(*NORMVCOMMENT) is specified.

  When the SQL precompiler is called with a value other than *NONE for new parameter RPGPPOPT, the precompiler will use this RPG preprocessor to handle /COPY, the conditional compilation directives and possibly the /INCLUDE directive. This will allow SQLRPGLE source to have nested /COPY statements, and conditionally used statements.

Table 28. Changed Language Elements Since V5R2

Language Unit	Element	Description
Control specification keywords	CCSID(*GRAPH:parameter| *UCS2:number| *CHAR:*JOBRUN)	Can now take a first parameter of *CHAR, with a second parameter of *JOBRUN, to control how character data is treated at runtime.
Built-in Functions	%DEC(expression {format})	Can now take a parameter of type Date, Time or Timestamp
	%TRIM(expression:expression)	Can now take a second parameter indicating the set of characters to be trimmed
Definition Specification Keywords	OPTIONS(*TRIM)	Indicates that blanks are to be trimmed from passed parameters
Definition Specifications	Length and decimal place entries	The length and number of decimal places can be 63 for packed and zoned fields.

Table 28. Changed Language Elements Since V5R2 (continued)

Language Unit	Element	Description
Input specifications	Length entry	The length can be 32 for packed fields and 63 for zoned fields.
	Decimal place entry	The number of decimal places can be 63 for packed and zoned fields.
Calculation specifications	Length and decimal place entries	The length and number of decimal places can be 63 for packed and zoned fields.
	CHAIN, READ, READE, READP, AND READPE operations	Allow a data structure to be specified in the result field when Factor 2 is the name of an externally-described file.
	CHAIN, READ, READC, READE, READP, READPE, WRITE, UPDATE operations	Allow an externally-described data structure to be specified in the result field when Factor 2 is the name of an externally-described record format.
	SORTA operation	Now has an extended Factor 2, allowing %SUBARR to be specified.

Table 29. New Language Elements Since V5R2

Language Unit	Element	Description
Built-in Functions	%SUBARR(array:starting element {:number of elements})	Returns a section of the array, or allows a section of the array to be modified.

What's New in V5R2?

The following list describes the enhancements made to ILE RPG in V5R2:

- Conversion from character to numeric

  Built-in functions %DEC, %DECH, %INT, %INTH, %UNS, %UNSH and %FLOAT are enhanced to allow character parameters. For example, %DEC('-12345.67' : 7 : 2) returns the numeric value `-12345.67`.
- Bitwise logical built-in functions

  %BITAND, %BITOR, %BITXOR and %BITNOT allow direct bit manipulation within RPG expressions.
- Complex data structures

  Data structure definition is enhanced to allow arrays of data structures and subfields of data structures defined with LIKEDS that are themselves data structures. This allows the coding of complex structures such as arrays of arrays, or arrays of structures containing subarrays of structures.

  ```
  Example:   family(f).child(i).hobbyInfo.pets(p).type = 'dog';
             family(f).child(i).hobbyInfo.pets(p).name = 'Spot';
  ```

  In addition, data structures can be defined the same as a record format, using the new LIKEREC keyword.
- Enhanced externally-described data structures

  Externally-described data structures can hold the programmer's choice of input, output, both, key or all fields. Currently, externally-described data structures can only hold input fields.

- Enhancments to keyed I/O

  Programmers can specify search arguments in keyed Input/Output operations in /FREE calculations in two new ways:

  1. By specifying the search arguments (which can be expressions) in a list.
  2. By specifying a data structure which contains the search arguments.

  ```
  Examples: D custkeyDS    e ds        extname(custfile:*key)
             /free
                CHAIN  (keyA : keyB : key3) custrec;
                CHAIN  %KDS(custkeyDS) custrec;
  ```
- Data-structure result for externally-described files

  A data structure can be specified in the result field when using I/O operations for externally-described files. This was available only for program-described files prior to V5R2. Using a data structure can improve performance if there are many fields in the file.
- UPDATE operation to update only selected fields

  A list of fields to be updated can be specified with an UPDATE operation. Tthis could only be done by using exception output prior to V5R2.

  `Example: update record %fields(salary:status).`
- 31 digit support

  Supports packed and zoned numeric data with up to 31 digits and decimal places. This is the maximum length supported by DDS. Only 30 digits and decimal places were supported prior to V5R2.
- Performance option for FEOD

  The FEOD operation is enhanced by supporting an extender N which indicates that the operation should simply write out the blocked buffers locally, without forcing a costly write to disk.
- Enhanced data area access

  The DTAARA keyword is enhanced to allow the name and library of the data area to be determined at runtime
- New assignment operators

  The new assignment operators +=, -=, *=, /=, **= allow a variable to be modified based on its old value in a more concise manner.

  ```
  Example: totals(current_customer) += count;
  ```

  This statement adds "count" to the value currently in "totals(current_customer)" without having to code "totals(current_customer)" twice.
- IFS source files

  The ILE RPG compiler can compile both main source files and /COPY files from the IFS. The /COPY and /INCLUDE directives are enhanced to support IFS file names.
- Program Call Markup Language (PCML) generation

  The ILE RPG compiler will generate an IFS file containing the PCML, representing the parameters to the program (CRTBNDRPG) or to the exported procedures (CRTRPGMOD).

Table 30. Changed Language Elements Since V5R1

Language Unit	Element	Description
Built-in functions	%DEC(expression)	Can now take parameters of type character.
	%DECH(expression)	
	%FLOAT(expression)	
	%INT(expression)	
	%INTH(expression)	
	%UNS(expression)	
	%UNSH(expression)	
Definition specification keywords	DTAARA({*VAR:}data-area-name)	The data area name can be a name, a character literal specifying 'LIBRARY/NAME' or a character variable which will determine the actual data area at runtime.
	DIM	Allowed for data structure specifications.
	LIKEDS	Allowed for subfield specifications.
	EXTNAME(filename{:extrecname}{:*ALL|*INPUT|*OUTPUT|*KEY})	The optional "type" parameter controls which type of field is extracted for the externally-described data structure.
Definition Specifications	Length and decimal place entries	The length and number of decimal places can be 31 for packed and zoned fields.
Operation codes	CHAIN, DELETE, READE, READPE, SETGT, SETLL	In free-form operations, Factor 1 can be a list of key values.
	CHAIN, READ, READC, READE, READP, READPE, UPDATE, WRITE	When used with externally-described files or record formats, a data structure may be specified in the result field.
	UPDATE	In free-form calculations, the final argument can contain a list of the fields to be updated.
	FEOD	Operation extender N is allowed. This indicates that the unwritten buffers must be made available to the database, but not necessarily be written to disk.
Calculation specifications	Length and decimal place entries	The length and number of decimal places can be 31 for packed and zoned fields.

Table 31. New Language Elements Since V5R1

Language Unit	Element	Description
Expressions	Assignment Operators += -= *= /= **=	When these assignment operators are used, the target of the operation is also the first operand of the operation.
Control Specification Keywords	DECPREC(30|31)	Controls the precision of decimal intermediate values for presentation, for example, for %EDITC and %EDITW
Definition specification keywords	LIKEREC(intrecname{:*ALL|*INPUT|*OUTPUT|*KEY})	Defines a data structure whose subfields are the same as a record format.

Table 31. New Language Elements Since V5R1 (continued)

Language Unit	Element	Description
Built-in functions	%BITAND(expression : expression)	Returns a result whose bits are on if the corresponding bits of the operands are both on.
	%BITNOT(expression)	Returns a result whose bits are the inverse of the bits in the argument.
	%BITOR(expression : expression)	Returns a result whose bits are on if either of the corresponding bits of the operands is on.
	%BITXOR(expression : expression)	Returns a result whose bits are on if exactly one of the corresponding bits of the operands is on.
	%FIELDS(name{:name...})	Used in free-form "UPDATE to specify the fields to be updated.
	%KDS(data structure)	Used in free-form keyed operation codes CHAIN, SETLL, SETGT, READE and READPE, to indicate that the keys for the operation are in the data structure.

What's New in V5R1?

The ILE RPG compiler is part of the IBM Rational Development Studio for i product, which now includes the C/C++ and COBOL compilers, and the Application Development ToolSet tools.

The major enhancements to RPG IV since V4R4 are easier interfacing with Java, new built-in functions, free form calculation specifications, control of which file is opened, qualified subfield names, and enhanced error handling.

The following list describes these enhancements:

- Improved support for calls between Java and ILE RPG using the Java Native Interface (JNI):
  - A new data type: Object
  - A new definition specification keyword: CLASS
  - The LIKE definition specification keyword has been extended to support objects.
  - The EXTPROC definition specification keyword has been extended to support Java procedures.
  - New status codes.
- New built-in functions:
  - Functions for converting a number into a duration that can be used in arithmetic expressions: %MSECONDS, %SECONDS, %MINUTES, %HOURS, %DAYS, %MONTHS, and %YEARS.
  - The %DIFF function, for subtracting one date, time, or timestamp value from another.
  - Functions for converting a character string (or date or timestamp) into a date, time, or timestamp: %DATE, %TIME, and %TIMESTAMP.
  - The %SUBDT function, for extracting a subset of a date, time, or timestamp.
  - Functions for allocating or reallocating storage: %ALLOC and %REALLOC.
  - Functions for finding an element in an array: %LOOKUP, %LOOKUPGT, %LOOKUPGE, %LOOKUPLT, and %LOOKUPLE.
  - Functions for finding an element in a table: %TLOOKUP, %TLOOKUPGT, %TLOOKUPGE, %TLOOKUPLT, and %TLOOKUPLE.
  - Functions for verifying that a string contains only specified characters (or finding the first or last exception to this rule): %CHECK and %CHECKR

  - The %XLATE function, for translating a string based on a list of from-characters and to-characters.
  - The %OCCUR function, for getting or setting the current occurrence in a multiple-occurrence data structure.
  - The %SHTDN function, for determining if the operator has requested shutdown.
  - The %SQRT function, for calculating the square root of a number.
- A new free-form syntax for calculation specifications. A block of free-form calculation specifcations is delimited by the compiler directives /FREE and /END-FREE.

  Note: These directives are no longer needed.
- You can specify the EXTFILE and EXTMBR keywords on the file specification to control which external file is used when a file is opened.
- Support for qualified names in data structures:
  - A new definition specification keyword: QUALIFIED. This keyword specifies that subfield names will be qualified with the data structure name.
  - A new definition specification keyword: LIKEDS. This keyword specifies that subfields are replicated from another data structure. The subfield names will be qualified with the new data structure name. LIKEDS is allowed for prototyped parameters; it allows the parameter's subfields to be used directly in the called procedure.
  - The INZ definition specification keyword has been extended to allow a data structure to be initialized based on its parent data structure.
- Enhanced error handling:
  - Three new operation codes (MONITOR, ON-ERROR, and ENDMON) allow you to define a group of operations with conditional error handling based on the status code.

Other enhancements have been made to this release as well. These include:

- You can specify parentheses on a procedure call that has no parameters.
- You can specify that a procedure uses ILE C or ILE CL calling conventions, on the EXTPROC definition specification keyword.
- The following /DEFINE names are predefined: *VnRnMn, *ILERPG, *CRTBNDRPG, and *CRTRPGMOD.
- The search string in a %SCAN operation can now be longer than string being searched. (The string will not be found, but this will no longer generate an error condition.)
- The parameter to the DIM, OCCURS, and PERRCD keywords no longer needs to be previously defined.
- The %PADDR built-in function can now take either a prototype name or an entry point name as its argument.
- A new operation code, ELSEIF, combines the ELSE and IF operation codes without requiring an additional ENDIF.
- The DUMP operation code now supports the A extender, which means that a dump is always produced - even if DEBUG(*NO) was specified.
- A new directive, /INCLUDE, is equivalent to /COPY except that /INCLUDE is not expanded by the SQL preprocessor. Included files cannot contain embedded SQL or host variables.
- The OFLIND file-specification keyword can now take any indicator, including a named indicator, as an argument.
- The LICOPT (licensed internal code options) keyword is now available on the CRTRPGMOD and CRTBNDRPG commands.
- The PREFIX file description keyword can now take an uppercase character literal as an argument. The literal can end in a period, which allows the file to be used with qualified subfields.
- The PREFIX definition specification keyword can also take an uppercase character literal as an argument. This literal cannot end in a period.

The following tables summarize the changed and new language elements, based on the part of the language affected.

Table 32. Changed Language Elements Since V4R4

Language Unit	Element	Description
Built-in functions	%CHAR(expression{:format})	The optional second parameter specifies the desired format for a date, time, or timestamp. The result uses the format and separators of the specified format, not the format and separators of the input.
	%PADDR(prototype-name)	This function can now take either a prototype name or an entry point name as its argument.
Definition specification keywords	EXTPROC(*JAVA:class-name:proc-name)	Specifies that a Java method is called.
	EXTPROC(*CL:proc-name)	Specifies a procedure that uses ILE CL conventions for return values.
	EXTPROC(*CWIDEN:proc-name)	Specifies a procedure that uses ILE C conventions with parameter widening.
	EXTPROC(*CNOWIDEN:proc-name)	Specifies a procedure that uses ILE C conventions without parameter widening.
	INZ(*LIKEDS)	Specifies that a data structure defined with the LIKEDS keyword inherits the initialization from its parent data structure.
	LIKE(object-name)	Specifies that an object has the same class as another object.
	PREFIX(character-literal{:number})	Prefixes the subfields with the specified character literal, optionally replacing the specified number of characters.
File specification keywords	OFLIND(name)	This keyword can now take any named indicator as a parameter.
	PREFIX(character-literal{:number})	Prefixes the subfields with the specified character literal, optionally replacing the specified number of characters.
Operation codes	DUMP (A)	This operation code can now take the A extender, which causes a dump to be produced even if DEBUG(*NO) was specified.

Table 33. New Language Elements Since V4R4

Language Unit	Element	Description
Data types	Object	Used for Java objects
Compiler directives	/FREE ... /END-FREE	The /FREE... /END-FREE compiler directives denote a free-form calculation specifications block.
	/INCLUDE	Equivalent to /COPY, except that it is not expanded by the SQL preprocessor. Can be used to inlcude nested files that are within the copied file. The copied file cannot have embedded SQlL or host variables.

Table 33. New Language Elements Since V4R4 (continued)

Language Unit	Element	Description
Definition specification keywords	CLASS(*JAVA:class-name)	Specifies the class for an object.
	LIKEDS(dsname)	Specifies that a data structure, prototyped parameter, or return value inherits the subfields of another data strucutre.
	QUALIFIED	Specifies that the subfield names in a data structure are qualified with the data structure name.
File specification keywords	EXTFILE(filename)	Specifies which file is opened. The value can be a literal or a variable. The default file name is the name specified in position 7 of the file specification. The default library is *LIBL.
	EXTMBR(membername)	Specifies which member is opened. The value can be a literal or a variable. The default is *FIRST.

Table 33. New Language Elements Since V4R4 (continued)

Language Unit	Element	Description
Built-in functions	%ALLOC(num)	Allocates the specified amount of storage.
	%CHECK(comparator:base{:start})	Finds the first character in the base string that is not in the comparator.
	%CHECKR(comparator:base{:start})	Finds the last character in the base string that is not in the comparator.
	%DATE(expression{:date-format})	Converts the expression to a date.
	%DAYS(num)	Converts the number to a duration, in days.
	%DIFF(op1:op2:unit)	Calculates the difference (duration) between two date, time, or timestamp values in the specified units.
	%HOURS(num)	Converts the number to a duration, in hours.
	%LOOKUPxx(arg:array{:startindex {:numelems}})	Finds the specified argument, or the specified type of near-match, in the specified array.
	%MINUTES(num)	Converts the number to a duration, in minutes.
	%MONTHS(num)	Converts the number to a duration, in months.
	%MSECONDS(num)	Converts the number to a duration, in microseconds.
	%OCCUR(dsn-name)	Sets or gets the current position of a multiple-occurrence data structure.
	%REALLOC(pointer:number)	Reallocates the specified amount of storage for the specified pointer.
	%SECONDS(num)	Converts the number to a duration, in seconds.
	%SHTDN	Checks if the system operator has requested shutdown.
	%SQRT(numeric-expression)	Calculates the square root of the specified number.
	%SUBDT(value:unit)	Extracts the specified portion of a date, time, or timestamp value.
	%THIS	Returns an Object value that contains a reference to the class instance on whose behalf the native method is being called.
	%TIME(expression{:time-format})	Converts the expression to a time.
	%TIMESTAMP(expression {:*ISO|*ISO0})	Converts the expression to a timestamp.
	%TLOOKUP(arg:search-table {:alt-table})	Finds the specified argument, or the specified type of near-match, in the specified table.
	%XLATE(from:to:string{:startpos})	Translates the specified string, based on the from-string and to-string.
	%YEARS(num)	Converts the number to a duration, in years.

Table 33. New Language Elements Since V4R4 (continued)

Language Unit	Element	Description
Operation codes	MONITOR	Begins a group of operations with conditional error handling.
	ON-ERROR	Performs conditional error handling, based on the status code.
	ENDMON	Ends a group of operations with conditional error handling.
	ELSEIF	Equivalent to an ELSE operation code followed by an IF operation code.
CRTBNDRPG and CRTRPGMOD keywords	LICOPT(options)	Specifies Licensed Internal Code options.

What's New in V4R4?

The major enhancements to RPG IV since V4R2 are the support for running ILE RPG modules safely in a threaded environment, the new 3-digit and 20-digit signed and unsigned integer data types, and support for a new Universal Character Set Version 2 (UCS-2) data type and for conversion between UCS-2 fields and graphic or single-byte character fields.

The following list describes these enhancements:

- Support for calling ILE RPG procedures from a threaded application, such as Domino® or Java.
  - The new control specification keyword THREAD(*SERIALIZE) identifies modules that are enabled to run in a multithreaded environment. Access to procedures in the module is serialized.
- Support for new 1-byte and 8-byte integer data types: 3I and 20I signed integer, and 3U and 20U unsigned integer
  - These new integer data types provide you with a greater range of integer values and can also improve performance of integer computations, taking full advantage of the 64-bit AS/400 RISC processor.
  - The new 3U type allows you to more easily communicate with ILE C procedures that have single-byte character (char) return types and parameters passed by value.
  - The new INTPREC control specification keyword allows you to specify 20-digit precision for intermediate values of integer and unsigned binary arithmetic operations in expressions.
  - Built-in functions %DIV and %REM have been added to support integer division and remainder operations.
- Support for new Universal Character Set Version 2 (UCS-2) or Unicode data type
  - The UCS-2 (Unicode) character set can encode the characters for many written languages. The field is a character field whose characters are two bytes long.
  - By adding support for Unicode, a single application can now be developed for a multinational corporation, minimizing the necessity to perform code page conversion. The use of Unicode permits the processing of characters in multiple scripts without loss of integrity.
  - Support for conversions between UCS-2 fields and graphic or single-byte character fields using the MOVE and MOVEL operations, and the new %UCS2 and %GRAPH built-in functions.
  - Support for conversions between UCS-2 fields or graphic fields with different Coded Character Set Identifiers (CCSIDs) using the EVAL, MOVE, and MOVEL operations, and the new %UCS2 built-in function.

Other enhancements have been made to this release as well. These include:

- New parameters for the OPTION control specification keyword and on the create commands:
  - *SRCSTMT allows you to assign statement numbers for debugging from the source IDs and SEU sequence numbers in the compiler listing. (The statement number is used to identify errors in the compiler listing by the debugger, and to identify the statement where a run-time error occurs.) *NOSRCSTMT specifies that statement numbers are associated with the Line Numbers of the listing and the numbers are assigned sequentially.
  - Now you can choose not to generate breakpoints for input and output specifications in the debug view with *NODEBUGIO. If this option is selected, a STEP on a READ statement in the debugger will step to the next calculation, rather than stepping through the input specifications.
- New special words for the INZ definition specification keyword:
  - INZ(*EXTDFT) allows you to use the default values in the DDS for initializing externally described data structure subfields.
  - Character variables initialized by INZ(*USER) are initialized to the name of the current user profile.
- The new %XFOOT built-in function sums all elements of a specified array expression.
- The new EVALR operation code evaluates expressions and assigns the result to a fixed-length character or graphic result. The assignment right-adjusts the data within the result.
- The new FOR operation code performs an iterative loop and allows free-form expressions for the initial, increment, and limit values.
- The new LEAVESR operation code can be used to exit from any point within a subroutine.
- The new *NEXT parameter on the OVERLAY(name:*NEXT) keyword indicates that a subfield overlays another subfield at the next available position.
- The new *START and *END values for the SETLL operation code position to the beginning or end of the file.
- The ability to use hexadecimal literals with integer and unsigned integer fields in initialization and free-form operations, such as EVAL, IF, etc.
- New control specification keyword OPENOPT{(*NOINZOFL | *INZOFL)} to indicate whether the overflow indicators should be reset to *OFF when a file is opened.
- Ability to tolerate pointers in teraspace — a memory model that allows more than 16 megabytes of contiguous storage in one allocation.

The following tables summarize the changed and new language elements, based on the part of the language affected.

Table 34. Changed Language Elements Since V4R2

Language Unit	Element	Description
Control specification keywords	OPTION(*{NO}SRCSTMT)	*SRCSTMT allows you to request that the compiler use SEU sequence numbers and source IDs when generating statement numbers for debugging. Otherwise, statement numbers are associated with the Line Numbers of the listing and the numbers are assigned sequentially.
	OPTION(*{NO}DEBUGIO)	*{NO}DEBUGIO, determines if breakpoints are generated for input and output specifications.

Table 34. Changed Language Elements Since V4R2 (continued)

Language Unit	**Element**	**Description**
Definition specification keywords	INZ(*EXTDFT)	All externally described data structure subfields can now be initialized to the default values specified in the DDS.
	INZ(*USER)	Any character field or subfield can be initialized to the name of the current user profile.
	OVERLAY(name:*NEXT)	The special value *NEXT indicates that the subfield is to be positioned at the next available position within the overlayed field.
	OPTIONS(*NOPASS *OMIT *VARSIZE *STRING *RIGHTADJ)	The new OPTIONS(*RIGHTADJ) specified on a value or constant parameter in a function prototype indicates that the character, graphic, or UCS-2 value passed as a parameter is to be right adjusted before being passed on the procedure call.
Definition specification positions 33-39 (To Position/Length)	3 and 20 digits allowed for I and U data types	Added to the list of allowed values for internal data types to support 1-byte and 8-byte integer and unsigned data.
Internal data type	C (UCS-2 fixed or variable-length format)	Added to the list of allowed internal data types on the definition specifications. The UCS-2 (Unicode) character set can encode the characters for many written languages. The field is a character field whose characters are two bytes long.
Data format	C (UCS-2 fixed or variable-length format)	UCS-2 format added to the list of allowed data formats on the input and output specifications for program described files.
Command parameter	OPTION	*NOSRCSTMT, *SRCSTMT, *NODEBUGIO, and *DEBUGIO have been added to the OPTION parameter on the CRTBNDRPG and CRTRPGMOD commands.

Table 35. New Language Elements Since V4R2

Language Unit	Element	Description
Control specification keywords	CCSID(*GRAPH: *IGNORE | *SRC | number)	Sets the default graphic CCSID for the module. This setting is used for literals, compile-time data and program-described input and output fields and definitions. The default is *IGNORE.
	CCSID(*UCS2: number)	Sets the default UCS-2 CCSID for the module. This setting is used for literals, compile-time data and program-described input and output fields and definitions. The default is 13488.
	INTPREC(10 | 20)	Specifies the decimal precision of integer and unsigned intermediate values in binary arithmetic operations in expressions. The default, INTPREC(10), indicates that 10-digit precision is to be used.
	OPENOPT{(*NOINZOFL | *INZOFL)}	Indicates whether the overflow indicators should be reset to *OFF when a file is opened.
	THREAD(*SERIALIZE)	Indicates that the module is enabled to run in a multithreaded environment. Access to the procedures in the module is to be serialized.
Definition specification keywords	CCSID(number | *DFT)	Sets the graphic and UCS-2 CCSID for the definition.
Built-in functions	%DIV(n:m)	Performs integer division on the two operands n and m; the result is the integer portion of n/m. The operands must be numeric values with zero decimal positions.
	%GRAPH(char-expr | graph-expr | UCS2-expr {: ccsid})	Converts to graphic data from single-byte character, graphic, or UCS-2 data.
	%REM(n:m)	Performs the integer remainder operation on two operands n and m; the result is the remainder of n/m. The operands must be numeric values with zero decimal positions.
	%UCS2(char-expr | graph-expr | UCS2-expr {: ccsid})	Converts to UCS-2 data from single-byte character, graphic, or UCS-2 data.
	%XFOOT(array-expr)	Produces the sum of all the elements in the specified numeric array expression.

Table 35. New Language Elements Since V4R2 (continued)

Language Unit	Element	Description
Operation codes	EVALR	Evaluates an assignment statement of the form result=expression. The result will be right-justified.
	FOR	Begins a group of operations and indicates the number of times the group is to be processed. The initial, increment, and limit values can be free-form expressions.
	ENDFOR	ENDFOR ends a group of operations started by a FOR operation.
	LEAVESR	Used to exit from anywhere within a subroutine.

What's New in V4R2?

The major enhancements to RPG IV since V3R7 are the support for variable-length fields, several enhancements relating to indicators, and the ability to specify compile options on the control specifications. These further improve the RPG product for integration with the operating system and ILE interlanguage communication.

The following list describes these enhancements:

- Support for variable-length fields

  This enhancement provides full support for variable-length character and graphic fields. Using variable-length fields can simplify many string handling tasks.
- Ability to use your own data structure for INDARA indicators

  Users can now access logical data areas and associate an indicator data structure with each WORKSTN and PRINTER file that uses INDARA, instead of using the *IN array for communicating values to data management.
- Ability to use built-in functions instead of result indicators

  Built-in functions %EOF, %EQUAL, %FOUND, and %OPEN have been added to query the results of input/output operations. Built-in functions %ERROR and %STATUS, and the operation code extender 'E' have been added for error handling.
- Compile options on the control specification

  Compile options, specified through the CRTBNDRPG and CRTRPGMOD commands, can now be specified through the control specification keywords. These compile options will be used on every compile of the program.

In addition, the following new function has been added:

- Support for import and export of procedures and variables with mixed case names
- Ability to dynamically set the DECEDIT value at runtime
- Built-in functions %CHAR and %REPLACE have been added to make string manipulation easier
- New support for externally defined *CMDY, *CDMY, and *LONGJUL date data formats
- An extended range for century date formats
- Ability to define indicator variables
- Ability to specify the current data structure name as the parameter for the OVERLAY keyword
- New status code 115 has been added to indicate variable-length field errors

- Support for application profiling
- Ability to handle packed-decimal data that is not valid when it is retrieved from files using FIXNBR(*INPUTPACKED)
- Ability to specify the BNDDIR command parameter on the CRTRPGMOD command.

The following tables summarize the changed and new language elements, based on the part of the language affected.

Table 36. Changed Language Elements Since V3R7

Language Unit	Element	Description
Control specification keywords	DECEDIT(*JOBRUN | 'value')	The decimal edit value can now be determined dynamically at runtime from the job or system value.
Definition specification keywords	DTAARA {(data_area_name)}	Users can now access logical data areas.
	EXPORT {(external_name)}	The external name of the variable being exported can now be specified as a parameter for this keyword.
	IMPORT {(external_name)}	The external name of the variable being imported can now be specified as a parameter for this keyword.
	OVERLAY(name{:pos})	The name parameter can now be the name of the current data structure.

Extended century format	*CYMD (cyy/mm/dd)	The valid values for the century character 'c' are now:
		`'c'  Years`
		`----------------------`
		`0  1900-1999`
		`1  2000-2099`
		`.  .`
		`.  .`
		`.  .`
		`9  2800-2899`

Internal data type	N (Indicator format)	Added to the list of allowed internal data types on the definition specifications. Defines character data in the indicator format.
Data format	N (Indicator format)	Indicator format added to the list of allowed data formats on the input and output specifications for program described files.
Data Attribute	*VAR	Added to the list of allowed data attributes on the input and output specifications for program described files. It is used to specify variable-length fields.
Command parameter	FIXNBR	The *INPUTPACKED parameter has been added to handle packed-decimal data that is not valid.

Table 37. New Language Elements Since V3R7

Language Unit	New	Description
Control specification keywords	ACTGRP(*NEW | *CALLER | 'activation- group-name')	The ACTGRP keyword allows you to specify the activation group the program is associated with when it is called.
	ALWNULL(*NO | *INPUTONLY | *USRCTL)	The ALWNULL keyword specifies how you will use records containing null-capable fields from externally described database files.
	AUT(*LIBRCRTAUT | *ALL | *CHANGE | *USE | *EXCLUDE | 'authorization-list-name')	The AUT keyword 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.
	BNDDIR('binding -directory-name' {:'binding- directory-name'...})	The BNDDIR keyword specifies the list of binding directories that are used in symbol resolution.
	CVTOPT(*{NO}DATETIME *{NO}GRAPHIC *{NO}VARCHAR *{NO}VARGRAPHIC)	The CVTOPT keyword is used to determine how the ILE RPG compiler handles date, time, timestamp, graphic data types, and variable-length data types that are retrieved from externally described database files.
	DFTACTGRP(*YES | *NO)	The DFTACTGRP keyword specifies the activation group in which the created program will run when it is called.
	ENBPFRCOL(*PEP | *ENTRYEXIT | *FULL)	The ENBPFRCOL keyword specifies whether performance collection is enabled.
	FIXNBR(*{NO}ZONED *{NO}INPUTPACKED)	The FIXNBR keyword specifies whether decimal data that is not valid is fixed by the compiler.
	GENLVL(number)	The GENLVL keyword controls the creation of the object.
	INDENT(*NONE | 'character-value')	The INDENT keyword specifies whether structured operations should be indented in the source listing for enhanced readability.
	LANGID(*JOBRUN | *JOB | 'language-identifier')	The LANGID keyword indicates which language identifier is to be used when the sort sequence is *LANGIDUNQ or *LANGIDSHR.
	OPTIMIZE(*NONE | *BASIC | *FULL)	The OPTIMIZE keyword specifies the level of optimization, if any, of the object.
	OPTION(*{NO}XREF *{NO}GEN *{NO}SECLVL *{NO}SHOWCPY *{NO}EXPDDS *{NO}EXT *{NO}SHOWSKP)	The OPTION keyword specifies the options to use when the source member is compiled.
	PRFDTA(*NOCOL | *COL)	The PRFDTA keyword specifies whether the collection of profiling data is enabled.

Table 37. New Language Elements Since V3R7 (continued)

Language Unit	New	Description
	SRTSEQ(*HEX | *JOB | *JOBRUN | *LANGIDUNQ | *LANGIDSHR | 'sort-table-name')	The SRTSEQ keyword specifies the sort sequence table that is to be used in the ILE RPG source program.
	TEXT(*SRCMBRTXT | *BLANK | 'description')	The TEXT keyword allows you to enter text that briefly describes the object and its function.
	TRUNCNBR(*YES | *NO)	The TRUNCNBR keyword specifies if the truncated value is moved to the result field or if an error is generated when numeric overflow occurs while running the object.
	USRPRF(*USER | *OWNER)	The USRPRF keyword specifies the user profile that will run the created program object.
File Description Specification keywords	INDDS(data_structure_name)	The INDDS keyword lets you associate a data structure name with the INDARA indicators for a workstation or printer file.
Definition specification keywords	VARYING	Defines variable-length fields when specified on character data or graphic data.
Built-in functions	%CHAR(graphic, date, time or timestamp expression)	Returns the value in a character data type.
	%EOF{file name}	Returns '1' if the most recent file input operation or write to a subfile (for a particular file, if specified) ended in an end-of-file or beginning-of-file condition; otherwise, it returns '0'.
	%EQUAL{file name}	Returns '1' if the most recent SETLL (for a particular file, if specified) or LOOKUP operation found an exact match; otherwise, it returns '0'.
	%ERROR	Returns '1' if the most recent operation code with extender 'E' specified resulted in an error; otherwise, it returns '0'.
	%FOUND{file name}	Returns '1' if the most recent relevant operation (for a particular file, if specified) found a record (CHAIN, DELETE, SETGT, SETLL), an element (LOOKUP), or a match (CHECK, CHECKR and SCAN); otherwise, it returns '0'.
	%OPEN(file name)	Returns '1' if the specified file is open and '0' if the specified file is closed.
	%REPLACE(replacement string: source string {:start position {:source length to replace}})	Returns the string produced by inserting a **replacement string** into a **source string**, starting at the **start position** and replacing the specified number of characters.

Table 37. New Language Elements Since V3R7 (continued)

Language Unit	New	Description
	%STATUS{file name}	If no program or file error occurred since the most recent operation code with extender 'E' specified, it returns 0. If an error occurred, it returns the most recent value set for any program or file status. If a file is specified, the value returned is the most recent status for that file.
Operation code Extender	E	Allows for error handling using the %ERROR and %STATUS built-in functions on the CALLP operation and all operations that allow error indicators.
New century formats	*CMDY (cmm/dd/yy)	To be used by the MOVE, MOVEL, and TEST operations.
	*CDMY (cdd/mm/yy)	To be used by the MOVE, MOVEL, and TEST operations.
New 4-digit year format	*LONGJUL (yyyy/ddd)	To be used by the MOVE, MOVEL, and TEST operations.
Command parameters	PRFDTA	The PRFDTA parameter specifies whether the collection of profiling data is enabled.
	BNDDIR	The BNDDIR parameter was previously only allowed on the CRTBNDRPG command and not on the CRTRPGMOD command, now it is allowed on both commands.

What's New in V3R7?

The major enhancements to RPG IV since V3R6 are the new support for database null fields, and the ability to better control the precision of intermediate results in expressions. Other enhancements include the addition of a floating point data type and support for null-terminated strings. These further improve the RPG product for integration with the operating system and ILE interlanguage communication. This means greater flexibility for developing applications.

The following is a list of these enhancements including a number of new built-in functions and usability enhancements:

- Support for database null fields

  This enhancement allows users to process database files which contain null-capable fields, by allowing these fields to be tested for null and set to null.
- Expression intermediate result precision

  A new control specification keyword and new operation code extenders on free-form expression specifications allow the user better control over the precision of intermediate results.
- New floating point data type

  The new floating point data type has a much larger range of values than other data types. The addition of this data type will improve integration with the database and improve interlanguage communication in an ILE environment, specifically with the C and C++ languages.
- Support for null terminated strings

The new support for null terminated strings improves interlanguage communication. It allows users full control over null terminated data by allowing users to define and process null terminated strings, and to conveniently pass character data as parameters to procedures which expect null terminated strings.

- Pointer addition and subtraction

  Free-form expressions have been enhanced to allow adding an offset to a pointer, subtracting an offset from a pointer, and determining the difference between two pointers.

- Support for long names

  Names longer than 10 characters have been added to the RPG language. Anything defined on the definition or procedure specifications can have a long name and these names can be used anywhere where they fit within the bounds of an entry. In addition, names referenced on any free-form specification may be continued over multiple lines.

- New built-in functions

  A number of new built-in functions have been added to the language which improve the following language facilities:

  - editing (%EDITW, %EDITC, %EDITFLT)
  - scanning strings (%SCAN)
  - type conversions (%INT, %FLOAT, %DEC, %UNS)
  - type conversions with half-adjust (%INTH, %DECH, %UNSH)
  - precision of intermediate results for decimal expressions (%DEC)
  - length and decimals of variables and expressions (%LEN, %DECPOS)
  - absolute value (%ABS)
  - set and test null-capable fields (%NULLIND)
  - handle null terminated strings (%STR)

- Conditional compilation

  RPG IV has been extended to support conditional compilation. This support will include the following:

  - defining conditions (/DEFINE, /UNDEFINE),
  - testing conditions (/IF, /ELSEIF, /ELSE, /ENDIF)
  - stop reading current source file (/EOF)
  - a new command option (DEFINE) to define up to 32 conditions on the CRTBNDRPG and CRTRPGMOD commands.

- Date enhancements

  Several enhancements have been made to improve date handling operations. The TIME operation code is extended to support Date, Time or Timestamp fields in the result field. Moving dates or times from and to character fields no longer requires separator characters. Moving UDATE and *DATE fields no longer requires a format code to be specified. Date fields can be initialized to the system (*SYS) or job (*JOB) date on the definition specifications.

- Character comparisons with alternate collating sequence

  Specific character variables can be defined so that the alternate collating sequence is not used in comparisons.

- Nested /COPY members

  You can now nest /COPY directives. That is, a /COPY member may contain one (or more) /COPY directives which can contain further /COPY directives and so on.

- Storage management

  You can now use the new storage management operation codes to allocate, reallocate and deallocate storage dynamically.

- Status codes for storage management and float underflow errors.

Two status codes 425 and 426 have been added to indicate storage management errors. Status code 104 was added to indicate that an intermediate float result is too small.

The following tables summarize the changed and new language elements, based on the part of the language affected.

Table 38. Changed Language Elements Since V3R6

Language Unit	Element	Description
Definition specification keywords	ALIGN	ALIGN can now be used to align float subfields along with the previously supported integer and unsigned alignment.
	OPTIONS(*NOPASS *OMIT *VARSIZE *STRING)	The *STRING option allows you to pass a character value as a null-terminated string.
Record address type	F (Float format)	Added to the list of allowed record address types on the file description specifications. Signals float processing for a program described file.
Internal data type	F (Float format)	Added to the list of allowed internal data types on the definition specifications. Defines a floating point standalone field, parameter, or data structure subfield.
Data format	F (Float format)	Added to the list of allowed data formats on the input and output specifications for program described files.

Table 39. New Language Elements Since V3R6

Language Unit	New	Description
Control specification keywords	COPYNEST('1-2048')	Specifies the maximum depth for nesting of /COPY directives.
	EXPROPTS(*MAXDIGITS | *RESDECPOS)	Expression options for type of precision (default or "Result Decimal Position" precision rules)
	FLTDIV{(*NO | *YES)}	Indicates that all divide operations in expressions are computed in floating point.
Definition specification keywords	ALTSEQ(*NONE)	Forces the normal collating sequence to be used for character comparison even when an alternate collating sequence is specified.
Built-in functions	%ABS	Returns the absolute value of the numeric expression specified as the parameter.
	%DEC and %DECH	Converts the value of the numeric expression to decimal (packed) format with the number of digits and decimal positions specified as parameters. %DECH is the same as %DEC, but with a half adjust applied.
	%DECPOS	Returns the number of decimal positions of the numeric variable or expression. The value returned is a constant, and may be used where a constant is expected.

Table 39. New Language Elements Since V3R6 (continued)

Language Unit	New	Description
	%EDITC	This function returns a character result representing the numeric value edited according to the edit code.
	%EDITFLT	Converts the value of the numeric expression to the character external display representation of float.
	%EDITW	This function returns a character result representing the numeric value edited according to the edit word.
	%FLOAT	Converts the value of the numeric expression to float format.
	%INT and %INTH	Converts the value of the numeric expression to integer. Any decimal digits are truncated with %INT and rounded with %INTH.
	%LEN	Returns the number of digits or characters of the variable expression.
	%NULLIND	Used to query or set the null indicator for null-capable fields.
	%SCAN	Returns the first position of the search argument in the source string, or 0 if it was not found.
	%STR	Used to create or use null-terminated strings, which are very commonly used in C and C++ applications.
	%UNS and %UNSH	Converts the value of the numeric expression to unsigned format. Any decimal digits are truncated with %UNS and rounded with %UNSH.
Operation code Extenders	N	Sets pointer to *NULL after successful DEALLOC
	M	Default precision rules
	R	No intermediate value will have fewer decimal positions than the result ("Result Decimal Position" precision rules)
Operation codes	ALLOC	Used to allocate storage dynamically.
	DEALLOC	Used to deallocate storage dynamically.
	REALLOC	Used to reallocate storage dynamically.

What's New in V3R6/V3R2?

The major enhancement to RPG IV since V3R1 is the ability to code a module with more than one procedure. What does this mean? In a nutshell, it means that you can code an module with one or more prototyped procedures, where the procedures can have return values and run without the use of the RPG cycle.

Writing a module with multiple procedures enhances the kind of applications you can create. Any application consists of a series of logical units that are conceived to accomplish a particular task. In order to develop applications with the greatest flexibility, it is important that each logical unit be as independent as possible. Independent units are:

- Easier to write from the point of view of doing a specific task.
- Less likely to change any data objects other than the ones it is designed to change.
- Easier to debug because the logic and data items are more localized.
- Maintained more readily since it is easier to isolate the part of the application that needs changing.

The main benefit of coding a module with multiple procedures is greater control and better efficiency in coding a modular application. This benefit is realized in several ways. You can now:

- Call procedures and programs by using the same call operation and syntax.
- Define a prototype to provide a check at compile time of the call interface.
- Pass parameters by value or by reference.
- Define a procedure that will return a value and call the procedure within an expression.
- Limit access to data items by defining local definitions of variables.
- Code a module that does not make use of the cycle.
- Call a procedure recursively.

The run-time behavior of the main procedure in a module is the same as that of a V3R1 procedure. The run-time behavior of any subsequent procedures differs somewhat from a V3R1 program, most notably in the areas of procedure end and exception handling. These differences arise because there is no cycle code that is generated for these procedures.

Other enhancements have been made to for this release as well. These include:

- Support for two new integer data types: signed integer (I), and unsigned integer (U)

  The use of the integer data types provides you with a greater range of values than the binary data type. Integer data types can also improve performance of integer computations.
- *CYMD support for the MOVE, MOVEL, and TEST operations

  You can now use the *CYMD date format in certain operations to work with system values that are already in this data format.
- Ability to copyright your programs and modules by using the COPYRIGHT keyword on the control specification

  The copyright information that is specified using this keyword becomes part of the DSPMOD, DSPPGM, or DSPSRVPGM information.
- User control of record blocking using keyword BLOCK

  You can request record blocking of DISK or SEQ files to be done even when SETLL, SETGT, or CHAIN operations are used on the file. You can also request that blocking not be done. Use of blocking in these cases may significantly improve runtime performance.
- Improved PREFIX capability

  Changes to the PREFIX keyword for either file-description and definition specifications allow you to replace characters in the existing field name with the prefix string.
- Status codes for trigger program errors

  Two status codes 1223 and 1224 have been added to indicate trigger program errors.

The following tables summarize the changed and new language elements, based on the part of the language affected.

Table 40. Changed Language Elements Since V3R1

Language Unit	Element	Description
File description specification keywords	PREFIX(prefix_string {:nbr_of_char_ replaced})	Allows prefixing of string to a field name or a partial rename of the field name
Definition specification keywords	CONST{(constant)}	Specifies the value of a named constant, or indicates that a prototyped parameter that is passed by reference has a constant value
	PREFIX(prefix_string {:nbr_of_char_ replaced})	Allows prefixing of string to a field name or a partial rename of the field name
Operation codes	RETURN	Returns control to the caller, and returns a value, if specified

Table 41. New Language Elements Since V3R1

Language Unit	New	Description
Control specification keywords	COPYRIGHT('copyright string')	Allows you to associate copyright information with modules and programs
	EXTBININT{(*NO | *YES)}	Specifies that binary fields in externally-described files be assigned an integer format during program processing
	NOMAIN	Indicates that the module has only subprocedures
File description specification keywords	BLOCK(*YES |*NO)	Allows you to control whether record blocking occurs (assuming other conditions are met)
Definition specification keywords	ALIGN	Specifies whether integer or unsigned fields should be aligned
	EXTPGM(name)	Indicates the external name of the prototyped program
	EXTPROC(name)	Indicates the external name of the prototyped procedure
	OPDESC	Indicates whether operational descriptors are to be passed for the prototyped bound call
	OPTIONS(*NOPASS *OMIT *VARSIZE)	Specifies various options for prototyped parameters
	STATIC	Specifies that the local variable is to use static storage
	VALUE	Specifies that the prototyped parameter is to be passed by value
Built-in functions	%PARMS	Returns the number of parameters passed on a call
Operation codes	CALLP	Calls a prototyped program or procedure

Table 41. New Language Elements Since V3R1 (continued)

Language Unit	**New**	**Description**
Specification type	Procedure specification	Signals the beginning and end of a subprocedure definition
Definition type	PR	Signals the beginning of a prototype definition
	PI	Signals the beginning of a procedure interface definition
	blank in positions 24-25	Defines a prototyped parameter

Chapter 3. ILE RPG Introduction

Before using ILE RPG 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 Integrated Language Environment components in RPG programming
- Integrated Language Environment program creation strategies
- Overview of coding a module with more than one procedure and prototyped calls

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 the *IBM Rational Development Studio for i: ILE RPG 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 seven types of RPG IV specifications. Each specification type is optional. Specifications must be entered into your source program in the order shown below.

Main source section:

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.

Subprocedure section:

1. **Procedure specifications** mark the beginning and end of the subprocedure, indicate the subprocedure name, and whether it is exported.
2. **Definition specifications** describe the local data used by the subprocedure.
3. **Calculation specifications** describe the calculations done on both the global and local data and the order of the calculations.

Cycle Programming

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

- The ILE RPG compiler
- The 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 end your program some other way, either by creating an end-of-file condition by setting on the last record (LR) indicator, by creating a return condition by setting on the return (RT) indicator, or by returning directly using the RETURN operation.

Note: No cycle code is generated for subprocedures or when MAIN or NOMAIN is specified on the control specification. See

Figure 1 on page 58 shows the specific steps in the general flow of the RPG program cycle.

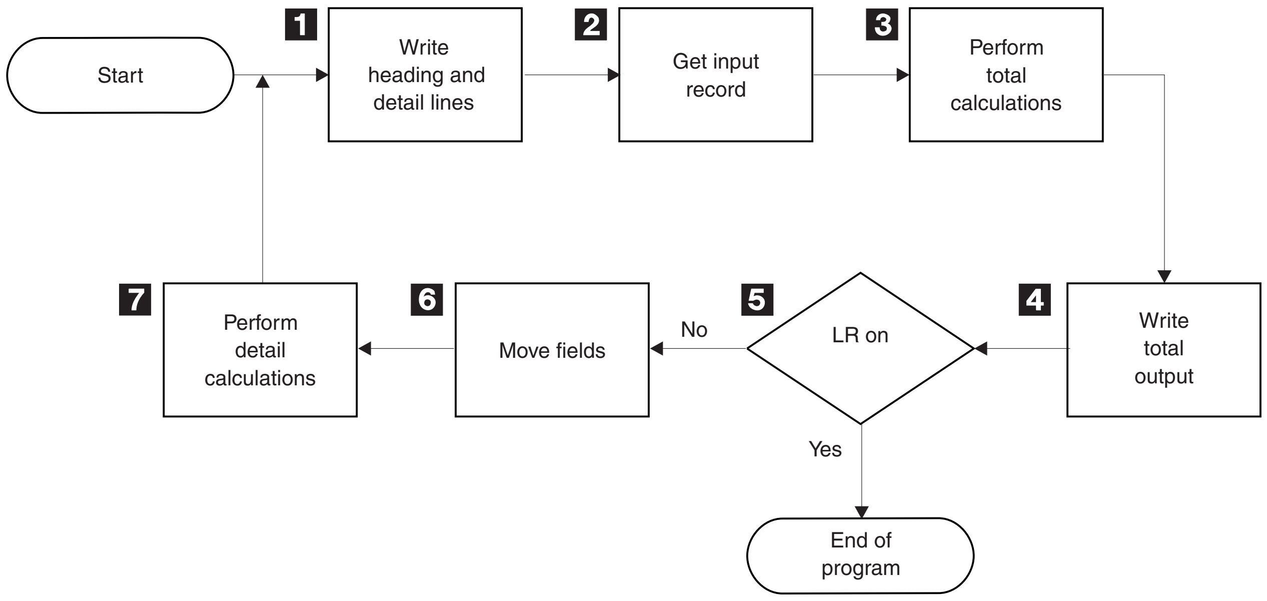

Start
1
Write heading and detail lines
2
Get input record
3
Perform total calculations
4
Write total output
5
LR on
No
Yes
End of program
6
Move fields
7
Perform detail calculations

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.

Subprocedure logic

The general flow of a subprocedure is much simpler: the calculations of a subprocedure are done once, and then the subprocedure returns. There is no cycle code generated for a subprocedure.

Indicators

An **indicator** is a one-byte character field that is either set on ('1') or off ('0'). It is generally used to indicate the result of an operation or to condition (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 or used.

Indicators can be defined as variables on the definition specifications. You can also use RPG IV indicators, which are defined either by an entry on a specification or by the RPG IV program itself.

Each RPG IV 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. You can use several types of these indicators; each type signals something different. 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.

Indicator variables can be used any place an indicator of the form *INxx may be used with the exception of the OFLIND and EXTIND keywords on the file description specifications.

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
- Conversion operations
- Data-area operations
- Date operations
- Declarative operations
- Error-handling operations
- File operations
- Indicator-setting operations
- Information operations
- Initialization operations
- Memory management operations
- Move operations
- Move zone operations
- Result operations
- Size operations
- String operations
- Structured programming operations
- Subroutine operations
- Test operations

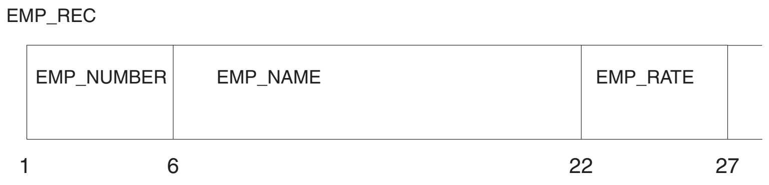
Example of an ILE RPG Program

This section illustrates a simple ILE RPG 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.

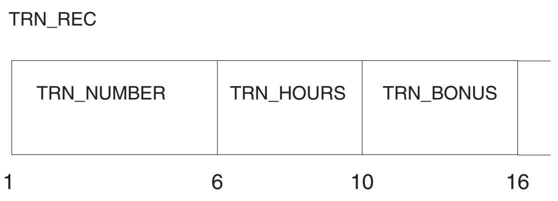
The first file, EMPLOYEE, contains employee records. The figure below shows the format of an employee record:



```
*.. 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. If more than 40 hours were worked, the employee is paid for for 1.5 times the normal rate.

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 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                                     ----  -------  ----------'
D CalcPay         PR            8P 2
D   Rate                        5P 2 VALUE
D   Hours                      10U 0 VALUE
D   Bonus                       5P 2 VALUE
```

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 ...+...

 /free
     chain trn_number emp_rec; 1
     *IN99 = NOT %found(employee); 2
     if %found(employee); 3
        pay = CalcPay (emp_rate: trn_hours: trn_bonus);
     endif;
 /end-free
```

The coding entries on the calculation specifications include:

1. 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.
2. *IN99 is assigned the opposite of %FOUND. A later output specification is conditioned by indicator 99; indicator 99 should have the value *ON if the CHAIN operation did not find a record.
3. If the CHAIN operation is successful (that is, %FOUND returns *ON), 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 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.

A Subprocedure

The subprocedure calculates the pay for the employee using the parameters passed to it. The resulting value is returned to the caller using the RETURN statement.

The procedure specifications indicate the beginning and end of the procedure. The definition specifications define the return type of the procedure, the parameters to the procedure, and the local variable Overtime.

```
P CalcPay         B
D CalcPay         PI             8P 2
D   Rate                         5P 2 VALUE
D   Hours                       10U 0 VALUE
D   Bonus                        5P 2 VALUE
D Overtime        S              5P 2 INZ(0)

 /free

    // Determine any overtime hours to be paid.

    if Hours > 40;
      Overtime = (Hours - 40) * Rate * 1.5;
      Hours = 40;
    endif;

    // Calculate the total pay and return it to the caller.

    return  Rate * Hours + Bonus + Overtime;
 /end-free
P CalcPay         E
```

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                                     ____  _______  __________'
 *-------------------------------------------------------------------------*
 * Prototype Definition for subprocedure CalcPay                           *
 *-------------------------------------------------------------------------*
D CalcPay         PR             8P 2
D   Rate                         5P 2 VALUE
D   Hours                       10U 0 VALUE
D   Bonus                        5P 2 VALUE
 *-------------------------------------------------------------------------*
 * For each record in the transaction file (TRANSACT), if the employee    *
 * is found, compute the employee's pay and print the details.            *
 *-------------------------------------------------------------------------*

 /free
   chain trn_number emp_rec;
   if %found(emp_rec);
      pay = CalcPay (emp_rate: trn_hours: trn_bonus);
   endif;
 /end-free

 *-------------------------------------------------------------------------*
 * 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'
```

```
 *-------------------------------------------------------------------------*
 * Subprocedure  -- calculates overtime pay.                               *
 *-------------------------------------------------------------------------*

P CalcPay         B
D CalcPay         PI             8P 2
D   Rate                         5P 2 VALUE
D   Hours                       10U 0 VALUE
D   Bonus                        5P 2 VALUE
D Overtime        S              5P 2 INZ(0)

 /free

   // Determine any overtime hours to be paid.
```

Using IBM i

The operating system that controls all of your interactions with IBM i Information Center is called the IBM i. From your workstation, IBM i 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.

You can obtain a complete list of publications that discuss the IBM i system at the IBM i Information Center.

Interacting with the System

You can manipulate the IBM i system using Command Language (CL). You interact with the system by entering or selecting CL commands. The 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 42. 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

Table 42. Commonly Used CL Commands (continued)

Action	**CL command** / **Result**
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

Rational Development Studio for i

IBM Rational Development Studio for i is an application development package to help you rapidly and cost-effectively increase the number of e-business applications for the IBM i.

The following components are included in Rational Development Studio for i.

- ILE RPG
- ILE COBOL
- ILE C/C++
- Application Development ToolSet (ADTS)

RPG Programming in ILE

ILE RPG 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 IBM i.

ILE is an approach to programming on IBM i. It is the result of major enhancements to the IBM i machine architecture and the IBM i operating system. The ILE family of compilers includes: ILE RPG, ILE C, ILE COBOL, ILE CL, and VisualAge® for C++. Table 43 on page 67 lists the programming languages supported by the IBM i operating system. In addition to the support for the ILE languages, support for the original program model (OPM) and extended program model (EPM) languages has been retained.

Table 43. Programming Languages Supported on the IBM i

Integrated Language Environment (ILE)	Original Program Model (OPM)	Extended Program Model (EPM)
C++	BASIC (PRPQ)	FORTRAN
C	CL	PASCAL (PRPQ)
CL	COBOL	
COBOL	PL/I (PRPQ)	
RPG	RPG	

Compared to OPM, 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 you do 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.

Service programs are a means of packaging the procedures in one or more modules into a separately bound object. Other ILE programs can access the procedures in the service program, although there is only one copy of the service program on the system. The use of service programs facilitates modularity and maintainability. You can use off-the-shelf service programs developed by third parties or, conversely, package your own service programs for third-party use. A service program is created using the Create Service Program (CRTSRVPGM) command.

You can create a binding directory to contain the names of modules and service programs that your program or service program may need. A list of binding directories can be specified when you create a program on the CRTBNDRPG, CRTSRVPGM, and CRTPGM commands. They can also be specified on the CRTRPGMOD command; however, the search for a binding directory is done when the module is bound at CRTPGM or CRTSRVPGM time. A binding directory can reduce program size because modules or service programs listed in a binding directory are used only if they are needed.

Figure 5 on page 68 shows the two approaches to program creation.

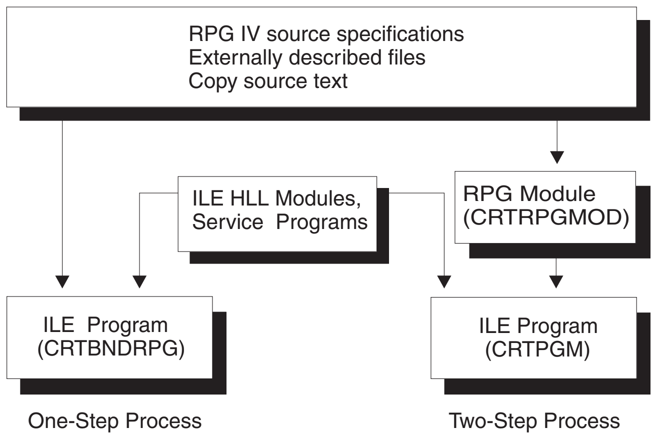


Figure 5. Program Creation in ILE

Once a program is created you can update the program using the Update Program (UPDPGM) or Update Service Program (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 "Creating a Program with the CRTBNDRPG Command" on page 101. For more information on the two-step process, see "Creating a Program with the CRTRPGMOD and CRTPGM Commands" on page 115. For more information on service programs, see "Creating a Service Program" on page 129.

Program Management

ILE provides a common basis for:

- Managing program flow
- Sharing resources
- Using application program interfaces (APIs)
- Handling exceptions during a program's run time

It gives RPG users much better control over resources than was previously possible.

ILE programs and service programs are activated into activation groups which are specified at program-creation time. The process of getting a program or service program ready to run is known as activation. Activation allocates resources 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 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 144.

You can dynamically allocate storage for a run-time 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 offer a storage model to languages that do not now provide one. RPG offers some storage management capabilities using operation codes. For more information on storage management, see "Managing Dynamically-Allocated Storage" on page 146.

Program Call

In ILE, you can write applications in which ILE RPG programs and OPM RPG/400® programs continue to interrelate through the traditional use of dynamic program calls. When using such calls, the calling program specifies 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.

You can also write ILE applications that can interrelate with faster static calls. Static calls involve calls between procedures. A procedure is a self-contained set of code that performs a task and then returns to the caller. An ILE RPG module consists of an optional main procedure followed by zero or more subprocedures. 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
- The passing of parameters by value
- The use of return values
- A greater number of parameters to be passed

Operational descriptors and omitted parameters can be useful when calling bindable APIs or procedures written in other ILE languages.

For information on running a program refer to "Running a Program" on page 138. For information on program/procedure call, refer to "Calling Programs and Procedures" on page 159.

Source Debugging

In ILE, you can perform source-level debugging on any single- or mixed-language ILE application. The ILE source debugger also supports OPM programs. You can control the flow of a program by using debug commands while the program is running. You can set conditional and unconditional job or thread breakpoints prior to running the program. After you call the program, you can then 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 run-time 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 "Debugging Programs" on page 241.

Bindable APIs

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

Some APIs that you may wish to consider using in an ILE RPG application include:

- CEETREC – Signal the Termination-Imminent Condition
- CEE4ABN – Abnormal End
- CEECRHP – Create your own heap
- CEEDSHP – Discard your own heap
- CEEFRST – Free Storage in your own heap
- CEEGTST – Get Heap Storage in your own heap
- CEECZST – Reallocate Storage in your own heap
- CEEDOD – Decompose Operational Descriptor

Note: You cannot use these or any other ILE bindable APIs from within a program created with DFTACTGRP(*YES). This is because bound calls are not allowed in this type of program.

For more information on these ILE bindable APIs, see "Running a Program" on page 138.

Multithreaded Applications

ILE RPG has two modes of operating in a multithreaded environment: concurrent and serialized. Each mode has advantages and disadvantages. You can choose which mode of operation fits each module in your application.

The RPG support for threads ensures that your static storage is handled in a threadsafe way. However, you are responsible for other aspects of thread-safety for your application. You must ensure that all the programs that your application uses are threadsafe, that you manage any shared storage in a threadsafe way, and that you only use those aspects of the system that are threadsafe.

Table 44. Comparison of thread-safety modes in RPG

Issue	THREAD(*CONCURRENT)	THREAD(*SERIALIZE)
Is source-modification required to achieve thread-safety (other than coding the THREAD keyword)?	No, although some source-code modification may be necessary to reduce the amount of static storage used by the module, if the number of concurrent threads might be very large.	No
Is there a deadlock risk due to the handling of static storage within the module?	Yes, if SERIALIZE is coded on a Procedure specification.	Yes, the risk is high. Deadlock is possible at a module level. If THREAD_A is in module MOD_1 and THREAD_B is in module MOD_2, and each thread is trying to call a procedure in the other module.
Does the module get the benefits of running multithreaded?	Yes	No
Is there a risk of bottlenecks?	Yes, if the SERIALIZE keyword is coded on a procedure specification.	Yes, the risk is high. The serialization of access to the module can cause the module to act as a bottleneck within the application. If one thread is active in the module, other threads must wait until the first thread is no longer running in the module, in any procedure.
Is thread-local storage supported?	Yes, it is the default type of static storage.	No
Is all-thread static storage supported?	Yes	Yes, it is the only supported type of static storage.
Can the RPG programmer choose whether a static variable is thread-local or shared by all threads?	Yes	No, only all-thread static storage is supported.
Is there a concern about the amount of memory required at runtime?	Possibly. The amount of static storage needed for the module is multiplied by the number of threads using the module.	No, all threads use the same static storage.

Table 44. Comparison of thread-safety modes in RPG (continued)

Issue	THREAD(*CONCURRENT)	THREAD(*SERIALIZE)
Who is the intended user?	An RPG programmer who wants the performance benefits of running in multiple threads, who is either willing to accept the amount of thread-local static storage used by each thread, and/or willing to rewrite the RPG module to use the least amount of static storage possible.	An RPG programmer who does not wish to rewrite the module to reduce the amount of static storage, or who is concerned about the additional storage per thread required by THREAD(*CONCURRENT). The RPG programmer is willing to accept the fact that the module can act as a bottleneck if more than one thread wants to run a procedure in the module at the same time.

For more information, see "Multithreading Considerations" on page 188.

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 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 that 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 that 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 you cannot code a bound procedure call in your source, nor can you use the BNDDIR or ACTGRP parameters on the CRTBNDRPG command when creating this program.

Example of OPM-Compatible Program

Figure 6 on page 72 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.

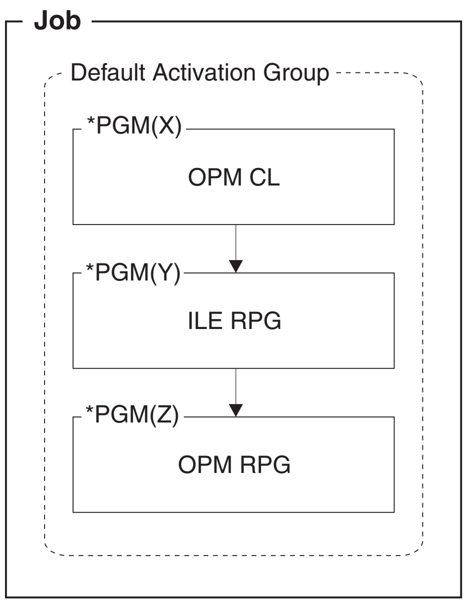


Figure 6. OPM-Compatible Application

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 that 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 428

One-step creation process
“Creating a Program with the CRTBNDRPG Command” on page 101

ILE static binding
“Calling Programs and Procedures” on page 159; also *ILE Concepts*

Exception handling differences
“Differences between OPM and ILE RPG Exception Handling” on page 294

Strategy 2: ILE Program Using CRTBNDRPG

Strategy 2 results in an ILE program that can take advantage of ILE static binding. Your source can contain static procedure calls 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 that 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 names of the binding directories, if any, to be used.

   It is assumed with this approach that if you are using a binding directory, it is one that is already created for you. For example, there may be a third-party service program that you may want to bind to 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 that 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 that are identified in a binding directory. If you want to bind two or more modules without using a binding directory when you create the program, you need to use the third strategy.

Example of ILE Program Using CRTBNDRPG

Figure 7 on page 74 shows the run-time view of an application in which an ILE CL program calls an ILE RPG program that is bound to a supplied service program. The application runs in the named activation group XYZ.

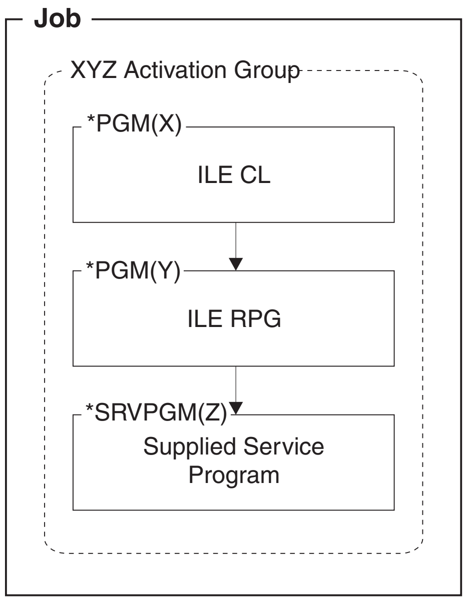


Figure 7. ILE Program Using CRTBNDRPG

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 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 that 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 main 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 that 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 428

One-step creation process
“Creating a Program with the CRTBNDRPG Command” on page 101

Activation groups
“Managing Activation Groups” on page 144

RCLRSC
“Reclaim Resources Command” on page 146

ILE static binding
“Calling Programs and Procedures” on page 159; also *ILE Concepts*

Exception handling differences
“Differences between OPM and ILE RPG Exception Handling” on page 294

Override and open scope
“Overriding and Redirecting File Input and Output” on page 336 and “Sharing an Open Data Path” on page 339; 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 74.

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 that 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 contain the procedure that 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. Identify the names of the binding directories, if any, to be used.

   It is assumed with this approach that if you are using a binding directory, it is one that is already created for you. For example, there may be a third-party service program that you may want to bind to your source. Consequently, all you need to know is the name of the binding directory.
5. Bind the appropriate modules and service programs together using CRTPGM, specifying values for the parameters based on the characteristics determined in step "2" on page 75.

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 that is called by an ILE RPG program. Figure 8 on page 76 shows the run-time view of this application.

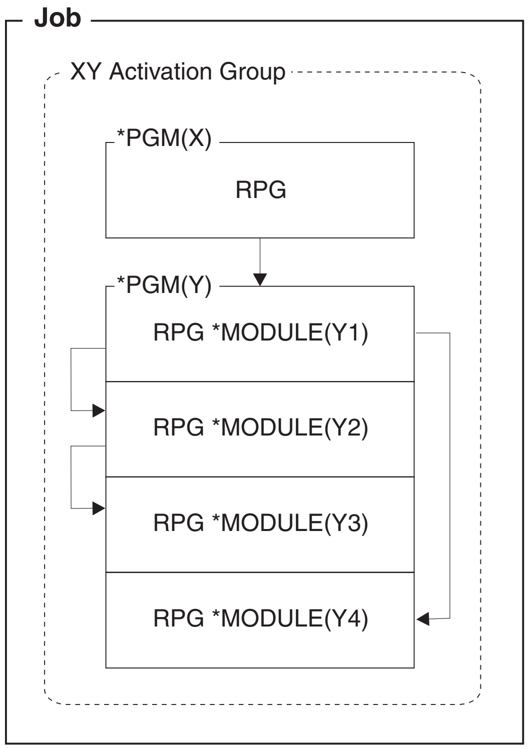


Figure 8. Single-Language Application Using CRTRPGMOD and CRTPGM

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 74 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 9 on page 77 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).

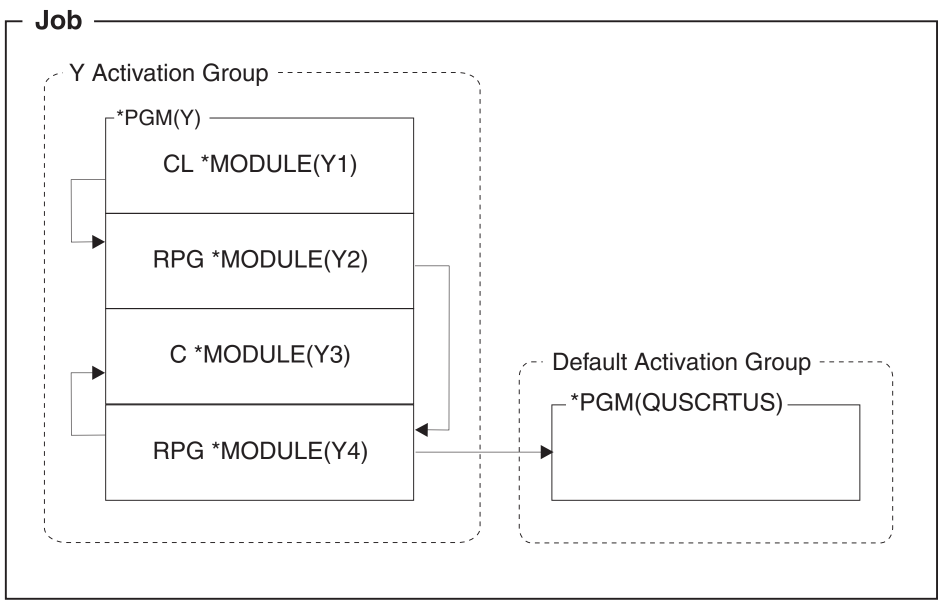


Figure 9. Mixed-Language Application

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 74 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 10 on page 77 shows an example in which an ILE program is bound to two service programs.

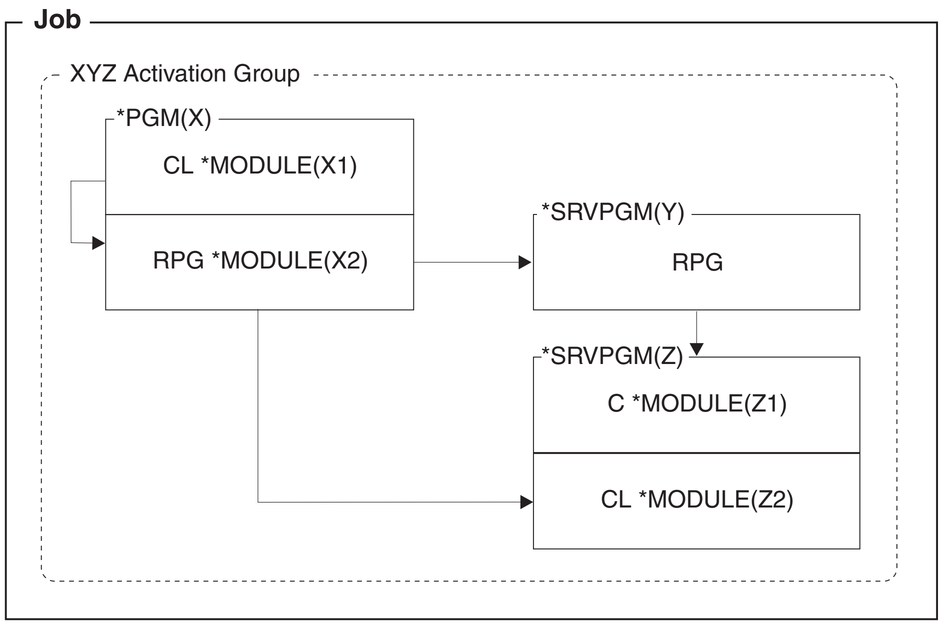


Figure 10. Advanced Application

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

See "Effect of ILE" on page 74 for details on the effects of ILE on the way your application handles calls, data, files and errors.

Related Information

Two-step creation process
"Creating a Program with the CRTRPGMOD and CRTPGM Commands" on page 115

Activation groups
"Managing Activation Groups" on page 144

ILE static binding
"Calling Programs and Procedures" on page 159; also *ILE Concepts*

Exception Handling
"Handling Exceptions" on page 290; also *ILE Concepts*

Service programs
"Creating a Service Program" on page 129; also *ILE Concepts*

Updating a Program
"Using the UPDPGM Command" on page 128

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 consisting of OPM and ILE programs is split across the OPM default activation group and a named activation group. In other words, try to avoid the scenario shown in Figure 11 on page 78.

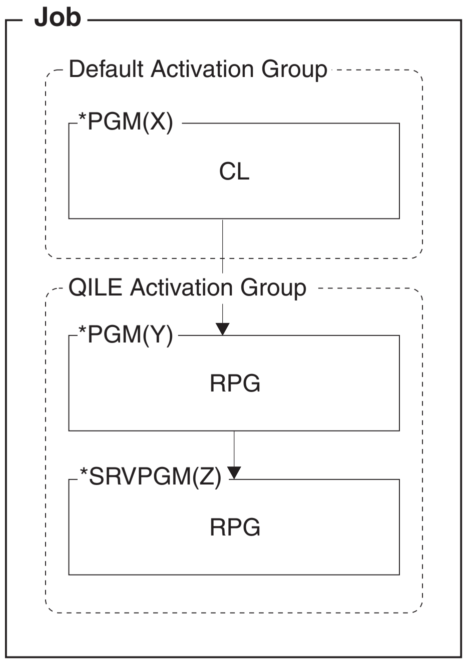


Figure 11. Scenario to Avoid

When an application is 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.

Note: Calling an ILE program from the command line, or from an OPM program that simply makes a call, is not a problem. The problems, which can all be solved, stem from OPM programs and ILE programs using shared resources such as overrides and commitment control, and from OPM programs trying to using OPM commands such as RCLRSC which have no effect on programs running in a named activation group.

Creating an Application Using Multiple Procedures

The ability to code more than one procedure in an ILE RPG module greatly enhances your ability to code a modular application. This chapter discusses why and how you might use such a module in your application. Specifically this chapter presents:

- Overview of key concepts
- Example of module with more than one procedure
- Coding considerations

Refer to the end of this section to see where to look for more detailed information on coding modules with multiple procedures.

A Multiple Procedures Module — Overview

An ILE program consists of one or more modules; a module is made up of one or more procedures.

1. A **procedure** is a self-contained unit of computation that is called using a bound call.
2. The RPG Compiler restricts the RPG programmer from calling a linear-main procedure with a bound call. Instead, the bound call to the linear-main procedure is made by the compiler-supplied **Program Entry Procedure (PEP)** of the program. The prototype for the linear-main procedure always uses the **EXTPGM** keyword, so calls using the prototype perform a program call.

Note: In the RPG documentation, the term 'procedure' refers to both main and subprocedures.

Main Procedures and Subprocedures

An ILE RPG module consists of zero or more subprocedures and optionally, a main procedure. A main procedure is a procedure that can be specified as the program entry procedure (and so receive control when an ILE program is first called). A cycle-main procedure can be defined in the main source section, which is the set of H, F, D, I, C, and O specifications that begin a module; a linear-main procedure can be specified in the subprocedure section and specially-designated with the MAIN keyword on a Control specification. For additional information about *procedures and the program logic cycle*, refer to the "WebSphere Development Studio ILE RPG Reference".

A **subprocedure** is a procedure that is specified after the main source section. A subprocedure differs from a main procedure primarily in that:

- Names that are defined within subprocedure are not accessible outside the subprocedure.
- No cycle code is generated for the subprocedure.
- The call interface must be prototyped.
- Calls to subprocedures must be bound procedure calls.
- Only P, F, D, and C specifications can be used.
- Other than being called through a program call rather than a bound call, a linear-main procedure is just like any other subprocedure.

Subprocedures can provide independence from other procedures because the data items are local. Local data items are normally stored in automatic storage, which means that the value of a local variable is not preserved between calls to the procedure.

Subprocedures offer another feature. You can pass parameters to a subprocedure by value, and you can call a subprocedure in an expression to return a value. Figure 12 on page 80 shows what a module might look like with multiple procedures.

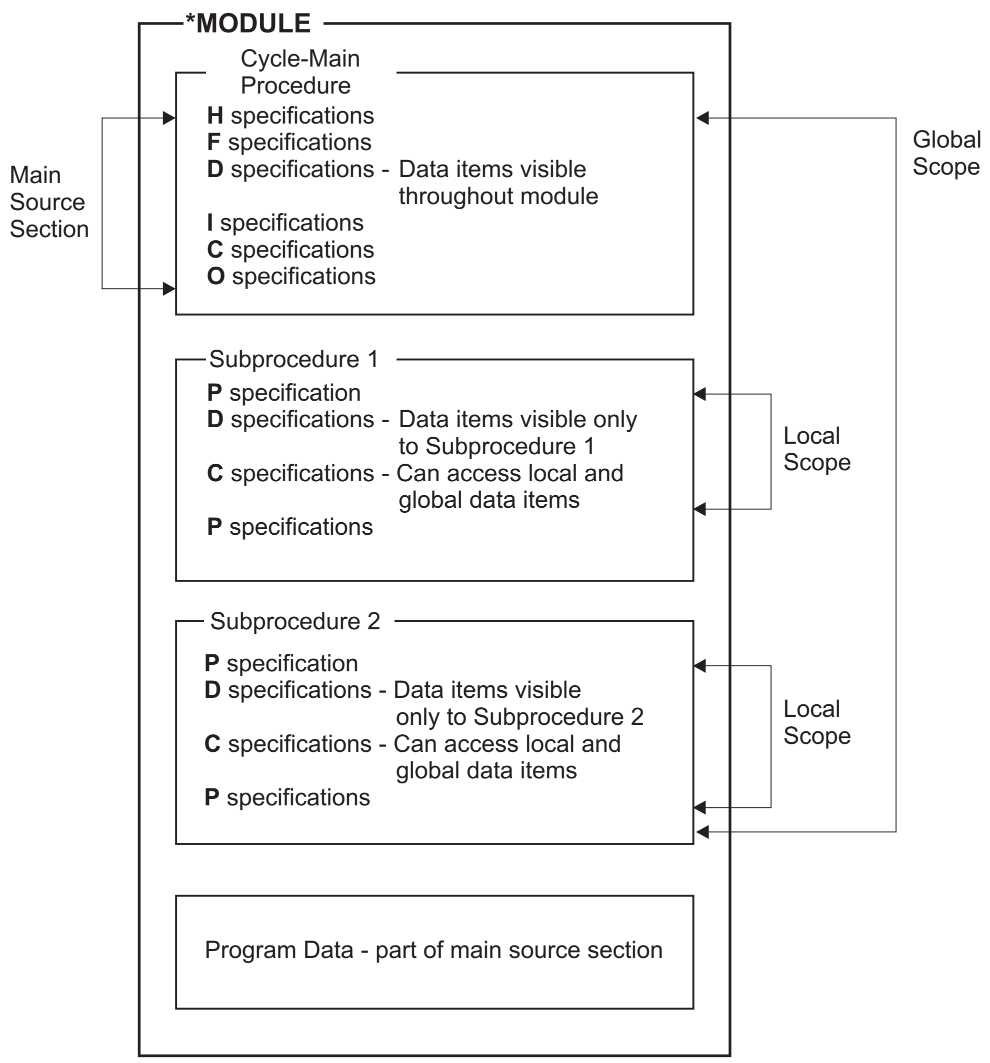

*MODULE
Cycle-Main Procedure
H specifications
F specifications
D specifications - Data items visible throughout module
I specifications
C specifications
O specifications
Main Source Section
Global Scope
Subprocedure 1
P specification
D specifications - Data items visible only to Subprocedure 1
C specifications - Can access local and global data items
P specifications
Local Scope
Subprocedure 2
P specification
D specifications - Data items visible only to Subprocedure 2
C specifications - Can access local and global data items
P specifications
Local Scope
Program Data - part of main source section

Figure 12. An ILE RPG Cycle-module with Multiple Procedures

As the picture suggests, you can now code subprocedures to handle particular tasks. These tasks may be needed by the main procedures or by other modules in the application. Furthermore, you can declare temporary data items in subprocedures and not have to worry if you have declared them elsewhere in the module.

Prototyped Calls

To call a subprocedure, you must use a prototyped call. You can also call any program or procedure that is written in any language in this way. A **prototyped call** is one where the call interface is checked at compile time through the use of a prototype. A **prototype** is a definition of the call interface. It includes the following information:

- Whether the call is bound (procedure) or dynamic (program)
- How to find the program or procedure (the external name)
- The number and nature of the parameters
- Which parameters must be passed, and which are optionally passed
- Whether operational descriptors are passed (for a procedure)
- The data type of the return value, if any (for a procedure)

The prototype is used by the compiler to call the program or procedure correctly, and to ensure that the caller passes the correct parameters. Figure 13 on page 81 shows a prototype for a procedure FmtCust, which formats various fields of a record into readable form. It has two output parameters.

```
      // Prototype for procedure FmtCust  (Note the PR on definition
      // specification.)  It has two output parameters.

     D FmtCust         PR
     D  Name                        100A
     D  Address                     100A
```

Figure 13. Prototype for FmtCust Procedure

To format an address, the application calls a procedure FmtAddr. FmtAddr has several input parameters, and returns a varying character field. Figure 14 on page 81 shows the prototype for FmtAddr.

```
      //---------------------------------------------------------
      // FmtAddr - procedure to produce an address in the form
      //---------------------------------------------------------
     D FmtAddr         PR           100A     VARYING
     D   streetNum                   10I 0   CONST
     D   streetName                  50A     CONST
     D   city                        20A     CONST
     D   state                       15A     CONST
     D   zip                          5P 0   CONST
```

Figure 14. Prototype for FmtAddr Procedure

If the procedure is coded in the same module as the call, specifying the prototype is optional. If the prototype is not specified, the compiler will generate the prototype from the procedure interface. However, if the procedure is exported and it is also called from another RPG module or program, a prototype should be specified in a copy file, and the copy file should be copied into both the calling module and the module that is exporting the procedure.

If the program or procedure is prototyped, you call it with CALLP or within an expression if you want to use the return value. You pass parameters in a list that follows the name of the prototype, for example, *name (parm1 : parm2 : …)*.

Figure 15 on page 81 shows a call to FmtCust. Note that the names of the output parameters, shown above in Figure 13 on page 81, do not match those in the call statement. The parameter names in a prototype are for documentation purposes only. The prototype serves to *describe* the attributes of the call interface. The actual definition of call parameters takes place inside the procedure itself.

```
     C                   CALLP     FmtCust(RPTNAME : RPTADDR)
```

Figure 15. Calling the FmtCust Procedure

Using prototyped calls you can call (with the same syntax):

- Programs that are on the system at run time
- Exported procedures in other modules or service programs that are bound in the same program or service program
- Subprocedures in the same module

FmtCust calls FmtAddr to format the address. Because FmtCust wants to use the return value, the call to FmtAddr is made in an expression. Figure 16 on page 82 shows the call.

```
         //--------------------------------------------------------------
         // Call the FmtAddr procedure to handle the address
         //--------------------------------------------------------------
         Address = FmtAddress (STREETNUM : STREETNAME :
                               CITY : STATE : ZIP);
```

Figure 16. Calling the FmtAddr Procedure

The use of procedures to return values, as in the above figure, allows you to write any user-defined function you require. In addition, the use of a prototyped call interface enables a number of options for parameter passing.

- Prototyped parameters can be passed in several ways: by reference, by value (for procedures only), or by read-only reference. The default method for RPG is to pass by reference. However, passing by value or by read-only reference gives you more options for passing parameters.
- If the prototype indicates that it is allowed for a given parameter, you may be able to do one or more of the following:
  - Pass *OMIT
  - Leave out a parameter entirely
  - Pass a shorter parameter than is specified (for character and graphic parameters, and for array parameters)

Example of Module with Multiple Procedures

Now let us look at an example of a multiple procedures module. In this 'mini-application' we are writing a program ARRSRPT to produce a report of all customers whose accounts are in arrears. We will create the basic report as a module, so that it can be bound to other modules, if necessary. There are two main tasks that are required for this module:

1. Determine that a record of an account from a customer file is in arrears.
2. Format the data into a form that is suitable for the report.

We have decided to code each task as a subprocedure. Conceptually, the module will look something like that shown in Figure 17 on page 82.

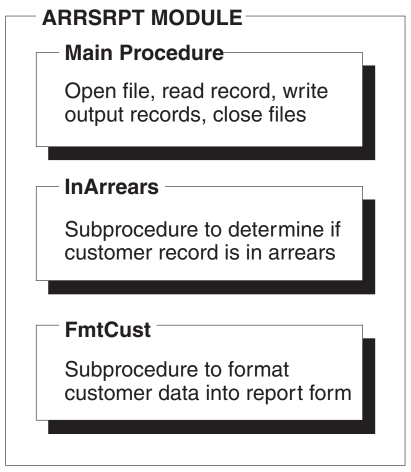

ARRSRPT MODULE
Main Procedure
Open file, read record, write output records, close files
InArrears
Subprocedure to determine if customer record is in arrears
FmtCust
Subprocedure to format customer data into report form

Figure 17. Components of the ARRSRPT Module

Now consider the first subprocedure, InArrears, which is shown in Figure 18 on page 83. InArrears is called by the main procedure to determine if the current record is in arrears.

TIP

When coding subprocedures that use global fields, you may want to establish a naming convention that shows the item to be global. In this example, the uppercase field names indicate DDS fields. Another option would be to prefix 'g_', or some other string to indicate global scope.

If the record is in arrears, the subprocedure returns '1' to the main procedure.

```
 //-------------------------------------------------------------
 // InArrears
 //
 // Parameters: (none)
 // Globals:    DUEDATE, AMOUNT, CurDate
 //
 // Returns:   '1' if the customer is in arrears
 //-------------------------------------------------------------

P InArrears       B                     1
D InArrears       PI             1A     2
 // Local declarations

D DaysLate        S             10I 0   3
D DateDue         S               D     3
 // Body of procedure

 /free
     DateDue = %date (DUEDATE: *ISO);
     DaysLate = %diff (CurDate: DateDue: *d);

     // The data in the input file comes from another type
     // of computer, and the AMOUNTC field is a character
     // string containing the numeric value.  This string
     // must be converted to the numeric AMOUNT field
     // for printing.


     AMOUNT = %dec(AMOUNTC : 31 : 9);
     if DaysLate > 60 AND AMOUNT > 100.00;
        return '1';                     4
     endif;
     return '0';                        4   5
 /end-free
P InArrears       E                     1
```

Figure 18. Source for Subprocedure InArrears

Figure 18 on page 83 shows the main elements that are common to all subprocedures.

1
All subprocedures begin and end with procedure specifications.

2
After the Begin-Procedure specification (B in position 24 of the procedure specification), you code a procedure interface definition. The return value, if any, is defined on the PI specification. Any parameters are listed after the PI specification.

3
Any variables or prototypes that are used by the subprocedure are defined after the procedure interface definition.

4
The return value, if specified, is returned to the caller with a RETURN operation.

5
If the record is not in arrears, the subprocedure returns '0' to the main procedure.

For all subprocedures, and also for a cycle-main procedure with prototyped entry parameters, you need to define a procedure interface. A **procedure interface definition** is a repeat of the prototype information, if the prototype was specified, within the definition of a procedure. It is used to define the entry parameters for the procedure. The procedure interface definition is also used to ensure that the internal definition of the procedure is consistent with the external definition (the prototype). When the prototype is not specified, the compiler generates the prototype from the procedure interface, so the procedure interface definition provides both the internal definition and the external definition. In the case of InArrears, there are no entry parameters.

Consider next the subprocedure FmtCust, which is shown in Figure 19 on page 84. FmtCust is called by ARRSRPT to format the relevant fields of a record into an output record for the final report. (The record represents an account that is in arrears.) FmtCust uses global data, and so does not have any input parameters. It formats the data into two output fields: one for the name, and one for the address.

```
 //-------------------------------------------------------------
 // FmtCust formats CUSTNAME, CUSTNUM, STREETNAME etc into
 // readable forms
 //
 // Parameters:   Name     (output)
 //               Address  (output)
 // Globals:      CUSTNAME, CUSTNUM, STREETNUM, STREETNAME, CITY
 //               STATE, ZIP
 //-------------------------------------------------------------



P FmtCust         B
D FmtCust         PI
D  Name                        100A
D  Address                     100A

 /free

     //-------------------------------------------------------------
     // CUSTNAME and CUSTNUM are formatted to look like this:
     // A&P Electronics     (Customer number 157)
     //-------------------------------------------------------------

     Name = CUSTNAME + ' ' + '(Customer number '
                       + %char(CUSTNUM) + ')';

     //-------------------------------------------------------------
     //   Call the FmtAddr procedure to handle the address
     //-------------------------------------------------------------

 Address = FmtAddress (STREETNUM : STREETNAME :
                             CITY : STATE : ZIP);
 /end-free
P FmtCust         E
```

Figure 19. Source for Subprocedure FmtCust

Finally, consider the last subprocedure of this application, FmtAddr. Notice that FmtAddr does not appear in the ARRSRPT module, that is shown in Figure 17 on page 82. We decided to place FmtAddr inside another module called FMTPROCS. FMTPROCS is a utility module that will contain any conversion procedures that other modules might need to use.

Figure 20 on page 85 shows the source of the module FMTPROCS. Since procedure FmtAddr is called from another module, a prototype is required. So that the prototype can be shared, we have placed the prototype into a /COPY file which is copied into both the calling module, to provide information about how to call the procedure, and into the module that defines the procedure, to ensure that the prototype matches the procedure interface.

```
     //=================================================================
     // Source for module FMTPROCS.  This module does not have a
     // main procedure, as indicated by the keyword NOMAIN.
     //=================================================================
    H NOMAIN
     //-----------------------------------------------------------------
     // The prototype must be available to EACH module containing
     // a prototyped procedure.  The /COPY pulls in the prototype
     // for FmtAddr.
     //-----------------------------------------------------------------
    D/COPY QRPGLESRC,FMTPROC_P
    P FmtAddr         B                     EXPORT
    D FmtAddr         PI           100A     VARYING
    D   streetNum                   10I 0   CONST
    D   streetName                  50A     CONST
    D   city                        20A     CONST
    D   state                       15A     CONST
    D   zip                          5P 0   CONST
     /free

         //-------------------------------------------------------------
         // STREETNUM, STREETNAME, CITY, STATE, and ZIP are formatted to
         // look like:
         //   27 Garbanzo Avenue, Smallville IN 51423
         //-------------------------------------------------------------
         return  %char(streetNum) + ' ' + %trimr(streetName)
                        + ', ' + %trim(city) + ' ' + %trim(state)
                       + ' ' + %editc(zip : 'X');
    P FmtAddr         E
```

Figure 20. Source for module FMTPROCS, containing subprocedure FmtAddr.

FMTPROCS is a **NOMAIN module**, meaning that it consists only of subprocedures; there is no main procedure. A NOMAIN module compiles faster and requires less storage because there is no cycle code that is created for the module. You specify a NOMAIN module, by coding the NOMAIN keyword on the control specification. For more information on NOMAIN modules, see "Program Creation" on page 89.

The Entire ARRSRPT Program

The ARRSRPT program consists of two modules: ARRSRPT and FMTPROCS. Figure 21 on page 85 shows the different pieces of our mini-application.

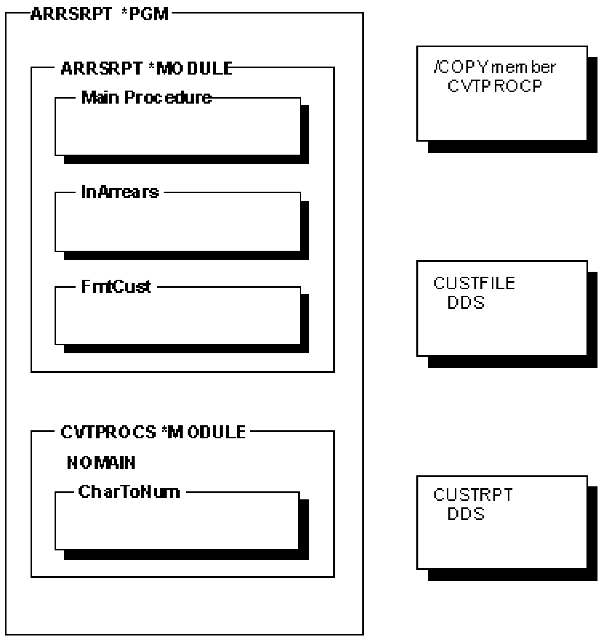


Figure 21. The ARRSRPT Application

Figure 22 on page 87 shows the source for the entire ARRSRPT module.

```
  //=================================================================
  // Source for module ARRSRPT.  Contains a cycle-main procedure and
  // two subprocedures: InArrears and FmtCust.
  //
  // Related Module:  CVTPROCS  (CharToNum called by InArrears)
  //=================================================================
  //--------------------------------------------------------------
  // F I L E S
  //
  // CUSTFILE - contains customer information
  // CUSTRPT  - printer file (using format ARREARS)
  //--------------------------------------------------------------

 FCUSTFILE  IP   E             DISK
 FCUSTRPT   O    E             PRINTER
  *--------------------------------------------------------------*
  * P R O T O T Y P E S
  *--------------------------------------------------------------*

  /COPY QRPGLE,FMTPROC_P
  *--------------------------------------------------------------*
  * InArrears returns '1' if the customer is in arrears
  *--------------------------------------------------------------*

 D InArrears       PR             1A
  *--------------------------------------------------------------*
  * FmtCust formats CUSTNAME, CUSTNUM, STREETNAME etc into
  * readable forms
  *--------------------------------------------------------------*

 D FmtCust         PR
 D  Name                        100A
 D  Address                     100A
```

```
  *--------------------------------------------------------------*
  * G L O B A L   D E F I N I T I O N S
  *--------------------------------------------------------------*

 D CurDate         S               D
 ICUSTREC       01
  *--------------------------------------------------------------*
  * M A I N   P R O C E D U R E
  *--------------------------------------------------------------*

 C                   IF        InArrears() = '1'
 C                   CALLP     FmtCust(RPTNAME : RPTADDR)
 C                   EVAL      RPTNUM = CUSTNUM
 C                   WRITE     ARREARS
 C                   ENDIF
 C     *INZSR        BEGSR
 C                   MOVEL     UDATE         CurDate
 C                   ENDSR

   *--------------------------------------------------------------*
   * S U B P R O C E D U R E S
   *--------------------------------------------------------------*
```

```
   //--------------------------------------------------------------
   // InArrears
   //
   // Parameters: (none)
   // Globals:    DUEDATE, AMOUNT, CurDate
   //
   // Returns:   '1' if the customer is in arrears
   //--------------------------------------------------------------
 P InArrears       B
 D InArrears       PI             1A
   // Local declarations
 D DaysLate        S             10I 0
 D DateDue         S               D
   // Body of procedure
  /free
      DateDue = %date (DUEDATE: *ISO);
      DaysLate = %diff (CurDate: DateDue: *d);
       // The data in the input file comes from another type
        // of computer, and the AMOUNTC field is a character
        // string containing the numeric value.  This string
        // must be converted to the numeric AMOUNT field
        // for printing.
       AMOUNT = %dec(AMOUNTC : 31 : 9);
      if DaysLate > 60 AND AMOUNT > 100.00;
         return '1';
      endif;
      return '0';
```

Note the following about ARRSRPT:

- The definition specifications begin with the prototypes for the prototyped calls. A /COPY file is used to supply the prototype for the called procedure FmtAddr.

  The prototypes do not have to be first, but you should establish an order for the different types of definitions for consistency.
- The date field CurDate is a global field, meaning that any procedure in the module can access it.
- The main procedure is simple to follow. It contains calculation specifications for the two main tasks: the I/O, and an initialization routine.
- Each subprocedure that follows the main procedure contains the details of one of the tasks.

Sample output for the program ARRSRPT is shown in Figure 23 on page 88.

```
      Customer number: 00001
          ABC Electronics      (Customer number 1)
          15 Arboreal Way, Treetop MN 12345
          Amount outstanding:          $1234.56   Due date: 1995-05-01

      Customer number: 00152
          A&P Electronics      (Customer number 152)
          27 Garbanzo Avenue, Smallville MN 51423
          Amount outstanding:         $26544.50   Due date: 1995-02-11
```

Figure 23. Output for ARRSRPT

Figure 24 on page 88 and Figure 25 on page 89 show the DDS source for the files CUSTFILE and CUSTRPT respectively.

```
      A*================================================================*
      A* FILE NAME     : CUSTFILE
      A* RELATED PGMS  : ARRSRPT
      A* DESCRIPTIONS  : THIS IS THE PHYSICAL FILE CUSTFILE.  IT HAS
      A*                 ONE RECORD FORMAT CALLED CUSTREC.
      A*================================================================*
      A* CUSTOMER MASTER FILE -- CUSTFILE

      A          R CUSTREC
      A            CUSTNUM        5  0       TEXT('CUSTOMER NUMBER')
      A            CUSTNAME      20          TEXT('CUSTOMER NAME')
      A            STREETNUM      5  0       TEXT('CUSTOMER ADDRESS')
      A            STREETNAME    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            AMOUNTC       15          TEXT('AMOUNT OUTSTANDING')
      A            DUEDATE       10          TEXT('DATE DUE')
```

Figure 24. DDS for CUSTFILE

```
     A*================================================================*
     A* FILE NAME     : CUSTRPT
     A* RELATED PGMS  : ARRSRPT
     A* DESCRIPTIONS  : THIS IS THE PRINTER FILE CUSTRPT.  IT HAS
     A*                 ONE RECORD FORMAT CALLED ARREARS.
     A*================================================================*

     A          R ARREARS
     A                               2  6
     A                                   'Customer number:'
     A            RPTNUM         5  0   2 23
     A                                   TEXT('CUSTOMER NUMBER')
     A            RPTNAME      100A     3 10
     A                                   TEXT('CUSTOMER NAME')
     A            RPTADDR      100A     4 10
     A                                   TEXT('CUSTOMER ADDRESS')
     A                               5 10'Amount outstanding:'
     A            AMOUNT        10  2   5 35EDTWRD('       $0.  ')
     A                                   TEXT('AMOUNT OUTSTANDING')
     A                               5 50'Due date:'
     A            DUEDATE       10      5 60
     A                                   TEXT('DATE DUE')
```

Figure 25. DDS for CUSTRPT

Coding Considerations

This section presents some considerations that you should be aware of before you begin designing applications with multiple-procedure modules. The items are grouped into the following categories:

- General
- Program Creation
- Main Procedures
- Subprocedures

General Considerations

- When coding a module with multiple procedures, you will want to make use of /COPY files, primarily to contain any prototypes that your application may require. If you are creating a service program, you will need to provide both the service program and the prototypes, if any.
- Maintenance of the application means ensuring that each component is at the most current level and that any changes do not affect the different pieces. You may want to consider using a tool such as Application Development Manager to maintain your applications.

  For example, suppose that another programmer makes a change to the /COPY file that contains the prototypes. When you request a rebuild of your application, any module or program that makes use of the /COPY file will be recompiled automatically. You will find out quickly if the changes to the /COPY file affect the calls or procedure interfaces in your application. If there are compilation errors, you can then decide whether to accept the change to prototypes to avoid these errors, or whether to change the call interface.

Program Creation

- If you specify that a module does not have a main procedure then you cannot use the CRTBNDRPG command to create the program. (A module does not have a main procedure if the NOMAIN keyword is specified on a control specification.) This is because the CRTBNDRPG command requires that the module contain a program entry procedure and only a main procedure can be a program entry procedure.
- Similarly, when using CRTPGM to create the program, keep in mind that a NOMAIN module cannot be an entry module since it does not have a program entry procedure.
- A program that is created to run in the default OPM activation group (by specifying DFTACTGRP(*YES) on the CRTBNDRPG command) cannot contain bound procedure calls.

Main Procedure Considerations

You cannot define return values for a main procedure, nor can you specify that its parameters be passed by value.

The following considerations apply only to a cycle-main procedure:

- Because the cycle-main procedure is the only procedure with a complete set of specifications available (except the P specification), it should be used to set up the environment of all procedures in the module.
- A cycle-main procedure is always exported, which means that other procedures in the program can call the main procedure by using bound calls.
- The call interface of a cycle-main procedure can be defined in one of two ways:
  1. Using a procedure interface and an optional prototype
  2. Using an *ENTRY PLIST without a prototype
- The functionality of an *ENTRY PLIST is similar to a prototyped call interface. However, a prototyped call interface is much more robust since it provides parameter checking at compile time. If you prototype the main procedure, then you specify how it is to be called by specifying either the EXTPROC or EXTPGM keyword on the prototype definition. If EXTPGM is specified, then an external program call is used; if EXTPROC is specified or if neither keyword is specified, it will be called by using a procedure call.

Subprocedure Considerations

These considerations apply to ordinary subprocedures and linear-main procedures except as otherwise noted.

- Any of the calculation operations may be coded in a subprocedure. However, input and output specifications are not supported in subprocedures, so data structure result fields must be used for file I/O operations to files defined locally in the subprocedure. All data areas must be defined in the main source section, although they can be used in a subprocedure.
- The control specification can only be coded in the main source section since it controls the entire module.
- A subprocedure can be called recursively.Each recursive call causes a new invocation of the procedure to be placed on the call stack. The new invocation has new storage for all data items in automatic storage, and that storage is unavailable to other invocations because it is local. (A data item that is defined in a subprocedure uses automatic storage unless the STATIC keyword is specified for the definition.)

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

  Recursion can be a powerful programming technique when properly understood.
- The run-time behavior of a subprocedure (including a linear-main procedure) differs somewhat from that of a cycle-main procedure, because there is no cycle code for the subprocedure.
  - When a subprocedure ends, any open local files in automatic storage are closed. However, none of the termination activities, such as closing of global files, occurs until the cycle-main procedure, if any, that are associated with the subprocedure itself ends. If you want to ensure that your global files are closed before the activation group ends, you can code a "cleanup" subprocedure that is called both by the program entry procedure at application-end, and by a cancel handler enabled for the program entry procedure.

    An alternative to using a cleanup procedure is to code the module so that there is no implicit file opening or data area locking, and that within any subprocedure, an open is matched by a close, an IN by an OUT, a CRT by a DLT, and so on. This alternative should be strongly considered for a cycle-module if it might have a subprocedure active when the cycle-main procedure is not active.

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

For Further Information

To find out more about the topics discussed here, consult the following list:

Main Procedures

Topic	See
Exception handling	"Exception Handling within a Cycle-Main Procedure" on page 293
Main Procedure End	"Returning from a Main Procedure" on page 183

Subprocedures

Topic	See
Defining	Chapter on subprocedures, in the *IBM Rational Development Studio for i: ILE RPG Reference*
NOMAIN module	"Creating a NOMAIN Module" on page 118
Exception handling	"Exception Handling within Subprocedures" on page 294
Procedure Specification	Chapter on procedure specifications, in the *IBM Rational Development Studio for i: ILE RPG Reference*
Procedure Interface	Chapter on defining data and prototypes in the *IBM Rational Development Studio for i: ILE RPG Reference*
Subprocedure End	"Returning from a Subprocedure" on page 185

Prototyped Call

Topic	See
Free-form call	"Using a Prototyped Call" on page 165
General Information	*IBM Rational Development Studio for i: ILE RPG Reference*, Chapter 24
Passing parameters	"Passing Prototyped Parameters" on page 167
Prototypes	Chapter on defining data and prototypes in the *IBM Rational Development Studio for i: ILE RPG Reference*

Chapter 4. Creating and Running an ILE RPG Application

This section provides you with the information that is needed to create and run ILE RPG 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 Integrated Language Environment terms and concepts are discussed briefly in the following pages. These terms and concepts are more fully discussed in *ILE Concepts*.

Using Source Files

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 Remote Systems LPEX Editor.

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 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 compiler will convert most of the source to uppercase when it compiles it. It will not convert literals, array data or table data.

Your source can be in two different kinds of files:

1. Source physical files
2. IFS (Integrated File System) files

Using Source Physical Files

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 structure as shown in Figure 26 on page 94.

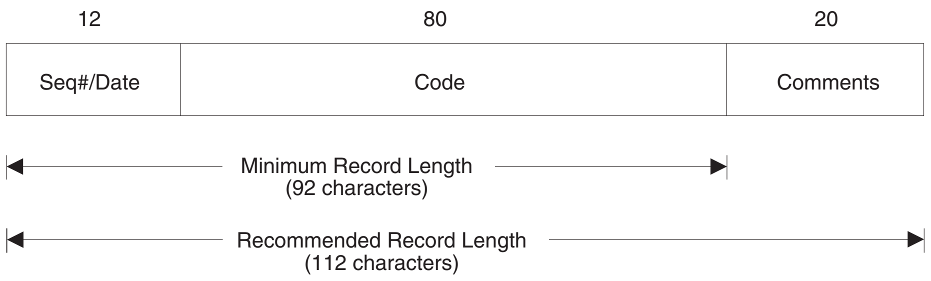

12
80
20
Seq#/Date
Code
Comments
Minimum Record Length (92 characters)
Recommended Record Length (112 characters)

Figure 26. ILE RPG Record Length Breakdown

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

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

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 the *ADTS for AS/400: Source Entry Utility* manual.

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 ILE 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 27 on page 95. Note that the screen is automatically shifted so that position 6 is (for specification type) 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 27. 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
   - IPD — for definition 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
   - IPPR — for procedure specifications

```
 *===============================================================*
 * MODULE NAME:    EMPRPT
 * RELATED FILES:  EMPMST   (PHYSICAL FILE)
 *                 QSYSPRT  (PRINTER FILE)
 * DESCRIPTION:    This program prints employee information
 *                 from the file EMPMST.
 *===============================================================*
FQSYSPRT   O    F   80        PRINTER
FEMPMST    IP   E           K DISK
D TYPE            S              8A
D EMPTYPE         PR             8A
D   CODE                         1A
IEMPREC        01
C                   EVAL      TYPE = EMPTYPE(ETYPE)
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    01
O                       ENAME              20
O                       ENUM               32
O                       EDEPT              45
O                       TYPE               60
 * Procedure EMPTYPE returns a string representing the employee
 *   type indicated by the parameter CODE.
P EMPTYPE         B
D EMPTYPE         PI             8A
D   CODE                         1A
C                   SELECT
C                   WHEN      CODE = 'M'
C                   RETURN    'Manager'
C                   WHEN      CODE = 'R'
C                   RETURN    'Regular'
C                   OTHER
C                   RETURN    'Unknown'
C                   ENDSL
P EMPTYPE         E
```

Figure 28. 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 29 on page 96 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 29. DDS for EMPRPT

To create a program from this source use the CRTBNDRPG command, specifying DFTACTGRP(*NO).

Using SQL Statements

The DB2® UDB for iSeries® database can be accessed from an ILE RPG 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 in free form or in fixed form
- In free form
  - You start your SQL statement using the delimiter "EXEC SQL".
  - The SQL statement can be placed on several lines. No continuation character is required.
  - Use a semicolon to signal the end of your SQL statement.
- In fixed form
  - Start your SQL statements using the delimiter/EXEC SQL in positions 7-15 (with the/in position 7)
  - 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.
- You can start entering your SQL statements on the same line as the starting delimiter

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

Figure 30 on page 97 and Figure 31 on page 98 show examples of embedded SQL statements.

```
...+....1....+....2....+....3....+....4....+....5....+....6....+....7..
     /FREE
         X = Y + Z; // ILE RPG calculation operations

         // The entire SQL statement is on one line
         EXEC SQL INSERT INTO MYLIB/MYFILE (FLD1) VALUES(12);

         // The SQL statement begins on the same line as
         // EXEC SQL and then it is is split across several lines
         EXEC SQL   INSERT
                       INTO MYLIB/MYFILE
                       (FLD1) VALUE(12);

         // The SQL statement begins on the line after
         // EXEC SQL and then it is is split across several lines
         EXEC SQL
             INSERT INTO MYLIB/MYFILE
                      (FLD1) VALUE(12);

         X = Y + Z; // ILE RPG calculation operations
     /END-FREE
```

Figure 30. Free Form SQL Statements in an ILE RPG Program

```
...+....1....+....2....+....3....+....4....+....5....+....6....+....7..
      * ILE RPG calculation operations
     C                   EVAL      X = Y + Z

      * The entire SQL statement is on one line
     C/EXEC SQL INSERT INTO MYLIB/MYFILE (FLD1) VALUES(12)
     C/END-EXEC

      * The SQL statement begins on the same line as
      * EXEC SQL and then it is is split across several lines
     C/EXEC SQL   INSERT
     C+             INTO MYLIB/MYFILE
     C+                 (FLD1) VALUE(12)
     C/END-EXEC

      * The SQL statement begins on the line after
      * EXEC SQL and then it is is split across several lines
     C/EXEC SQL
     C+      INSERT INTO MYLIB/MYFILE
     C+                (FLD1) VALUE(12)
     C/END-EXEC

      * ILE RPG calculation operations
     C                   EVAL      X = Y + Z
     C
```

Figure 31. Fixed Form SQL Statements in an ILE RPG Program

You must enter a separate command to process the SQL statements. For more information, refer to the *Db2® for i* section of the *Database and File Systems* category in the **IBM i Information Center** at this Web site - http://www.ibm.com/systems/i/infocenter/.

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

Using IFS Source Files

The CRTBNDRPG and CRTRPGMOD commands include parameters to allow the source files to be either in the QSYS file system or the Integrated File System. These are:

SRCSTMF
SRCSTMF is used instead of SRCFILE and SRCMBR to indicate a stream file is the main source file.

INCDIR
INCDIR is used to list the directories that will contain copy files.

The stream file specified for the SRCSTMF can be an absolute path to the file (beginning with a slash), or it can be a path relative to the current directory.

Include files

The /COPY and /INCLUDE directives allow the specification of files in either the QSYS file system or the IFS. In cases where the compiler cannot tell which file system the directive refers to, the search will begin in the file system of the file containing the /COPY directive.

When the compiler encounters a /COPY statement, the statement could refer to a file in the IFS or in the QSYS file system. If the name begins with a slash or is specified in single quotes, the name can only refer to a file in the IFS. A name in the IFS can be specified in double quotes as well. Where only part of the name is in double quotes, for example:

```
/copy "SOME-LIB"/QRPGLESRC,MBR
```

the name can only be a QSYS file system name.

If the name could be either in the QSYS file system or the IFS, the file system of the file containing the /COPY statement will be searched first. Note that upper-casing occurs for the QSYS file system (except with extended names specified with double quotes, such as "A/B") but not for the IFS. (The IFS is not case sensitive.)

Table 45. /Copy File Name Intepretation for QSYS and IFS

/Copy statement	QSYS interpretation	IFS interpretation (see below for the meaning of ".suffix")
/COPY MYMBR	FILE(*LIBL/QRPGLESRC) MBR(MYMBR)	MYMBR or MYMBR.suffix in one of the directories in the include path
/COPY mymbr	FILE(*LIBL/QRPGLESRC) MBR(MYMBR)	mymbr or mymbr.suffix in one of the directories in the include path
/COPY myfile,mymbr	FILE(*LIBL/MYFILE) MBR(MYMBR)	myfile,mymbr or myfile,mymbr.suffix (note that MYFILE,MYMBR is a valid name in the Integrated File System)
/COPY mylib/myfile,mymbr	FILE(MYLIB/MYFILE) MBR(MYMBR)	mylib/myfile,mymbr (directory mylib and file myfile,mymbr)
/COPY "A/b",mymbr	FILE(*LIBL/"A/b") MBR(MYMBR)	n/a (only part of name is in double quotes
/COPY "A/B"	FILE(*LIBL/QRPGLESRC) MBR("A/B")	A/B
/COPY a b	FILE(*LIBL/QRPGLESRC) MBR(A) (everything after a blank is assumed to be a comment)	a or a.suffix (everything after a blank is assumed to be a comment)
/COPY 'a b'	N/A (name in single quotes)	a b or a b.suffix
/COPY /home/mydir/myfile.rpg	N/A (name begins with slash)	/home/mydir/myfile.rpg
/COPY /QSYS.LIB/ L.LIB/F.FILE/M.MBR	N/A (name begins with slash)	/QSYS.LIB/L.LIB/F.FILE/ M.MBR (which is actually a file in the QSYS file system, but is considered to be an IFS file by RPG)

Note: When searching for files in the IFS, if the file name does not contain a dot, the RPG compiler will look for files with the following suffixes (in this order):

1. no suffix (abc)
2. .rpgleinc (abc.rpgleinc)
3. .rpgle (abc.rpgle)

Search Path Within The IFS

You have two ways to indicate where /COPY and /INCLUDE files can be found in the IFS:

1. The INCDIR parameter, which lists the directories in the order you want them to be searched.

2. The RPGINCDIR environment variable, which has a colon-separated list of directores in the order you want them to be searched. To set the environment variable, use the ADDENVVAR or CHGENVVAR command.

   For Example: `ADDENVVAR ENVVAR(RPGINCDIR) VALUE('/home/mydir:/project/ prototypes')ADDENVVAR`

When searching for a `relative` file in the IFS (one whose path does not begin with /), the file will be searched for in the following places, in this order

1. The current directory.
2. The path specified by the INCDIR comand parameter.
3. The directories in the RPGINCDIR environment variable.
4. The source directory (if the source is an IFS file).

For example, if:

- The current directory is /home/auser.
- The INCDIR parameter is /driver/r1.2/inc:/driver/r1.1/inc.
- The RPGINCDIR environment variable is /home/auser/temp.
- The source is in directory /home/auser/src.

The directory search path takes precedence over the default-suffix order. If a file with no extension is searched for in several different directories, all suffixes will be tried in each directory before the next directory is tried.

Table 46. Search Order for /Copy Files

/Copy statement	Files searched for
Assume the source file containing the /COPY is /driver/src/main.rpg, in the IFS /COPY file.rpg	In IFS: /home/auser/file.rpg /driver/r1.2/inc/file.rpg /driver/r1.1/inc/file.rpg /home/auser/temp/file.rpg /home/auser/src/file.rpg In QSYS: FILE(*LIBL/QRPGLESRC) MBR(FILE.RPG)

Table 46. Search Order for /Copy Files (continued)

/Copy statement	Files searched for
Assume the source file containing the /COPY is MYLIB/QRPGLESRC MYMBR, in the QSYS file system /COPY file	In QSYS: FILE(*LIBL/QRPGLESRC) MBR(FILE) In IFS: /home/auser/file /home/auser/file.rpgleinc /home/auser/file.rpgle /driver/r1.2/inc/file /driver/r1.2/inc/file.rpgleinc /driver/r1.2/inc/file.rpgle /driver/r1.1/inc/file /driver/r1.1/inc/file.rpgleinc /driver/r1.1/inc/file.rpgle /home/auser/temp/file /home/auser/temp/file.rpgleinc /home/auser/temp/file.rpgle /home/auser/src/file /home/auser/src/file.rpgleinc /home/auser/src/file.rpgle

Creating a Program with the CRTBNDRPG Command

This chapter shows you 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:

1. OPM-compatible programs with no static binding
2. Single-module ILE programs 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 that 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. The inability to have static binding means that you cannot:

- Use the CALLB operation in your source
- Call a prototyped procedure
- 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 call procedures from 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 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 "Creating a Program with the CRTRPGMOD and CRTPGM Commands" on page 115.

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).

"CRTBNDRPG Parameters and Their Default Values Grouped by Function" on page 102 summarizes the parameters of the CRTBNDRPG command and shows their default values.

CRTBNDRPG Parameters and Their Default Values Grouped by Function

Table 47. Program Identification

Parameter	Description
PGM(*CURLIB/*CTLSPEC)	Determines created program name and library
SRCFILE(*LIBL/QRPGLESRC)	If specified, identifies source file and library
SRCMBR(*PGM)	If specified, identifies file member containing source specifications
SRCSTMF(path)	If specified, indicates the path to the source file in the IFS
INCDIR('path to directory 1:path to directory 2')	Identifies a list of directories to search for /copy and /include files
TEXT(*SRCMBRTXT)	Provides brief description of program

Table 48. Program Creation

Parameter	Description
GENLVL(10)	Conditions program creation to error severity (0-20)
OPTION(*DEBUGIO)	*DEBUGIO/*NODEBUGIO, determines if breakpoints are generated for input and output specifications
OPTION(*GEN)	*GEN/*NOGEN, determines if program is created
OPTION(*NOSRCSTMT)	Specifies how the compiler generates statement numbers for debugging
OPTION(*UNREF)	*UNREF/*NOUNREF Determines whether unreferenced fields are placed in the program object
DBGVIEW(*STMT)	Specifies type of debug view, if any, to be included in program

Table 48. Program Creation (continued)

Parameter	Description
DBGENCKEY(*NONE)	Specifies the encryption for the listing debug view for the program
OPTIMIZE(*NONE)	Determines level of optimization, if any
REPLACE(*YES)	Determines if program should replace existing program
BNDDIR(*NONE)	Specifies the binding directory to be used for symbol resolution
USRPRF(*USER)	Specifies the user profile that will run program
AUT(*LIBCRTAUT)	Specifies type of authority for created program
TGTRLS(*CURRENT)	Specifies the release level the object is to be run on
ENBPFRCOL(*PEP)	Specifies whether performance collection is enabled
DEFINE(*NONE)	Specifies condition names that are defined before the compilation begins
PRFDTA(*NOCOL)	Specifies the program profiling data attribute
STGMDL(*SNGLVL)	Specifies the storage model for the program

Table 49. Compiler Listing

Parameter	Description
OUTPUT(*PRINT)	Determines if there is a compiler listing
INDENT(*NONE)	Determines if indentation should show in listing, and identifies character for marking it
OPTION(*XREF *NOSECLVL *SHOWCPY *EXPDDS *EXT *NOSHOWSKP *NOSRCSTMT)	Specifies the contents of compiler listing

Table 50. Data Conversion Options

Parameter	Description
CVTOPT(*NONE)	Specifies how various data types from externally described files are handled
ALWNULL(*NO)	Determines if the program will accept values from null-capable fields
FIXNBR(*NONE)	Determines which decimal data that is not valid is to be fixed by the compiler

Table 51. Run-Time Considerations

Parameter	Description
DFTACTGRP(*YES)	Identifies whether this program always runs in the OPM default activation group
OPTION(*DEBUGIO)	*DEBUGIO/*NODEBUGIO, determines if breakpoints are generated for input and output specifications
ACTGRP(*STGMDL)	Identifies the activation group in which the program should run
SRTSEQ(*HEX)	Specifies the sort sequence table to be used.

Table 51. Run-Time Considerations (continued)

Parameter	Description
LANGID(*JOBRUN)	Used with SRTSEQ to specify the language identifier for sort sequence
TRUNCNBR(*YES)	Specifies the action to take when numeric overflow occurs for packed-decimal, zoned-decimal, and binary fields in fixed-format operations.
INFOSTMF(path)	Used with PGMINFO, specifies the stream file in the IFS to receive the PCML
PGMINFO(*NONE)	*PCML indicates that PCML (Program Call Markup Language) should be generated for the program; the second parameter indicates whether it should be generated into a stream file or into the module.
LICOPT(options)	Specifies Licensed Internal Code options.

See "Appendix C. The Create Commands" on page 448 for the syntax diagram and parameter descriptions of CRTBNDRPG.

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 28 on page 96.

1. To create the object type:

   ```
   CRTBNDRPG PGM(MYLIB/EMPRPT) DBGVIEW(*SOURCE) DFTACTGRP(*NO)
   ```

   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 run in the default named activation group, QILE. This program object can be debugged using a source view.
2. To debug the program type:

   ```
   STRDBG EMPRPT
   ```

   Figure 32 on page 105 shows the screen which appears after entering the above command.

```
                            Display Module Source
 Program:   EMPRPT         Library:   MYLIB          Module:   EMPRPT
                *============================================================*
                * MODULE NAME:    EMPRPT
                * RELATED FILES:  EMPMST   (PHYSICAL FILE)
                *                 QSYSPRT  (PRINTER FILE)
                * DESCRIPTION:    This program prints employee information
                *                 from the file EMPMST.
                *============================================================*
               FQSYSPRT   O    F   80        PRINTER
               FEMPMST    IP   E           K DISK

               D TYPE            S              8A
               D EMPTYPE         PR             8A
               D   CODE                         1A

               IEMPREC        01
                                                                      More...
 Debug . . .   _________________________________________________________________
_________________________________________________________________________________
 F3=End program   F6=Add/Clear breakpoint   F10=Step   F11=Display variable
 F12=Resume       F17=Watch variable   F18=Work with watch   F24=More keys
```

Figure 32. 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 "Debugging Programs" on page 241.

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 that 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 "Creating a Service Program" on page 129.

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 the payroll program, shown in Figure 33 on page 107.

1. To create the object, type:

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

The CRTBNDRPG command creates the program PAYROLL 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 contains any bound procedure calls, an error is issued and the compilation ends.

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

```
 *------------------------------------------------------------------------*
 * 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
```

```
 *------------------------------------------------------------------------*
 * 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 33. A Sample Payroll Calculation Program

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 466 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)

***SHOWSKP**
Source lines excluded by conditional compilation directives (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 and *SHOWSKP, 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.

The information contained in a compiler listing is also dependent on whether *SRCSTMT or *NOSRCSTMT is specified for the OPTION parameter. For details on how this information changes, see "*NOSRCSTMT Source Heading" and "*SRCSTMT Source Heading".

If any compile option keywords are specified on the control specification, the compiler options in effect will appear in the source section of the listing.

Customizing a Compiler Listing

You can customize a compiler listing in any or all of the following ways:

- Customize the page heading
- Customize the spacing
- Indent structured operations

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 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 *IBM Rational Development Studio for i: ILE RPG Reference*.

Indenting Structured Operations

Note: Calculations can only be indented if they are written with traditional syntax. The RPG compiler does not change the indentation of your free-form calculations (between /FREE and /END-FREE) in the listing. You may indent the free-form claculations directly in your source file.

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 34 on page 110 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********************************************************************                                                002000
    34 C*    MAINLINE                                                      *                                                002100
    35 C********************************************************************                                                002200
    36 C                   WRITE                      FOOT1                                                                 002300
    37 C                   WRITE                      HEAD                                                                  002400
    38 C                   EXFMT                      PROMPT                                                                002500
    39 C*                                                                                                                   002600
    40 C                   DOW                        NOT *IN03                                                             002700
    41 C     CSTKEY        | SETLL                    CMLREC2                            ----20                             002800
    42 C                   | IF                       *IN20                                                                 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*                                                                                                                   004500
    58 C                   SETON                                                         LR----                             004600
```

Figure 34. 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 you encounter any errors when testing your program, 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 check that all your match fields are the correct lengths, and in the correct positions.
- Output-buffer positions

  Lists the start and end positions along with the literal text or field names. Use this to check for errors in your output specifications.
- 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 you have specified the compile-time data in the correct order, and that you have specified the correct values for the SRTSEQ and LANGID parameters to the compiler.

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 35 on page 111 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
======>                                                                 cccccc
*RNF5051 20 a      003100  Resulting-Indicator entry is not valid; defaults to blanks.
*RNF5051 20 b      003100  Resulting-Indicator entry is not valid; defaults to blanks.
*RNF5053 30 c      003100  Resulting-Indicators entry is blank for specified
```

Figure 35. 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 'cccccc' 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 'cccccc' 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 severity 30 and 40 errors, since these are often the cause of other errors.

Non-finger in-line diagnostic messages also indicate errors. However, they are not issued immediately following the line in error. Figure 36 on page 111 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
     1 D FLD1            S             +5    LIKE(FLD2)                                                                       000100
     2 D FLD2            S               D                                                                                    000200
*RNF3479 20      1 000100  A length adjustment is not allowed for a field of the
                           specified data type.
```

Figure 36. Sample Non-Finger In-Line Diagnostic Messages

In this example, FLD1 is defined like FLD2 with a length 5 bytes greater. Later, FLD2 is defined as a date, which makes the length adjustment in the definition of FLD1 invalid. Message RNF3479 is issued pointing at listing line 1. Note that the SEU sequence number (000100) is also given, to aid you in finding the source line in error more quickly. (The SEU sequence number can also be found at listing line 1).

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 checking that portion of the source that a problem exists. However, when possible, the message line includes either the listing Line Number and SEU sequence number, or the 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 sequence number is found just before the message text.

For complete information on browsing a compiler listing, see *ADTS for AS/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.

If OPTION(*NOSRCSTMT) is specified, 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 identify 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.

If OPTION(*SRCSTMT) is specified, the Statement Number on the *right* side of the compiler listing corresponds to the statement number in the run-time error message. If the statement is from the main source member, this is the same as the statement on the *left* side of the compiler listing, and is also the same as the SEU sequence number.

If you have a /COPY member, you can find the source ID number of the actual file in the /COPY Member table at the end of the listing. For an example of a /COPY Member table, see "/COPY Member Table" on page 475.

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 statement numbers for setting breakpoints are those identified in the source listing. The statement numbers are listed in the column labeled as the Line Number when OPTION(*NOSRCSTMT) is specified, and in the column labeled as the Statement Number when OPTION(*SRCSTMT) is specified.
- 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) — these are the defaults.

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 *SHOWSKP).

Note: Except for *SHOWSKP, this is the default setting for each of these parameters on 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 provide additional information as indicated:

- *SHOWCPY and *EXPDDS provide a complete description of the program, including all specifications from ⁄COPY members, and generated specifications from externally described files.
- *SHOWSKP allows you to see the statements that are ignored by the compiler as a result of /IF, / ELSEIF, /ELSE, OR /EOF directives.
- *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 116) 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 that 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 that the compilation failed because of compiler errors.

3
Character '1' means that 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 that are 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 (or -1 if an error occurs while this time is being calculated)

335
Not set. Always blank

336-340
Elapsed compile time to the nearest 10th of a second (or -1 if an error occurs while this time is being calculated)

341-345
Elapsed translator time to the nearest 10th of a second (or -1 if an error occurs while this time is being calculated)

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 that is used by compiler to the nearest 10th of a second

391-395
CPU time that is used by the translator to the nearest 10th of a second

396-400
Not set. Always blank

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

Use Rational Developer for i. This is the recommended method and documentation about creating an ILE RPG program appears in that product's online help.

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.

An ILE RPG module consists of one or more procedures, and the file control blocks and static storage used by all the procedures in the module. The procedures that can make up an ILE RPG module are:

- an optional **cycle-main procedure** which consists of the set of H, F, D, I, C, and O specifications that begin the source. The cycle-main procedure has its own LR semantics and logic cycle; neither of which is affected by those of other ILE RPG modules in the program.
- zero or more **subprocedures**, which are coded on P, D, and C specifications. Subprocedures do not use the RPG cycle. A subprocedure may have local storage that is available for use only by the subprocedure itself. One of the subprocedures may be designated as a linear-main procedure, if a cycle-main procedure is not coded.

The main procedure (if coded) can always be called by other modules in the program. Subprocedures may be local to the module or exported. If they are local, they can only be called by other procedures in the module; if they are exported from the module, they can be called by any procedure in the program.

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. This generally results 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 that 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 *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).

"CRTRPGMOD Parameters and Their Default Values Grouped by Function" on page 116 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 448.

CRTRPGMOD Parameters and Their Default Values Grouped by Function

Table 52. Module Identification

Parameter	Description
MODULE(*CURLIB/*CTLSPEC)	Determines created module name and library
SRCFILE(*LIBL/QRPGLESRC)	If specified, identifies source file and library
SRCMBR(*MODULE)	If specified, identifies file member containing source specifications
SRCSTMF(path)	If specified, indicates the path to the source file in the IFS
INCDIR('path to directory 1:path to directory 2')	Identifies a list of modules to search for /copy and /include files
TEXT(*SRCMBRTXT)	Provides brief description of module

Table 53. Module Creation

Parameter	Description
GENLVL(10)	Conditions module creation to error severity (0-20)
OPTION(*DEBUGIO)	*DEBUGIO/*NODEBUGIO, determines if breakpoints are generated for input and output specifications
OPTION(*GEN)	*GEN/*NOGEN, determines if module is created
OPTION(*NOSRCSTMT)	Specifies how the compiler generates statement numbers for debugging
OPTION(*UNREF)	*UNREF/*NOUNREF Determines whether unreferenced fields are placed in the module
DBGVIEW(*STMT)	Specifies type of debug view, if any, to be included in module
DBGENCKEY(*NONE)	Specifies the encryption for the listing debug view for the module
OPTIMIZE(*NONE)	Determines level of optimization, if any
REPLACE(*YES)	Determines if module should replace existing module
AUT(*LIBCRTAUT)	Specifies type of authority for created module
TGTRLS(*CURRENT)	Specifies the release level the object is to be run on
BNDDIR(*NONE)	Specifies the binding directory to be used for symbol resolution

Table 53. Module Creation (continued)

Parameter	Description
ENBPFRCOL(*PEP)	Specifies whether performance collection is enabled
DEFINE(*NONE)	Specifies condition names that are defined before the compilation begins
PRFDTA(*NOCOL)	Specifies the program profiling data attribute
STGMDL(*INHERIT)	Specifies the storage model for the module

Table 54. Compiler Listing

Parameter	Description
OUTPUT(*PRINT)	Determines if there is a compiler listing
INDENT(*NONE)	Determines if indentation should show in listing, and identify character for marking it
OPTION(*XREF *NOSECLVL *SHOWCPY *EXPDDS *EXT *NOSHOWSKP *NOSRCSTMT)	Specifies the contents of compiler listing

Table 55. Data Conversion Options

Parameter	Description
CVTOPT(*NONE)	Specifies how various data types from externally described files are handled
ALWNULL(*NO)	Determines if the module will accept values from null-capable fields
FIXNBR(*NONE)	Determines which decimal data that is not valid is to be fixed by the compiler

Table 56. Run-Time Considerations

Parameter	Description
SRTSEQ(*HEX)	Specifies the sort sequence table to be used
OPTION(*DEBUGIO)	*DEBUGIO/*NODEBUGIO, determines if breakpoints are generated for input and output specifications
LANGID(*JOBRUN)	Used with SRTSEQ to specify the language identifier for sort sequence
INFOSTMF(path)	Used with PGMINFO, specifies the stream file in the IFS to receive the PCML
PGMINFO(*NONE)	*PCML indicates that PCML (Program Call Markup Language) should be generated for the module; the second parameter indicates whether it should be generated into a stream file or into the module.
TRUNCNBR(*YES)	Specifies action to take when numeric overflow occurs for packed-decimal, zoned-decimal, and binary fields in fixed format operations.
LICOPT(options)	Specifies Licensed Internal Code options.

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 binding section.)

For information on using the compiler listing, see "Using a Compiler Listing" on page 107. A sample compiler listing is provided in "Appendix D. Compiler Listings" on page 466.

Creating a NOMAIN Module

In this example you create an NOMAIN module object TRANSSVC using the CRTRPGMOD command and its default settings. TRANSSVC contains prototyped procedures that perform transaction services for procedures in other modules. The source for TRANSSVC is shown in Figure 37 on page 119. The prototypes for the procedures in TRANSSVC are stored in a /COPY member, as shown in Figure 38 on page 120.

1. To create a module object, type:

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

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

   You bind a module containing NOMAIN to another module using one of the following commands:

   a. CRTPGM command
   b. CRTSRVPGM command
   c. CRTBNDRPG command where the NOMAIN module is included in a binding directory.
2. Once it is bound, this module object can be debugged using a statement view. A compiler listing for the module is also produced.
3. 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:    TRANSSVC (Transaction Services)
      * RELATED FILES:  N/A
      * RELATED SOURCE: TRANSRPT
      * EXPORTED PROCEDURES:  Trans_Inc  -- calculates the income
      *     for the transaction using the data in the fields in the
      *     parameter list.  It returns to the caller after all
      *     the calculations are done.
      *
      *     Prod_Name --  retrieves the product name based on the
      *     input parameter with the product number.
      *=================================================================*
      * This module contains only subprocedures; it is a NOMAIN module.
     H  NOMAIN
      *-----------------------------------------------------------------
      * Pull in the prototypes from the /COPY member
      *-----------------------------------------------------------------
      /COPY TRANSP
```

```
      *-----------------------------------------------------------------
      * Subprocedure Trans_Inc
      *-----------------------------------------------------------------
     P Trans_Inc       B                     EXPORT
     D  Trans_Inc      PI            11P 2
     D    ProdNum                    10P 0    VALUE
     D    Quantity                    5P 0    VALUE
     D    Discount                    2P 2    VALUE
     D  Factor         S              5P 0
      *
     C                   SELECT
     C                   WHEN      ProdNum = 1
     C                   EVAL      Factor = 1500
     C                   WHEN      ProdNum = 2
     C                   EVAL      Factor = 3500
     C                   WHEN      ProdNum = 5
     C                   EVAL      Factor = 20000
     C                   WHEN      ProdNum = 8
     C                   EVAL      Factor = 32000
     C                   WHEN      ProdNum = 12
     C                   EVAL      Factor = 64000
     C                   OTHER
     C                   EVAL      Factor = 0
     C                   ENDSL
     C                   RETURN    Factor * Quantity * (1 - Discount)
     P  Trans_Inc      E
```

```
      *-----------------------------------------------------------------
      * Subprocedure Prod_Name
      *-----------------------------------------------------------------
     P Prod_Name       B                     EXPORT
     D  Prod_Name      PI            40A
     D    ProdNum                    10P 0    VALUE
      *
     C                   SELECT
     C                   WHEN      ProdNum = 1
     C                   RETURN    'Large'
     C                   WHEN      ProdNum = 2
     C                   RETURN    'Super'
     C                   WHEN      ProdNum = 5
     C                   RETURN    'Super Large'
     C                   WHEN      ProdNum = 8
     C                   RETURN    'Super Jumbo'
     C                   WHEN      ProdNum = 12
     C                   RETURN    'Incredibly Large Super Jumbo'
     C                   OTHER
     C                   RETURN    '***Unknown***'
     C                   ENDSL
     P  Prod_Name      E
```

Figure 37. Source for TRANSSVC member

```
     * Prototype for Trans_Inc
    D  Trans_Inc      PR            11P 2
    D    Prod                       10P 0    VALUE
    D    Quantity                    5P 0    VALUE
    D    Discount                    2P 2    VALUE

     * Prototype for Prod_Name
    D  Prod_Name      PR            40A
    D    Prod                       10P 0    VALUE
```

Figure 38. Source for TRANSP /COPY member

Creating a Module for Source Debugging

In this example, you create an ILE RPG module object that you can debug using the source debugger. The module TRANSRPT contains a main procedure which drives the report processing. It calls the procedures in TRANSSVC to perform certain required tasks. The source for this module is shown in Figure 39 on page 121.

To create a module object, type:

```
CRTRPGMOD MODULE(MYLIB/TRANSRPT)  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, TRANSRPT. 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 244.

A compiler listing for the TRANSRPT module will be produced.

```
 *==================================================================*
 * MODULE NAME:    TRANSRPT
 * RELATED FILES:  TRNSDTA  (PF)
 * RELATED SOURCE: TRANSSVC (Transaction services)
 * EXPORTED PROCEDURE:  TRANSRPT
 *       The procedure TRANSRPT reads every tranasction record
 *       stored in the physical file TRNSDTA. It calls the
 *       subprocedure Trans_Inc which performs calculations and
 *       returns a value back.  Then it calls Prod_Name to
 *       to determine the product name.  TRANSRPT then prints
 *       the transaction record out.
 *==================================================================*
FTRNSDTA   IP   E             DISK
FQSYSPRT   O    F   80        PRINTER       OFLIND(*INOF)
 /COPY QRPGLE,TRANSP
 * Define the readable version of the product name like the
 * return value of the procedure 'Prod_Name'
D    ProdName     S             30A
D    Income       S             10P 2
D    Total        S             +5      LIKE(Income)
 *
ITRNSREC       01
 *  Calculate the income using subprocedure Trans_Inc
C                   EVAL      Income = Trans_Inc(PROD : QTY : DISC)
C                   EVAL      Total = Total + Income
 *  Find the name of the product
C                   EVAL      ProdName = Prod_Name(PROD)
OQSYSPRT   H    1P                     1
O         OR    OF
O                                          12 'Product name'
O                                          40 'Quantity'
O                                          54 'Income'
OQSYSPRT   H    1P                     1
O         OR    OF
O                                          30 '----------+
O                                             ----------+
O                                             ----------'
O                                          40 '--------'
O                                          60 '------------'
OQSYSPRT   D    01                     1
O                       ProdName           30
O                       QTY           1    40
O                       Income        1    60
OQSYSPRT   T    LR                     1
O                                             'Total: '
O                       Total         1
```

Figure 39. Source for TRANSRPT module

The DDS for the file TRNSDTA is shown in Figure 40 on page 121. The /COPY member is shown in Figure 38 on page 120.

```
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 TRNSREC
A             PROD          10S 0       TEXT('Product')
A             QTY            5S 0       TEXT('Quantity')
A             DISCOUNT       2S 2       TEXT('Discount')
```

Figure 40. DDS for TRNSDTA

Additional Examples

For additional examples of creating modules, see:

- “Sample Service Program” on page 132, for an example of creating a module for a service program.
- “Binding to a Program” on page 135. for an example of creating a module to be used with a service program.
- “Managing Your Own Heap Using ILE Bindable APIs” on page 153, for an example of creating a module for dynamically allocating storage for a run-time array
- “Sample Source for Debug Examples” on page 286, for example of creating an RPG and C module for use in a sample debug program.

Behavior of Bound ILE RPG Modules

In ILE RPG, the *cycle-main procedure* is the boundary for the scope of LR semantics and the RPG cycle. The *module* is the boundary for the scope of open files.

In any ILE program, there may be several RPG cycles active; there is one RPG cycle for each RPG module that has a cycle-main procedure. The cycles are independent: setting on LR in one cycle-main procedure has no effect on the cycle in another. An RPG module which has a linear-main procedure or has no main procedure does not use the RPG cycle; nor will it effect the cycle in another module.

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 and APIs* section of the *Programming* category in the **IBM i Information Center** at this Web site - http://www.ibm.com/systems/i/infocenter/.

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 IBM i.

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.

If an ILE RPG module contains a main procedure, it implicitly also contains a program entry procedure. Therefore, any ILE RPG module may be specified as the entry module as long as it is not a NOMAIN module.

Figure 41 on page 123 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 TRANSRPT and TRANSSVC. TRANSRPT is the entry module.

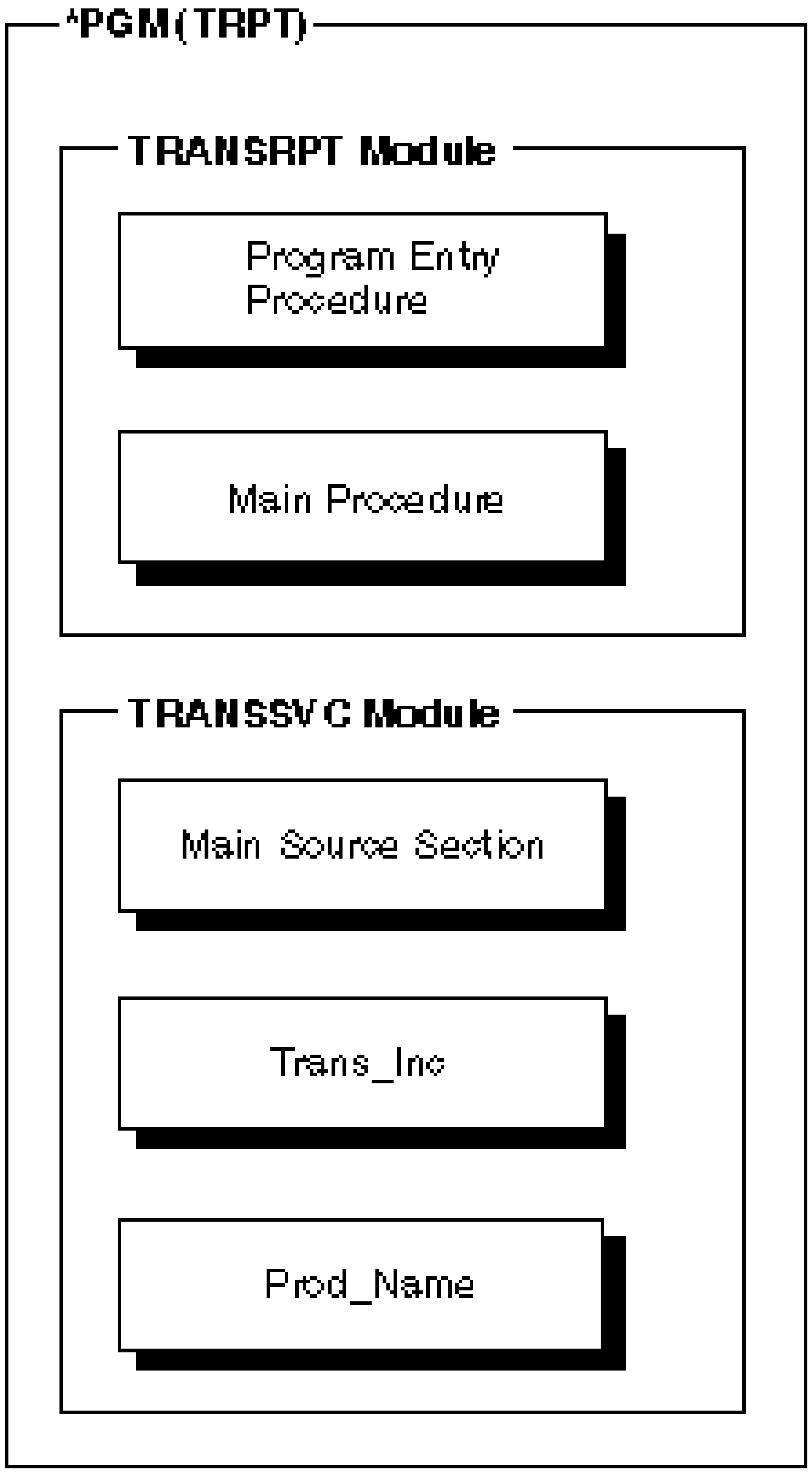


Figure 41. Structure of Program TRPT

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 "Creating a Service Program" on page 129.

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 prior to using the CRTPGM command.

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 entry module.

   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.

   Assuming you have only one module with a main procedure, that is, all modules but one have NOMAIN specified, you can accept the default (*FIRST). Alternatively, you can specify (*ONLY); this will provide a check that in fact only one module has a main procedure. For example, in both of the following situations you could specify ENTMOD(*ONLY).

   - You bind an RPG module to a C module without a main() function.
   - You bind two RPG modules, where one has NOMAIN on the control specification.

   Note: If you are binding more than one ILE RPG module with a main procedure, then you should specify the name of the module that you want to receive control when the program is called. You can also specify *FIRST if the module with a main procedure precedes any other modules with main procedures on the list specified for the MODULE parameter.
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. 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 144.

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 57 on page 125 lists the CRTPGM command parameters and their default values. For a full description of the CRTPGM command and its parameters, refer to the *CL and APIs* section of the *Programming* category in the **IBM i Information Center** at this Web site - http://www.ibm.com/systems/i/infocenter/.

Table 57. 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 or procedure is to be exported (using the EXPORT keyword), it is possible that the variable or procedure name will be identical to a variable or procedure 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 modules into a program TRPT. In this program, the following occurs:

- The module TRANSRPT reads each transaction record from a file TRNSDTA.
- It then calls procedure Trans_Inc and Proc_Name in module TRANSSVC using bound calls within expressions.
- Trans_Inc calculates the income pertaining to each transaction and returns the value to the caller
- Proc_Name determines the product name and returns it
- TRANSRPT then prints the transaction record.

Source for TRANSRPT, TRANSSVC, and TRNSDTA is shown in Figure 39 on page 121, Figure 37 on page 119 and Figure 40 on page 121 respectively.

1. First create the module TRANSRPT. Type:

   ```
   CRTRPGMOD MODULE(MYLIB/TRANSRPT)
   ```

2. Then create module TRANSSVC by typing:

   ```
   CRTRPGMOD MODULE(MYLIB/TRANSSVC)
   ```

3. To create the program object, type:

   ```
   CRTPGM PGM(MYLIB/TRPT) MODULE(TRANSRPT TRANSSVC)
          ENTMOD(*FIRST) ACTGRP(TRPT)
   ```

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

Note that TRANSRPT is listed first in the MODULE parameter. ENTMOD(*FIRST) will find the first module with a program entry procedure. Since only one of the two modules has a program entry procedure, they can be entered in either order.

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

Figure 42 on page 126 shows an output file created when TRPT is run.

```
Product name                    Quantity        Income
------------------------------  --------        ------------
Large                                245          330,750.00
Super                                 15           52,500.00
Super Large                            0                 .00
Super Jumbo                          123        2,952,000.00
Incredibly Large Super Jumbo          15          912,000.00
***Unknown***                         12                 .00
Total:         4,247,250.00
```

Figure 42. File QSYSPRT for TRPT

Additional Examples

For additional examples of creating programs, see:

- "Binding to a Program" on page 135, for an example of binding a module and a service program.
- "Sample Source for Debug Examples" on page 286, 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 and APIs* section of the *Programming* category in the **IBM i Information Center** at this Web site - http://www.ibm.com/systems/i/infocenter/.

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 58. 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 138.

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. For example, if you add a new procedure to a module, you recompile the module object, and then update the program. 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*.

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 298.

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 the 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, 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 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.

Profiling Data
: Represented by the *BLKORD and *PRCORD values. This data is necessary to allow the system to re-apply block order and procedure order profiling data.

Use the CHGPGM command or the CHGMOD command to remove some or all the data from a program or module respectively. 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

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 and APIs* section of the *Programming* category in the IBM i Information Center at this Web site - http://www.ibm.com/systems/i/infocenter/.

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

Use Rational Developer for i. This is the recommended method and documentation about creating a service program appears in that product's online help.

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 run-time 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 bound procedure calls. The initial call has a certain amount of overhead because the binding is not completed until the service program is called. 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 and procedure names are available for use by the calling procedures. To see the exports associated with service program PAYROLL, you would enter:

```
    DSPSRVPGM PAYROLL DETAIL(*PROCEXP *DATAEXP)
```

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.

TIP

If you are planning a module with only subprocedures (that is, with a module with keyword NOMAIN specified on the control specification) you may want to create it as a service program. Only one copy of a service program is needed on a system, and so you will need less storage for the module.

Also, you can copyright your service programs using the COPYRIGHT keyword on the control specification.

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 132 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 59 on page 131 lists the CRTSRVPGM parameters and their defaults. For a full description of the CRTSRVPGM command and its parameters, refer to the *CL and APIs* section of the *Programming* category in the **IBM i Information Center** at this Web site - http://www.ibm.com/systems/i/infocenter/.

Table 59. 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 135 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 127.

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 43 on page 133 shows the source for the service program. Figure 44 on page 134 shows the /COPY member containing the prototype for CvtToHex.

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.

   Note that GetHex is coded as a subroutine rather than a subprocedure, in order to improve run-time performance. An EXSR operation runs much faster than a bound call, and in this example, GetHex is called many times.
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, or when OPDESC is specified on the prototype.

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 170.

```
 *=================================================================*
 * CvtToHex - convert input string to hex output string
 *=================================================================*
H COPYRIGHT('(C) Copyright MyCompany 1995')
D/COPY RPGGUIDE/QRPGLE,CVTHEXPR
 *-----------------------------------------------------------------*
 * Main entry parameters
 * 1. Input:   string                  character(n)
 * 2. Output:  hex string              character(2 * n)
 *-----------------------------------------------------------------*
D CvtToHex        PI                   OPDESC
D   InString                 16383     CONST OPTIONS(*VARSIZE)
D   HexString                32766     OPTIONS(*VARSIZE)

 *-----------------------------------------------------------------*
 * Prototype for CEEDOD (Retrieve operational descriptor)
 *-----------------------------------------------------------------*
D CEEDOD          PR
D ParmNum                       10I 0 CONST
D                               10I 0
D                               10I 0
D                               10I 0
D                               10I 0
D                               10I 0
D                               12A   OPTIONS(*OMIT)

 * Parameters passed to CEEDOD
D DescType        S             10I 0
D DataType        S             10I 0
D DescInfo1       S             10I 0
D DescInfo2       S             10I 0
D InLen           S             10I 0
D HexLen          S             10I 0

 *-----------------------------------------------------------------*
 * Other fields used by the program                                *
 *-----------------------------------------------------------------*
D HexDigits       C                   CONST('0123456789ABCDEF')
D IntDs           DS
D   IntNum                       5I 0 INZ(0)
D   IntChar                      1    OVERLAY(IntNum:2)
D HexDs           DS
D   HexC1                        1
D   HexC2                        1
D InChar          S              1
D Pos             S              5P 0
D HexPos          S              5P 0
```

Figure 43. Source for Service Program CvtToHex

```
      *-----------------------------------------------------------------*
      * Use the operational descriptors to determine the lengths of     *
      * the parameters that were passed.                                *
      *-----------------------------------------------------------------*
     C                   CALLP     CEEDOD(1         : DescType : DataType :
     C                                    DescInfo1 : DescInfo2: Inlen    :
     C                                    *OMIT)
     C                   CALLP     CEEDOD(2         : DescType : DataType :
     C                                    DescInfo1 : DescInfo2: HexLen   :
     C                                    *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

      *=================================================================*
      * 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      IntChar = InChar
     C     IntNum        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
```

```
      *=================================================================*
      * CvtToHex - convert input string to hex output string
      *
      * Parameters
      * 1. Input:   string                   character(n)
      * 2. Output:  hex string               character(2 * n)
      *=================================================================*
     D CvtToHex        PR                    OPDESC
     D   InString                 16383      CONST OPTIONS(*VARSIZE)
     D   HexString                32766      OPTIONS(*VARSIZE)
```

Figure 44. Source for /COPY Member with Prototype for 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 45 on page 135 shows the binder language needed to define the exports of the service program CVTTOHEX. This source is used in the EXPORT, SRCFILE and SRCMBR parameters of the CRTSRVPGM command.

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

Figure 45. 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 43 on page 133, 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 45 on page 135.

   ```
   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 46 on page 137, 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
      * Prototype for CvtToHex
     D/COPY RPGGUIDE/QRPGLE,CVTHEXPR
     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 prototyped call.  Operational descriptors are          *
      * passed, as required by the called procedure CvtToHex.          *
      *----------------------------------------------------------------*
     C                   EVAL      Comment = '10 character output'
     C                   CLEAR                   ResultDS
     C                   CALLP     CvtToHex(Instring : Result10)
     C                   EXCEPT

      *----------------------------------------------------------------*
      * Pass character string and the 14-character result field        *
      * using a CALLB(D). The operation extender (D) will create       *
      * operational descriptors for the passed parameters.  CALLB      *
      * is used here for comparison with the above CALLP.              *
      *----------------------------------------------------------------*
     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 46. 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, there are two ways to add a new procedure to CVTTOHEX, depending on whether the new procedure goes into the existing module or into a new one.

To add a new procedure to an existing module, you would:

1. Add the new procedure to the existing module.
2. Recompile the changed module.
3. 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.
4. Recreate the service program using CRTSRVPGM.

To add a new procedure using a new module, 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, as mentioned above.
3. Bind the new module to service program CVTTOHEX by re-creating the service program.

With either method, 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 47 on page 138 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 127 and also *ILE Concepts*.

```
                                                   Create Program                                                       Page    1
5769WDS V5R2M0  020719                                                             MYLIB/CVTHEXPGM    ISERIES1  08/15/02
  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
5769WDS V5R2M0  020719                                                             MYLIB/CVTHEXPGM    ISERIES1  08/15/02
  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
5769WDS V5R2M0  020719                                                             MYLIB/CVTHEXPGM    ISERIES1  08/15/02
  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 47. Basic Binder listing for CVTHEXPGM

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. The *CL Programming, SC41-5721-06* manual contains information on this menu.
- The Start Programming Development Manager (STRPDM) command. The *ADTS/400: Programming Development Manager* manual contains information on this command.
- The QCMDEXC program. The *CL Programming* manual contains information on this program.
- A high-level language. "Calling Programs and Procedures" on page 159 provides information on running programs from another HLL or calling service programs and procedures.,

Note: Use Rational Developer for i. This is the recommended method and documentation about running a program appears in that product's online help.

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
```

For further information about using the CL CALL command, see the *CL and APIs* section of the *Programming* category in the **IBM i Information Center** at this Web site - http://www.ibm.com/systems/i/infocenter/.

Once you call your program, the IBM i 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 procedure is the program entry procedure, then that program must have one and only one *ENTRY PLIST specified. The parameters that 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 section on "Passing Parameters between Programs" in the *CL Programming* manual 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 48 on page 140 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
```

```
      *-----------------------------------------------------------------*
      * 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 48. ILE RPG Program that Requires Parameters at Run Time

Figure 49 on page 141 shows the DDS that is referenced by the EMPRPT2 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 49. DDS for EMPRPT2

Running a Program From a Menu-Driven Application

Another way to run an ILE program is from a menu-driven application. The workstation user selects an option from a menu, which in turn calls a particular program. Figure 50 on page 141 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 50. Example of an Application Menu

The menu shown in Figure 50 on page 141 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 51 on page 142 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 51. Data Description Specification of an Application Menu

Figure 52 on page 142 shows the source of the application menu illustrated in Figure 50 on page 141. 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 52. 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 52 on page 142 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* manual for further information about using command definitions.

Replying to Run-Time Inquiry Messages

When you run a program with ILE RPG procedures, run-time inquiry messages may be generated. They occur when the default error handler is invoked for a function check in a cycle-main procedure. See "Exception Handling within a Cycle-Main Procedure" on page 293. The inquiry messages require a response before the program continues running.

Note: Inquiry messages are never issued for subprocedures (including those designated as linear-main procedures), since the default error handling for a function check in a subprocedure causes the subprocedure to be cancelled, causing the exception to percolate to the caller of the subprocedure. See Exception Handling within Subprocedures.

If the caller of the subprocedure is an RPG procedure, the call will fail with status 00202, independent of the status code associated with the actual exception. If the failed call causes an RPG cycle-main procedure to invoke its default handler, inquiry message RNQ0202 will be issued.

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.

To find all the ILE RPG inquiry messages you can add to the system reply list, use the following command:

```
DSPMSGD RANGE(RNQ0100 RNQ9999)
        MSGF(QRNXMSG) DETAIL(*BASIC) OUTPUT(*PRINT)
```

Note: ILE RPG 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. For details of the ADDRPYLE and WRKRPYLE commands, see the *CL and APIs* section of the *Programming* category in the **IBM i Information Center** at this Web site - http://www.ibm.com/systems/i/infocenter/.

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 291.

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 that 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 that 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 that 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 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 IBM i 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 Program Compatibility

If you have an OPM application that 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 428 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 71.

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.

An ILE RPG program created DFTACTGRP(*YES) will have its storage released when it ends with LR on or abnormally.

Note: The storage associated with ILE programs running in the default activation group via *CALLER is not released until you sign off (for an interactive job) or until the job ends (for a batch job).

If many ILE RPG 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.

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 the RCLACTGRP command, refer to the see the *CL and APIs* section of the *Programming* category in the **IBM i Information Center** at this Web site - http://www.ibm.com/systems/i/infocenter/. 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 that 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 or service programs that were activated into the OPM default activation group because they were created with *CALLER, files will be closed when the RCLRSC command is issued. For programs, the storage will be re-initialized; however, the storage will not be released. For service programs, the storage will neither be re-initialized nor released.

Note: This means that if you have a service program that ran in the default activation group and left files open, and a RCLRSC is issued, when you call the service program again, the files will still appear to be open, so so any I/O operations will result in an error.

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 the *CL and APIs* section of the *Programming* category in the **IBM i Information Center** at this Web site - http://www.ibm.com/systems/i/infocenter/. 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.

To manage heaps, you can use:

- The ALLOC, REALLOC, and DEALLOC operation codes
- The %ALLOC and %REALLOC built-in functions
- The ILE bindable APIs

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

```
 * Two counters are kept:
 * 1. The current number of array elements
 * 2. The number of array elements that are allocated for the array

D arrInfo         DS                  QUALIFIED
D   pArr                          *   INZ(*NULL)
D   numElems                    10I 0 INZ(0)
D   numAlloc                    10I 0 INZ(0)
D arr             S             20A   VARYING DIM(32767)
D                                     BASED(arrInfo.pArr)
D i               S             10I 0
 /free
    // Allocate storage for a few array elements
    // (The number of elements that the array is considered to
    // actually have remains zero.)

    arrInfo.numAlloc = 2;
    arrInfo.pArr = %alloc(arrInfo.numAlloc * %size(arr));

    // Add two elements to the array

    if arrInfo.numAlloc < arrInfo.numElems + 2;
      // There is no room for the new elements.
      // Allocate a few more elements.

      arrInfo.numAlloc += 10;
      arrInfo.pArr = %realloc (arrInfo.pArr
                             : arrInfo.numAlloc * %size(arr));
    endif;
    arrInfo.numElems += 1;
    arr(arrInfo.numElems) = 'XYZ Electronics';
    arrInfo.numElems += 1;
    arr(arrInfo.numElems) = 'ABC Tools';

    // Search the array

    i = %lookup ('XYZ Electronics' : arr : 1 : arrInfo.numElems);
    // i = 1

    // Sort the array

    sorta %subarr(arr : 1 : arrInfo.numElems);
```

Figure 53. Allocating, sorting and searching dynamically-allocated arrays

```
    // Search the array again

    i = %lookup ('XYZ Electronics' : arr : 1 : arrInfo.numElems);
    // Now, i = 2, since the array is now sorted

    // Remove the last element from the array

    arrInfo.numElems -= 1;

    // Clear the array
    // This can be done simply by setting the current number of
    // elements to zero.  It is not necessary to actually clear
    // the data in the previously used elements.

    arrInfo.numElems = 0;

    // Free the storage for the array

    dealloc arrInfo.pArr;
    reset arrInfo;

    return;
```

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

Managing the Default Heap Using RPG Operations

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 use the following RPG operations on the default heap:

- The ALLOC operation code and the %ALLOC built-in function allocate storage within the default heap.
- The DEALLOC operation code frees one previous allocation of heap storage from any heap.
- The REALLOC operation code and the %REALLOC built-in function change the size of previously allocated storage from any heap.

Note: Although ALLOC and %ALLOC work only with the default heap, DEALLOC, REALLOC, and %REALLOC work with both the default heap and user-created heaps.

Figure 54 on page 149 shows an example of how the memory management operation codes can be used to build a linked list of names.

```
      *-----------------------------------------------------------------*
      * Prototypes for subprocedures in this module                    *
      *-----------------------------------------------------------------*
     D AddName         PR
     D   name_parm                   40A
     D Display         PR
     D Free            PR
      *-----------------------------------------------------------------*
      * Each element in the list contains a pointer to the             *
      * name and a pointer to the next element                         *
      *-----------------------------------------------------------------*
     D elem            DS                  BASED(elem@)
     D   name@                        *
     D   next@                        *
     D   name_len                    5U 0
     D nameVal         S             40A    BASED(name@)
     D elemSize        C                   %SIZE(elem)
      *-----------------------------------------------------------------*
      * The first element in the list is in static storage.           *
      * The name field of this element is not set to a value.         *
      *-----------------------------------------------------------------*
     D first           DS
     D                                *   INZ(*NULL)
     D                                *   INZ(*NULL)
     D                               5U 0 INZ(0)
      *-----------------------------------------------------------------*
      * This is the pointer to the current element.                    *
      * When elem@ is set to the address of <first>, the list is       *
      * empty.                                                         *
      *-----------------------------------------------------------------*
     D elem@           S              *   INZ(%ADDR(first))
      *-----------------------------------------------------------------*
      * Put 5 elements in the list                                     *
      *-----------------------------------------------------------------*
     C                   DO        5
     C     'Name?'       DSPLY                   name             40
     C                   CALLP     AddName(name)
     C                   ENDDO

      *-----------------------------------------------------------------*
      * Display the list and then free it.                             *
      *-----------------------------------------------------------------*
     C                   CALLP     Display
     C                   CALLP     Free
     C                   EVAL      *INLR = '1'
```

Figure 54. Memory Management - Build a Linked List of Names

```
 *-----------------------------------------------------------------*
 * S U B P R O C E D U R E S                                       *
 *-----------------------------------------------------------------*
 *-----------------------------------------------------------------*
 * AddName - add a name to the end of the list                     *
 *-----------------------------------------------------------------*
P AddName         B
D AddName         pi
D   name                        40A
 *-----------------------------------------------------------------*
 * Allocate a new element for the array, pointed at by the         *
 * 'next' pointer of the current end of the list.                  *
 *                                                                 *
 * Before:                                                         *
 *                                                                 *
 *  .-------------.                                                *
 *  |             |                                                *
 *  | name     *--->abc                                            *
 *  | name_len 3  |                                                *
 *  | next     *-------|||                                         *
 *  |             |                                                *
 *  '-------------'                                                *
 *                                                                 *
 *-----------------------------------------------------------------*
C                   ALLOC     elemSize      next@
 *-----------------------------------------------------------------*
 *                                                                 *
 * After: Note that the old element is still the current one       *
 *        because elem@ is still pointing to the old element       *
 *                                                                 *
 *  .-------------.               .--------------.                 *
 *  |             |       .------>|              |                 *
 *  | name     *--->abc   |       |              |                 *
 *  | name_len 3  |       |       |              |                 *
 *  | next     *----------'       |              |                 *
 *  |             |               |              |                 *
 *  '-------------'               '--------------'                 *
 *                                                                 *
 * Now set elem@ to point to the new element                       *
 *-----------------------------------------------------------------*
C                   EVAL      elem@ = next@
```

```
      *-----------------------------------------------------------------*
      *                                                                 *
      * After: Now the names name@, name_len and next@ refer            *
      *        to storage in the new element                            *
      *                                                                 *
      *  .-------------.              .--------------.                  *
      *  |             |       .------>|              |                  *
      *  |          *--->abc   |       | name      *  |                  *
      *  |          3  |       |       | name_len     |                  *
      *  |          *----------'       | next      *  |                  *
      *  |             |               |              |                  *
      *  '-------------'               '--------------'                  *
      *                                                                 *
      * Now set the values of the new element.                          *
      * The next pointer is set to *NULL to indicate that it is the     *
      * end of the list.                                                *
      *-----------------------------------------------------------------*
     C                   EVAL      next@ = *NULL
      *-----------------------------------------------------------------*
      * Save the length of the name (not counting trailing blanks)
      *-----------------------------------------------------------------*
     C                   EVAL      name_len = %len(%trimr(name))
      *-----------------------------------------------------------------*
      * Storage is allocated for the name and then set to the value of
      * the name.
      *-----------------------------------------------------------------*
     C                   ALLOC     name_len      name@
     C                   EVAL      %SUBST(nameVal:1:name_len) = name
      *-----------------------------------------------------------------*
      *                                                                 *
      * After:                                                          *
      *                                                                 *
      *  .-------------.              .--------------.                  *
      *  |             |       .------>|              |                  *
      *  |          *--->abc   |       | name      *--->newname         *
      *  |          3  |       |       | name_len  nn |                  *
      *  |          *----------'       | next      *--->|||              *
      *  |             |               |              |                  *
      *  '-------------'               '--------------'                  *
      *-----------------------------------------------------------------*
     P AddName         E
```

```
      *-----------------------------------------------------------------*
      * Display - display the list                                      *
      *-----------------------------------------------------------------*
     P Display         B
     D saveElem@       S               *
     D dspName         S             40A
      *-----------------------------------------------------------------*
      * Save the current elem pointer so the list can be restored after *
      * being displayed and set the list pointer to the beginning of    *
      * the list.                                                       *
      *-----------------------------------------------------------------*
     C                   EVAL      saveElem@ = elem@
     C                   EVAL      elem@ = %ADDR(first)
      *-----------------------------------------------------------------*
      * Loop through the elements of the list until the next pointer is *
      * *NULL                                                           *
      *-----------------------------------------------------------------*
     C                   DOW       next@ <> *NULL
     C                   EVAL      elem@ = next@
     C                   EVAL      dspName = %SUBST(nameVal:1:name_len)
     C     'Name: '      dsply                   dspName
     C                   ENDDO
      *-----------------------------------------------------------------*
      * Restore the list pointer to its former place
      *-----------------------------------------------------------------*
     C                   EVAL      elem@ = saveElem@
     P Display         E
```

```
      *-----------------------------------------------------------------*
      * Free - release the storage used by the list                    *
      *-----------------------------------------------------------------*
     P Free            B
     D prv@            S               *
      *-----------------------------------------------------------------*
      * Loop through the elements of the list until the next pointer is *
      * *NULL, starting from the first real element in the list        *
      *-----------------------------------------------------------------*
     C                   EVAL      elem@ = %ADDR(first)
     C                   EVAL      elem@ = next@
     C                   DOW       elem@ <> *NULL
      *-----------------------------------------------------------------*
      * Free the storage for name                                      *
      *-----------------------------------------------------------------*
     C                   DEALLOC                 name@
      *-----------------------------------------------------------------*
      * Save the pointer to current elem@
      *-----------------------------------------------------------------*
     C                   EVAL      prv@ = elem@
      *-----------------------------------------------------------------*
      * Advance elem@ to the next element
      *-----------------------------------------------------------------*
     C                   EVAL      elem@ = next@

      *-----------------------------------------------------------------*
      * Free the storage for the current element
      *-----------------------------------------------------------------*
     C                   DEALLOC                 prv@
     C                   ENDDO

      *-----------------------------------------------------------------*
      * Ready for a new list:
      *-----------------------------------------------------------------*
     C                   EVAL      elem@ = %ADDR(first)
     P Free            E
```

Heap Storage Problems

Figure 55 on page 152 shows possible problems associated with the misuse of heap storage.

```
 *..1....+....2....+....3....+....4....+....5....+....6....+....7...+....
 *-----------------------------------------------------------------*
 * Heap Storage Misuse                                             *
 *-----------------------------------------------------------------*
D Fld1            S             25A    BASED(Ptr1)
D Ptr1            S               *

 /FREE
    Ptr1 = %ALLOC(25);
    DEALLOC Ptr1;

  // After this point, Fld1 should not be accessed since the
  // basing pointer Ptr1 no longer points to allocated storage.

    SomePgm();

  // During the previous call to 'SomePgm', several storage allocations
  // may have been done.  In any case, it is extremely dangerous to
  // make the following assignment, since 25 bytes of storage will
  // be filled with 'a'.  It is impossible to know what that storage
  // is currently being used for.

    Fld1 = *ALL'a';
 /END-FREE
```

Figure 55. Heap Storage Misuse

Similarly, errors can occur in the following cases:

- A similar error can be made if a pointer is copied before being reallocated or deallocated. Great care must be taken when copying pointers to allocated storage, to ensure that they are not used after the storage is deallocated or reallocated.
- If a pointer to heap storage is copied, the copy can be used to deallocate or reallocate the storage. In this case, the original pointer should not be used until it is set to a new value.
- If a pointer to heap storage is passed as a parameter, the callee could deallocate or reallocate the storage. After the call returns, attempts to access the pointer could cause problems.
- If a pointer to heap storage is set in the *INZSR, a later RESET of the pointer could cause the pointer to get set to storage that is no longer allocated.
- Another type of problem can be caused if a pointer to heap storage is lost (by being cleared, or set to a new pointer by an ALLOC operation, for example). Once the pointer is lost, the storage it pointed to cannot be freed. This storage is unavailable to be allocated since the system does not know that the storage is no longer addressable.

  The storage will not be freed until the activation group ends.

Managing Your Own Heap Using ILE Bindable APIs

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

The following example shows you how to manage dynamic storage for a run-time array with a user-created heap from an ILE RPG procedure. In this example, the procedures in the module DYNARRAY dynamically allocate storage for a practically unbounded packed array. The procedures in the module perform the following actions on the array:

- Initialize the array
- Add an element to the array
- Return the value of an element
- Release the storage for the array.

DYNARRAY performs these actions using the three ILE bindable storage APIs, CEECRHP (Create Heap), CEEGTST (Get Storage), and CEEDSHP (Discard Heap), as well as the REALLOC operation code. For specific information about the storage management bindable APIs, refer to the *CL and APIs* section of the *Programming* category in the **IBM i Information Center** at this Web site - http://www.ibm.com/systems/i/infocenter/.

Figure 56 on page 154 shows the /COPY file DYNARRI containing the prototypes for the procedures in DYNARRAY. This /COPY file is used by the DYNARRAY module as well as any other modules that call the procedures in DYNARRAY.

DYNARRAY 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 DYNA_TYPE to a character field.

```
      *=================================================================
      * DYNARRAY :     Handle a (practically) unbounded run-time
      *                Packed(15,0) array.   The DYNARRAY module contains
      *                procedures to allocate the array, return or set
      *                an array value and deallocate the array.
      *=================================================================
     D DYNA_TYPE       S             15P 0
     D DYNA_INIT       PR
     D DYNA_TERM       PR
     D DYNA_SET        PR
     D   Element                             VALUE LIKE(DYNA_TYPE)
     D   Index                        5I 0   VALUE
     D DYNA_GET        PR                    LIKE(DYNA_TYPE)
     D   Index                        5I 0   VALUE
```

Figure 56. /COPY file DYNARRI containing prototypes for DYNARRAY module

Figure 57 on page 154 shows the beginning of module DYNARRAY containing the Control specification, and Definition specifications.

```
      *=================================================================
      * DYNARRAY :     Handle a (practically) unbounded run-time
      *                Packed(15,0) array.   This module contains
      *                procedures to allocate the array, return or set
      *                an array value and deallocate the array.
      *=================================================================
     H NOMAIN
      *-----------------------------------------------------------------
      * Prototypes for the procedures in this module.
      *-----------------------------------------------------------------
      /COPY DYNARRI
      *-----------------------------------------------------------------
      * Interface to the CEEGTST API (Get Heap Storage).
      *  1) HeapId = Id of the heap.
      *  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.
      *-----------------------------------------------------------------
     D CEEGTST         PR
     D   HeapId                      10I 0       CONST
     D   Size                        10I 0       CONST
     D   RetAddr                       *
     D   Feedback                    12A         OPTIONS(*OMIT)
      *-----------------------------------------------------------------
      * Interface to the CEECRHP API (Create Heap).
      *  1) HeapId     = Id of the heap.
      *  2) InitSize   = Initial size of the heap.
      *  3) Incr       = Number of bytes to increment if heap must be
      *                  enlarged.
      *  4) AllocStrat = Allocation strategy for this heap.  We will
      *                  specify a value of 0 which allows the system
      *                  to choose the optimal strategy.
      *  5) *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.
      *-----------------------------------------------------------------
     D CEECRHP         PR
     D   HeapId                      10I 0
     D   InitSize                    10I 0       CONST
     D   Incr                        10I 0       CONST
     D   AllocStrat                  10I 0       CONST
     D   Feedback                    12A         OPTIONS(*OMIT)
```

Figure 57. Global variables and local prototypes for DYNARRAY

```
 *-----------------------------------------------------------------
 * Interface to the CEEDSHP API (Discard Heap).
 *  1) HeapId     = Id of the heap.
 *  2) *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.
 *-----------------------------------------------------------------
D CEEDSHP         PR
D   HeapId                      10I 0
D   Feedback                    12A         OPTIONS(*OMIT)
 *-----------------------------------------------------------------
 * Global variables.
 *-----------------------------------------------------------------
D HeapVars        DS
D   HeapId                      10I 0
D   DynArr@                       *
 *-----------------------------------------------------------------
 * 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(DYNA_TYPE)
 *-----------------------------------------------------------------
 * Global to keep track of the current number of elements
 * in the dynamic array.
 *-----------------------------------------------------------------
D NumElems        S             10I 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 58 on page 156 shows the subprocedures in DYNARRAY.

```
      *=================================================================
      * DYNA_INIT: Initialize the array.
      *
      * Function: Create the heap and allocate an initial amount of
      *           storage for the run time array.
      *=================================================================
     P DYNA_INIT       B                   EXPORT
      *-----------------------------------------------------------------
      * Local variables.
      *-----------------------------------------------------------------
     D Size            S             10I 0
      *
      * Start with a pre-determined number of elements.
      *
     C                   Z-ADD     INITALLOC     NumElems
      *
      * Determine the number of bytes needed for the array.
      *
     C                   EVAL      Size = NumElems * %SIZE(DynArr)
      *
      * Create the heap
      *
     C                   CALLP     CEECRHP(HeapId : Size : 0 : 0 : *OMIT)

      *
      * Allocate the storage and set the array basing pointer
      * to the pointer returned from the API.
      *
      * Note that the ALLOC operation code uses the default heap so
      * we must use the CEEGTST API to specify a different heap.
      *
     C                   CALLP     CEEGTST(HeapId : Size : DynArr@ : *OMIT)

      *
      * Initialize the storage for the array.
      *
     C     1             DO        NumElems      I                 5 0
     C                   CLEAR                   DynArr(I)
     C                   ENDDO
     P DYNA_INIT       E
```

Figure 58. DYNARRAY Subprocedures

```
      *=================================================================
      * DYNA_TERM: Terminate array handling.
      *
      * Function: Delete the heap.
      *=================================================================
     P DYNA_TERM       B                   EXPORT
     C                   CALLP     CEEDSHP(HeapId : *OMIT)
     C                   RESET                   HeapVars
     P DYNA_TERM       E
```

```
      *=================================================================
      * DYNA_SET: Set an array element.
      *
      * Function: Ensure the array is big enough for this element,
      *           and set the element to the provided value.
      *=================================================================
     P DYNA_SET        B                   EXPORT
      *-----------------------------------------------------------------
      * Input parameters for this procedure.
      *-----------------------------------------------------------------
     D DYNA_SET        PI
     D   Element                             VALUE LIKE(DYNA_TYPE)
     D   Index                        5I 0   VALUE
      *-----------------------------------------------------------------
      * Local variables.
      *-----------------------------------------------------------------
     D Size            S             10I 0
      *-----------------------------------------------------------------
      * 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     Index         IFGT      NumElems
     C                   EXSR      REALLOC
     C                   ENDIF
     C                   EVAL      DynArr(Index) = Element
      *=================================================================
      * REALLOC: Reallocate storage subroutine
      *
      *         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.  The new storage has the same value
      * as the old storage.
      *
     C                   REALLOC   Size          DynArr@
      *
      * Initialize the new elements for the array.
      *
     C     1             ADD       OldElems      I
     C     I             DO        NumElems      I                 5 0
     C                   CLEAR                   DynArr(I)
     C                   ENDDO
     C                   ENDSR
     P DYNA_SET        E
```

```
      *=================================================================
      * DYNA_GET: Return an array element.
      *
      * Function: Return the current value of the array element if
      *           the element is within the size of the array, or
      *           the default value otherwise.
      *=================================================================
     P DYNA_GET        B                   EXPORT
      *-----------------------------------------------------------------
      * Input parameters for this procedure.
      *-----------------------------------------------------------------
     D DYNA_GET        PI                    LIKE(DYNA_TYPE)
     D   Index                         5I 0   VALUE
      *-----------------------------------------------------------------
      * Local variables.
      *-----------------------------------------------------------------
     D Element         S                     LIKE(DYNA_TYPE) INZ
      *-----------------------------------------------------------------
      * If the element requested is within the current size of the
      * array then return the element's current value.  Otherwise
      * the default (initialization) value can be used.
      *-----------------------------------------------------------------
     C     Index         IFLE      NumElems
     C                   EVAL      Element = DynArr(Index)
     C                   ENDIF
     C                   RETURN    Element
     P DYNA_GET        E
```

The logic of the subprocedures is as follows:

1. DYNA_INIT creates the heap using the ILE bindable API CEECRHP (Create Heap), storing the heap Id in a global variable HeapId. It 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. DYNA_TERM destroys the heap using the ILE bindable API CEEDSHP (Discard Heap).
3. DYNA_SET sets the value of an element in the array.

   Before adding an element to the array, the procedure checks to see if there is sufficient heap storage. If not, it uses operation code REALLOC to acquire additional storage.
4. DYNA_GET returns the value of a specified element. 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.

To create the module DYNARRAY, type:

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

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

Figure 59 on page 159 shows another module that tests the procedures in DYNARRAY.

```
      *=================================================================
      * DYNTEST: Test program for DYNARRAY module.
      *=================================================================
      /COPY EXAMPLES,DYNARRI
     D X               S                    LIKE(DYNA_TYPE)
      * Initialize the array
     C                   CALLP     DYNA_INIT
      * Set a few elements
     C                   CALLP     DYNA_SET (25 : 3)
     C                   CALLP     DYNA_SET (467252232 : 1)
     C                   CALLP     DYNA_SET (-2311 : 750)
      * Retrieve a few elements
     C                   EVAL      X = DYNA_GET (750)
     C     '750'         DSPLY                   X
     C                   EVAL      X = DYNA_GET (8001)
     C     '8001'        DSPLY                   X
     C                   EVAL      X = DYNA_GET (2)
     C     '2'           DSPLY                   X

      * Clean up
     C                   CALLP     DYNA_TERM
     C                   SETON                                        LR
```

Figure 59. Sample module using procedures in DYNARRAY

Calling Programs and Procedures

In ILE, it is possible to call either a program or procedure. Furthermore, ILE RPG provides the ability to call prototyped or non-prototyped programs and procedures. (A prototype is an external definition of the call interface that allows the compiler to check the interface at compile time.)

The recommended way to call a program or procedure is to use a prototyped call. The syntax for calling and passing parameters to prototyped procedures or programs uses the same free-form syntax that is used with built-in functions or within expressions. For this reason, a prototyped call is sometimes referred to as a 'free-form' call.

In cases where there will be no RPG callers of a program or procedure, or where the procedure is not exported from the module, it is optional to specify the prototype. If the prototype is not specified, the RPG compiler will generate the prototype from the procedure interface. If the procedure does not contain a procedure interface, the RPG compiler will generate a prototype with no return value and no parameters. It is still considered a prototyped call to call such a procedure that does not have an explicit prototype.

Use the CALL or CALLB operations to call a program or procedure when:

- You have an extremely simple call interface
- You require the power of the PARM operation with factor 1 and factor 2.
- You want more flexibility than is allowed by prototyped parameter checking.

This chapter describes how to:

- Call a program or procedure
- Use a prototyped call
- Pass prototyped parameters
- Use a fixed-form call
- 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 module contains an optional main procedure and zero or more subprocedures. In this chapter, the term 'procedure' applies to both main procedures and subprocedures.

An ILE '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 that is designated at program creation time to receive control when a program is called. If the entry module of the program is an ILE RPG module, then the main procedure of that module is called by the program entry procedure immediately after the program is called.

This section contains general information on:

- Program call compared to procedure call
- Call stack (or how a series of calls interact)
- Recursion
- Parameter passing considerations

Calling Programs

You can call OPM or ILE programs by using program calls. A **program call** is a call that is 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 CALLP operation or both the CALL and PARM operations to make a program call. If you use the CALL and PARM operations, then the compiler cannot perform type checking on the parameters, which may result in run-time errors.

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 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 that are 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 that is specified in the passed parameter. You can also manipulate arrays of procedure names or addresses to dynamically route a procedure call to different procedures. If the called procedure is in the same activation group, the 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.
- A procedure in a separate ILE service program.

Any procedure that can be called by using a static procedure call can also be called through a procedure pointer.

For a list of examples using static procedure calls, see "Examples of Free-Form Call" on page 167 and "Examples of CALL and CALLB" on page 181. For examples of using procedure pointers, see the section on the procedure pointer data type in *IBM Rational Development Studio for i: ILE RPG Reference*.

You use the CALLP or both the CALLB and PARM operations to make a procedure call. You can also call a prototyped procedure with an expression if the procedure returns a value. If you use the CALLB and PARM operations, then the compiler cannot perform type checking on the parameters, which may result in run-time errors.

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 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 (the main procedure) is added. When a procedure is called, only the user's procedure (a main procedure or subprocedure) is added; there is no overhead of a program entry procedure.

Figure 60 on page 161 shows a call stack for an application consisting of an OPM program which calls an ILE program. The RPG main procedure of the ILE program calls an RPG subprocedure, which in turn calls a C procedure. Note that in the diagrams in this book, the most recent entry is at the bottom of the stack.

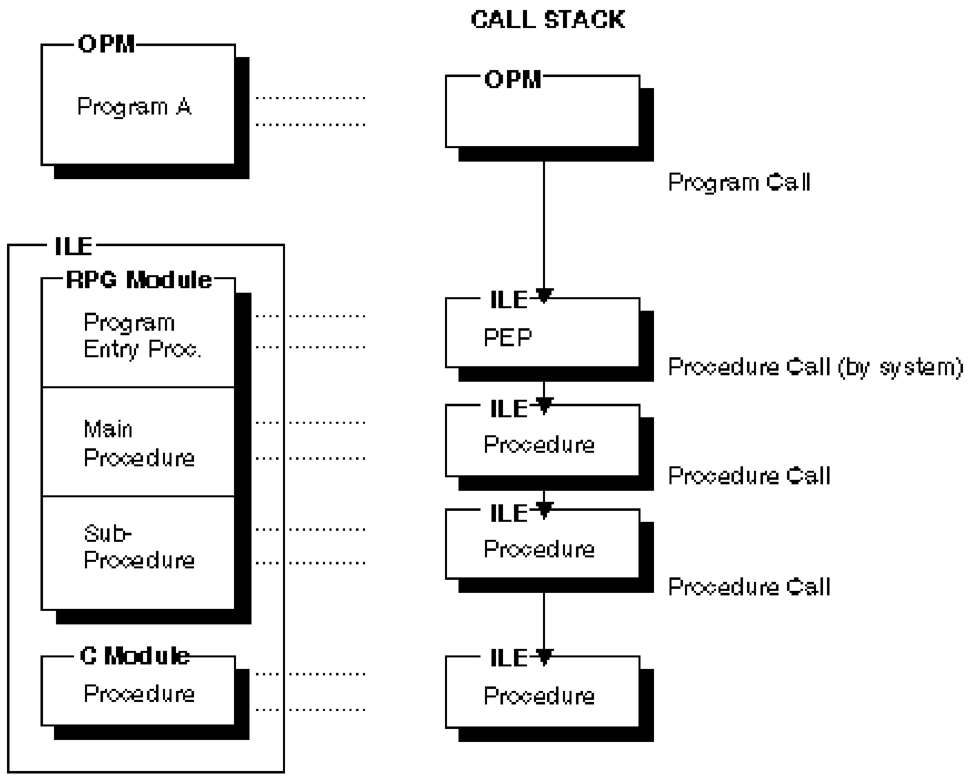


Figure 60. Program and Procedure Calls on the Call Stack

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.

Recursive Calls

Recursive calls are allowed for subprocedures. A recursive call is one where procedure A calls itself or calls procedure B which then calls procedure A again. Each recursive call causes a new invocation of the procedure to be placed on the call stack. The new invocation has new storage for all data items in automatic storage, and that storage is unavailable to other invocations because it is local. (A data item that is defined in a subprocedure uses automatic storage unless the STATIC keyword is specified for the definition.) Note also that the automatic storage that is associated with earlier invocations is unaffected by later invocations. The new invocation uses the same static storage as the previous invocation, both the global static storage of the module, and the local static storage in the procedure.

Recursive calls are also allowed for programs whose main procedure is a linear-main procedure. A linear-main procedure can only be called through a program call, so when a linear-main procedure calls itself recursively, the program containing the linear-main procedure is called again. Otherwise, the behavior for a linear-main procedure calling itself recursively is the same as for an ordinary subprocedure calling itself recursively.

A cycle-main procedure that is on the call stack cannot be called until it returns to its caller. Therefore, be careful not to call a procedure that might call an already active cycle-main procedure.

Try to avoid situations that might inadvertently lead to recursive calls. For example, suppose there are three modules, as shown in Figure 61 on page 162.

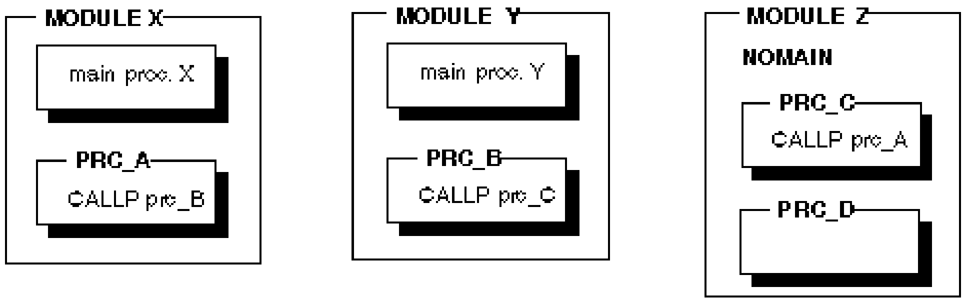


Figure 61. Three Modules, each with subprocedures

You are running a program where procedure A in module X calls procedure B in module Y. You are not aware of what procedure B does except that it processes some fields. Procedure B in turn calls procedure C, which in turn calls procedure A. Once procedure C calls procedure A, a recursive call has been made. The call stack sequence is shown in Figure 62 on page 163. Note that the most recent call stack entry is at the bottom.

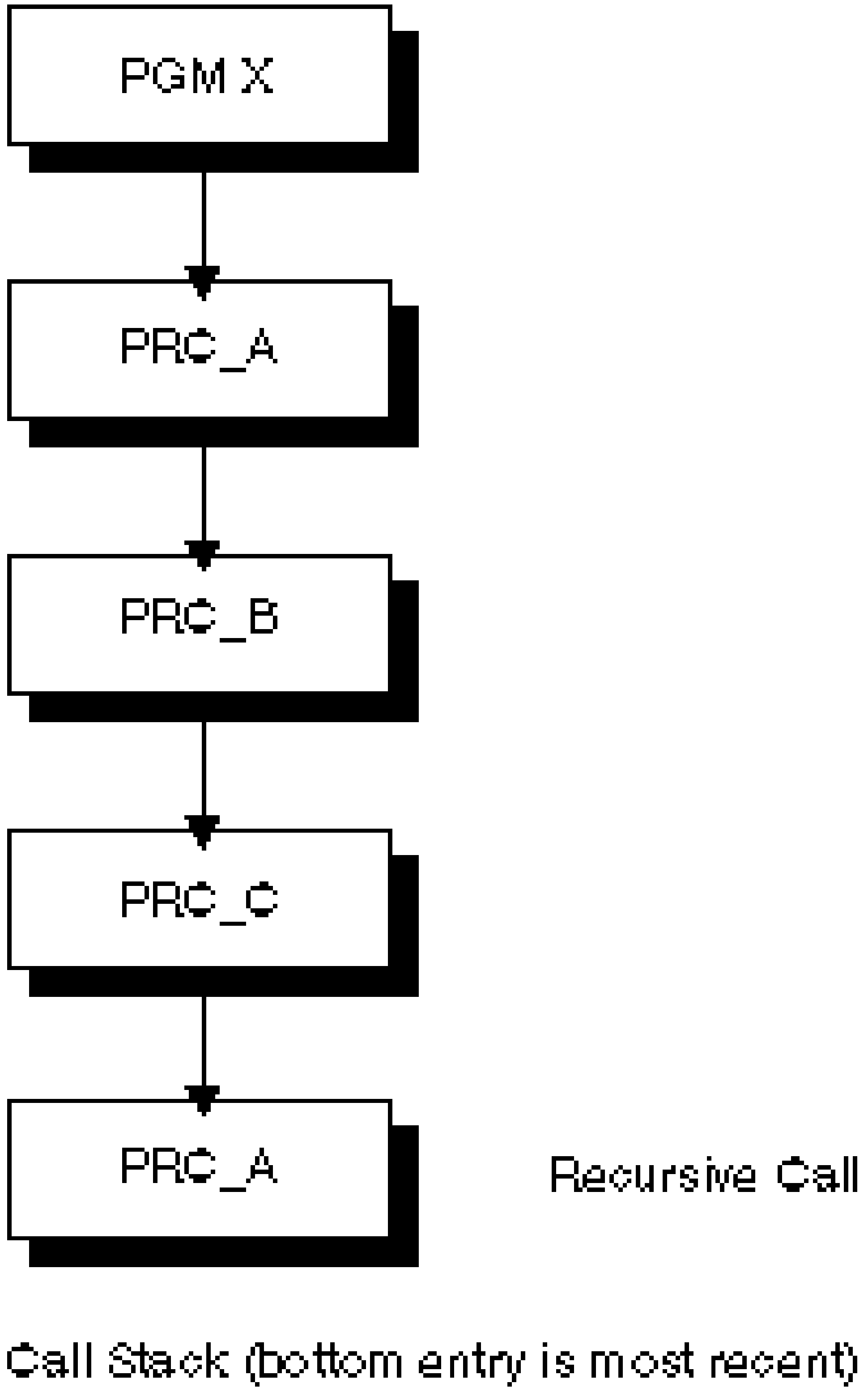


Figure 62. Recursive Call Stack To Be Avoided

So while subprocedures can be called recursively, if you are not aware that recursion is occurring, you may exhaust system resources.

Attention!

Unconditional recursive calls can lead to infinite recursion which leads to excessive use of system resources. Infinite recursion can be avoided with proper programming. In general, a proper recursive procedure begins with a test to determine if the desired result has been obtained. If it has been obtained, then the recursive procedure returns to the most recent caller.

Parameter-Passing Considerations

When designing a call interface, you must make a number of decisions in terms of how parameters will be passed. On the other hand, if you are the caller then most of the decisions have already been made for you. The following lists some of the parameter-passing considerations to keep in mind when you are designing a call interface.

- Compile-time parameter checking

  The call interface of a prototyped call is checked at compile time. This checking ensures that:

  - the data types are correctly used
  - correct files are passed to file parameters
  - all required parameters are passed
  - *OMIT is only passed where it is allowed.
- Parameter passing method

  Each HLL provides one or more ways of passing parameters. These may include: passing a pointer to the parameter value, passing a copy of the value, or passing the value itself.
- Passing operational descriptors

  Sometimes you may not be sure of the exact format of the data that is being passed to you. In this case you may request that operational descriptor be passed to provide additional information regarding the format of the passed parameters.
- 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 are expected, and the callee references a parameter for which no data was passed, then the callee will get an error.
- Passing less data

  If you pass a parameter and you pass too little data, your application may not work correctly. If changing the parameter, you may overwrite storage. If using the parameter, you may misinterpret the parameter. By prototyping the parameter, the compiler will check to see that the length is appropriate for the parameter.

  If the callee has indicated (through documentation or through that prototype) that a parameter can be shorter than the maximum length, you can safely pass shorter parameters. (Note, however, that the called procedure must be written in a way to handle less data than required.)
- Order of evaluation

  There is no guaranteed order for evaluation of parameters on a prototyped call. This fact may be important, if a parameter occurs more than once in the parameter list, and there is the possibility of side effects.
- Interlanguage call considerations

  Different HLLs support different ways of representing data as well as different ways of sending and receiving data between programs and procedures. In general, you should only pass data which has a data type common to the calling and called program or procedure, using a method supported by both.

  RPG file parameters are not related to file parameters of other HLLs; you can only pass an RPG file parameter to another RPG program or procedure.

Table 60 on page 165 associates the above considerations with the two types parameters: prototyped or non-prototyped.

Table 60. Parameter Passing Options

Parameter Option	**Prototyped**	**Not Prototyped**	**See Page**
Compile-time parameter checking	Yes		"Passing Prototyped Parameters" on page 167
Pass by reference	Yes	Yes	"Passing by Reference" on page 168
Pass by value	Yes (b)		"Passing by Value" on page 168
Pass by read-only reference	Yes		"Passing by Read-Only Reference" on page 168
Pass operational descriptors	Yes (b)	Yes (b)	"Using Operational Descriptors" on page 170
Pass *OMIT	Yes	Yes (b)	"Omitting Parameters" on page 171
Control parameter omission	Yes	Yes	"Leaving Out Parameters" on page 171
Get number of passed parameters	Yes	Yes	"Checking for the Number of Passed Parameters" on page 172
Disallow incorrect parameter length	Yes		"Passing Less Data Than Required" on page 177
Pass file parameters	Yes		Passing File Parameters
Note: (b) – applies to bound procedures only.			

Using a Prototyped Call

A prototyped call is one for which there is a prototype that is available to do parameter checking. The prototype may be explicitly specified, or it may be implicitly generated by the compiler from the procedure interface, if the procedure is specified in the same module as the call. A prototyped call has a much simpler call interface and offers more function. For example, using a prototyped call you can call (with the same syntax):

- Programs that are on the system at run time
- Exported procedures in other modules or service programs that are bound in the same program or service program
- Subprocedures in the same module

In RPG, prototyped calls are also known as free-form calls. **Free-form call** refers to the call syntax where the arguments for the call are specified using free-form syntax, much like the arguments for built-in functions. It contrasts with fixed-form call, where the arguments are placed in separate specifications. There are two ways to make a free-form call, depending on whether there is a return value that is to be used. If there is no return value, use the CALLP operation. If there is one, and you want to use the value that is returned, then place the prototyped procedure within an expression, for example, with EVAL. If you use CALLP to a procedure that returns a value, the return value is ignored.

Note: Only prototyped procedures can return values; prototyped programs cannot.

You can optionally code parentheses on procedure calls that do not have any parameters. This makes it easier to distinguish procedure calls from scalar variable names.

For information on passing prototyped parameters, see "Passing Prototyped Parameters" on page 167.

Using the CALLP Operation

You use the CALLP (Call a Prototyped procedure) operation to call a prototyped program or procedure written in any language. The CALLP operation uses the following extended-factor 2 syntax:

```
     C                   CALLP     NAME{ (PARM1 {:PARM2 ...}) }
```

In free-form calculations, you can omit CALLP if there are no operation extenders. The free-form operation can use either of the following forms:

```
      /free
          callp name { (parm1 { :parm2 ...} ) };
          name( {parm1 {:parm2 ... }} );
      /end-free
```

To call a prototyped program or procedure follow these general steps:

1. Include the prototype of the program or procedure to be called in the definition specifications. This step is optional if the procedure is in the same module as the call, and there are no other modules that call the procedure.
2. Enter the prototype name of the program or procedure in the extended Factor-2 field, followed by the parameters if any, within parentheses. Separate the parameters with a colon (:). Factor 1 must be blank.

The following example shows a call to a procedure Switch, which changes the state of the indicator that is passed to it, in this case *IN10..

```
     C                   CALLP     Switch(*in10)
```

A maximum of 255 parameters are allowed on a program call, and a maximum of 399 for a procedure call.

You can use CALLP from anywhere within the module. If the keyword EXTPGM is specified on the prototype, the call will be a dynamic external call; otherwise it will be a bound procedure call.

Note that if CALLP is used to call a procedure which returns a value, that value will not be available to the caller. If the value is required, call the prototyped procedure within an expression.

Calling within an Expression

If a prototyped procedure is defined to return a value then you must call the procedure within an expression if you want to make use of the return value. Use the procedure name in a manner that is consistent with the data type of the specified return value. For example, if a procedure is defined to return a numeric, then the call to the procedure within an expression must be where a numeric would be expected.

Figure 63 on page 167 shows the prototype for a procedure CVTCHR that takes a numeric input parameter and returns a character string. Figure 64 on page 167 shows how the procedure might be used in an expression.

```
      * Prototype for CVTCHR
      * - returns a character representation of the numeric parameter
      * Examples:   CVTCHR(5) returns '5                    '
      *             CVTCHR(15-124) returns '-109                   '
     D CVTCHR          PR            31A
     D   NUM                         30P 0   VALUE
```

Figure 63. Prototype for CVTCHR

```
     C                   EVAL      STRING = 'Address: ' +
     C                                      %TRIM(CVTCHR(StreetNum))
     C                                      + ' ' + StreetName
      * If STREETNUM = 427 and STREETNAME = 'Mockingbird Lane', after the
      * EVAL operation STRING = 'ADDRESS: 427 Mockingbird Lane'
```

Figure 64. Calling a Prototyped Procedure within an Expression

Examples of Free-Form Call

For examples of using the CALLP operation, see:

- Figure 22 on page 87
- Figure 44 on page 134
- Figure 129 on page 287
- Figure 72 on page 176
- Figure 142 on page 311

For examples of calling by using an expression, see:

- Figure 4 on page 64
- Figure 19 on page 84
- Figure 39 on page 121
- Figure 129 on page 287

Passing Prototyped Parameters

When you pass prototyped parameters:

- The compiler verifies, when compiling both the caller and the callee, that the parameter definitions match, provided that both are compiled using the same prototype.
- Fewer specifications are needed, since you do not need the PARM operations.

This section discusses the various options that are available when defining prototyped parameters, and the impact of these options on the call interface.

Parameter Passing Styles

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

- By reference
- By value
- By read-only reference

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

Passing by Reference

The default parameter passing style for ILE RPG is to pass by reference. Consequently, you do not have to code any keywords on the parameter definition to pass the parameter by reference. You should pass parameters by reference to a procedure when you expect the callee to modify the field passed. You may also want to pass by reference to improve run-time performance, for example, when passing large character fields. Note also that parameters that are passed on external program calls can only be passed by reference.

Passing by Value

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

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

To pass a parameter by value, specify the keyword VALUE on the parameter definition in the prototype, as shown in the figures below.

Note: IBM i program calls require that parameters be passed by reference. Consequently, you cannot pass a parameter by value to a program.

Passing by Read-Only Reference

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

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

When a parameter is passed by read-only reference, the compiler may copy the parameter to a temporary field and pass the address of the temporary. Some conditions that would cause this are: the passed parameter is an expression or the passed parameter has a different format.

Note: If the called program or procedure is compiled using a prototype in a language that enforces the read-only reference method (either ILE RPG using prototypes, or C), then the parameter will not be changed. If the called program or procedure does not use a prototype, then the compiler cannot ensure that the parameter is not changed. In this case, the person defining the prototype must be careful when specifying this parameter-passing method.

To pass a parameter by read-only reference, specify the keyword CONST on the definition specification of the parameter definition in the prototype. Figure 67 on page 170 shows an example of a prototype definition for the ILE CEE API CEETSTA (Test for omitted argument).

Advantages of passing by value or read-only reference

Passing by value or read-only reference allows you to:

- Pass literals and expressions as parameters.
- Pass parameters that do not match exactly the type and length that are expected.
- Pass a variable that, from the caller's perspective, will not be modified.

One primary use for passing by value or read-only reference is that you can allow less stringent matching of the attributes of the passed parameter. For example, if the definition is for a numeric field of type packed-decimal and length 5 with 2 decimal positions, you must pass a numeric value, but it can be:

- A packed, zoned or binary constant or variable, with any number of digits and number of decimal positions
- A built-in function returning a numeric value
- A procedure returning a numeric value

- A complex numeric expression such as

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

If the prototype requires an array of 4 elements, the passed parameter can be:

- An array with fewer than 4 elements. In this case, the remaining elements in the received parameter will contain the default value for the type.
- An array with 4 elements. In this case, each element of the received parameter will correspond to an element of the passed parameter.
- An array with more than 4 elements. In this case, some of the elements of the passed array will not be passed to the received parameter.
- A non-array. In this case, each element of the received parameter will contain the passed parameter value.

Choosing between parameter passing styles

If you are calling an existing program or procedure, you must pass the parameters in the way the procedure expects them, either by reference or by value. If the parameter must be passed by reference, and it will not be modified by the called procedure program or procedure, pass it by read-only reference (using the CONST keyword). When you are free to choose between passing by value or by read-only reference, pass by read-only reference for large parameters. Use the following general guideline:

- If the parameter is numeric or pointer, **and it is not an array**, pass it by read-only reference or by value. Passing these data types by value may have a very slight performance benefit.
- Otherwise, pass it by read-only reference.

```
      *-------------------------------------------------------------
      *  The procedure returns a value of a 10-digit integer value.
      *  The 3 parameters are all 5-digit integers passed by value.
      *-------------------------------------------------------------
     D MyFunc          PR            10I 0 EXTPROC('DO_CALC')
     D                                5I 0 VALUE
     D                                5I 0 VALUE
     D                                5I 0 VALUE
     ....
```

Figure 65. Prototype for Procedure DO_CALC with VALUE Parameters

```
     P DO_CALC         B                   EXPORT
      *-------------------------------------------------------------
      * This procedure performs a function on the 3 numeric values
      * passed to it as value parameters.  It also returns a value.
      *-------------------------------------------------------------
     D DO_CALC         PI            10I 0
     D    Term1                       5I 0 VALUE
     D    Term2                       5I 0 VALUE
     D    Term3                       5I 0 VALUE
     D Result          S             10I 0
     C                   EVAL      Result = Term1 ** 2 * 17
     C                                    + Term2      * 7
     C                                    + Term3
     C                   RETURN    Result * 45 + 23
     P                 E
```

Figure 66. Procedure Interface Definition for DO_CALC Procedure

```
 *-----------------------------------------------------------------
 * CEETSTA (Test for omitted argument) -- ILE CEE API
 *     1. Presence flag                       Output   Binary(4)
 *     2. Argument number                     Input    Binary(4)
 *-----------------------------------------------------------------
D CEETSTA         PR                  EXTPROC('CEETSTA')
D   Present                     10I 0
D   ArgNum                      10I 0   CONST
D   Feedback                    12A     OPTIONS(*OMIT)
...
D HaveParm        S             10I 0
...
C                   CALLP    CEETSTA(HaveParm : 3 : *OMIT)
C                   IF       HaveParm = 1
 *        do something with third parameter
C                   ENDIF
```

Figure 67. Prototype for ILE CEE API CEETSTA with CONST Parameter

The second parameter passed to CEETSTA can be any numeric field, a literal, a built-in function, or expression.

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.

Many ILE bindable APIs expect operational descriptors. If any parameter is defined as 'by descriptor', then you should pass operational descriptors to the API. An example of this is the ILE CEE API CEEDATM (Convert Seconds to Character Timestamp). The second and third parameters require an operational descriptor.

Note: Currently, the ILE RPG compiler only supports operational descriptors for character and graphic types. Operational descriptors are not available for arrays or tables, or for data of type numeric, date, timestamp, basing pointer or procedure pointer. In addition, operational descriptors are not available for data structures for non-protototyped calls made using CALLB. However, for prototyped calls, data structures are considered to be character data, and operational descriptors are available.

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 request operational descriptors for both prototyped and non-prototyped parameters. For prototyped parameters, you specify the keyword OPDESC on the prototype definition. For non-prototyped parameters, you specify (D) as the operation code extender of the CALLB operation. In either case, operational descriptors are then built by the calling procedure and passed as hidden parameters to the called procedure. Operational descriptors will not be built for omitted parameters.

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 bound calls. Furthermore, for non-prototyped calls, an error message will be issued by the compiler if the 'D' operation code extender is specified on a CALL operation.

Figure 68 on page 171 shows an example of the keyword OPDESC.

```
      *-------------------------------------------------------------
      *  Len returns a 10-digit  integer value.  The parameter
      * is a character string passed by read-only reference.
      * Operational descriptors are required so that Len knows
      * the length of the parameter.
      *  OPTIONS(*VARSIZE) is required so that the parameter can
      *  be less than 32767 bytes.
      *-------------------------------------------------------------
     D Len             PR            10I 0 OPDESC
     D                            32767A   OPTIONS(*VARSIZE) CONST
```

Figure 68. Requesting Operational Descriptors for a Prototyped Procedure

For an example of how to use operational descriptors see "Sample Service Program" on page 132. 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 program or procedure, you may sometimes want to leave out a parameter. It may be that it is not relevant to the called procedure. For example, this situation might arise when you are calling the ILE bindable APIs. Another reason might be that you are calling an older procedure that does not handle this particular parameter. If you need to omit a parameter on a call, you have two choices:

- Specify OPTIONS(*OMIT) and pass *OMIT
- Specify OPTIONS(*NOPASS) and do not pass the parameter.

The primary difference between the two methods has to do with how you check to see if a parameter has been omitted. In either case, an omitted parameter cannot be referenced by the called procedure; if it is, unpredictable results will occur. So if the called procedure is designed to handle different numbers of parameters, you will have to check for the number of parameters passed. If *OMIT is passed, it will 'count' as a parameter.

*Passing *OMIT*

You can pass *OMIT for a prototyped parameter if the called procedure is aware that *OMIT might be passed. In other words, you can pass *OMIT if the keyword OPTIONS(*OMIT) is specified on the corresponding parameter definition in the prototype. When *OMIT is specified, the compiler will generate the necessary code to indicate to the called procedure that the parameter has been omitted.

Note: *OMIT can only be specified for parameters passed by reference.

To determine if *OMIT has been passed to an ILE RPG procedure, use the %ADDR built-in function to check the address of the parameter in question. If the address is *NULL, then *OMIT has been passed. You can also use the CEETSTA (Check for Omitted Argument) bindable API. (See Figure 67 on page 170 for a brief example.)

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.

See "Sample Service Program" on page 132 for an example of using *OMIT.

Leaving Out Parameters

The other way to omit a parameter is to simply leave it out on the call. This must be expected by the called procedure, which means that it must be indicated on the prototype. To indicate that a prototyped parameter does not have to be passed on a call, specify the keyword OPTIONS(*NOPASS) on the corresponding parameter definition. Note that all parameters following the first *NOPASS one must also be specified with OPTIONS(*NOPASS).

You can specify both *NOPASS and *OMIT for the same parameter, in either order, that is, OPTIONS(*NOPASS:*OMIT) or OPTIONS(*OMIT:*NOPASS).

As an example of OPTIONS(*NOPASS), consider the system API QCMDEXC (Execute Command) which has an optional third parameter. To allow for this parameter, the prototype for QCMDEXC could be written as shown in Figure 69 on page 172.

```
  *-------------------------------------------------------------
  *  This prototype for QCMDEXC defines three parameters:
  *   1- a character field that may be shorter in length
  *      than expected
  *   2- any numeric field
  *   3- an optional character field
  *-------------------------------------------------------------
 D qcmdexc         PR                  EXTPGM('QCMDEXC')
 D   cmd                       3000A   OPTIONS(*VARSIZE)  CONST
 D   cmdlen                      15P 5 CONST
 D                                3A   CONST OPTIONS(*NOPASS)
```

Figure 69. Prototype for System API QCMDEXC with Optional Parameter

Checking for the Number of Passed Parameters

At times it may be necessary to check for the number of parameters that are passed on a call. Depending on how the procedure has been written, this number may allow you to avoid references to parameters that are not passed. For example, suppose that you want to write a procedure which will sometimes be passed three parameters and sometimes four parameters. This might arise when a new parameter is required. You can write the called procedure to process either number depending on the value that is returned by the built-in function %PARMS. New calls may pass the parameter. Old calls can remain unchanged.

%PARMS does not take any parameters. The value returned by %PARMS also includes any parameters for which *OMIT has been passed, and it also includes the additional first parameter that handles the return value for a procedure that has the RTNPARM keyword specified. For a cycle-main procedure, %PARMS returns the same value as contained in the *PARMS field in a PSDS, although to use the *PARMS field, you must also code the PSDS.

If you want to check whether a particular parameter was passed to a procedure, you can use the %PARMNUM built-in function to obtain the number of the parameter. The value returned by %PARMNUM reflects the true parameter number if the RTNPARM keyword was coded for the procedure.

For both *PARMS and %PARMS, if the number of passed parameters is not known, the value -1 is returned. (In order to determine the number of parameters passed, a minimal operational descriptor must be passed. ILE RPG always passes one on a call; however other ILE languages may not.) If the main procedure is not active, *PARMS is unreliable. It is not recommended to reference *PARMS from a subprocedure.

Using %PARMS

In this example, a procedure FMTADDR has been changed several times to allow for a change in the address information for the employees of a company. FMTADDR is called by three different procedures. The procedures 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 procedures 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 procedure 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 or province values or both. Figure 70 on page 174 shows the source for this procedure. Figure 71 on page 175 shows the source for /COPY member containing the prototype.

The main logic of FMTADDR is as follows:

1. Check to see how many parameters were passed by using %PARMS. This built-in function returns the number of passed parameters.
   - 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
      *
      * Interface 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)
      *=================================================================*
      * Pull in the prototype from the /COPY member
      /COPY  FMTADDRP
     DFmtAddr          PI
     D Address                       70
     D Street#                        5  0 CONST
     D Street                        20    CONST
     D P_City                        15    OPTIONS(*NOPASS) CONST
     D P_Province                    15    OPTIONS(*NOPASS) CONST
      *-----------------------------------------------------------------*
      * Default values for parameters that might not be passed.
      *-----------------------------------------------------------------*
     D City            S             15    INZ('Toronto')
     D Province        S             15    INZ('Ontario')
      *-----------------------------------------------------------------*
      * Check whether the province parameter was passed.  If it was,
      * replace the default with the parameter value.
      *-----------------------------------------------------------------*
     C                   IF        %PARMS > 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        %PARMS > 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           EVAL     ADDRESS = %TRIMR(CSTREET#) + ' ' +
     C                              %TRIMR(CITY) + ' ,' +
     C                              %TRIMR(PROVINCE)
     C                   RETURN
```

Figure 70. Source for procedure FMTADDR

```
      *=================================================================*
      * 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
```

```
      *=================================================================*
      * Prototype for FMTADDR - format an address
      *=================================================================*
     DFmtAddr          PR
     D  addr                         70
     D  strno                         5  0 CONST
     D  st                           20    CONST
     D  cty                          15    OPTIONS(*NOPASS) CONST
     D  prov                         15    OPTIONS(*NOPASS) CONST
```

Figure 71. Source for /COPY member with Prototype for Procedure FMTADDR

Figure 72 on page 176 shows the source for the procedure PRTADDR. This procedure serves to illustrate the use of FMTADDR. For convenience, the three procedures which would each call FMTADDR are combined into this single procedure. Also, for the purposes of the example, the data is program-described.

Since PRTADDR is 'three procedures-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 procedure PRTADDR ends.

```
 *=================================================================*
 * PRTADDR - Print an address
 *          Calls FmtAddr to format the address
 *=================================================================*
FQSYSPRT   O    F   80        PRINTER
 *-----------------------------------------------------------------*
 * Prototype for FmtAddr
 *-----------------------------------------------------------------*
DFmtAddr          PR
D  addr                             70
D  strno                             5  0
D  st                               20
D  cty                              15    OPTIONS(*NOPASS)
D  prov                             15    OPTIONS(*NOPASS)
DAddress          S                 70
 *-----------------------------------------------------------------*
 * 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                   CALLP     FMTADDR (Address:Street#1(X):StreetNam1(X))
C                   EXCEPT
C                   ENDDO
```

Figure 72. Source for procedure PRTADDR

```
      *-----------------------------------------------------------------*
      * 'Program 2'- Use of FMTADDR before province parameter was added.*
      *-----------------------------------------------------------------*
     C                   DO        2             X                 5 0
     C                   CALLP     FMTADDR (Address:Street#2(X):
     C                             StreetNam2(X):City2(X))
     C                   EXCEPT
     C                   ENDDO
      *-----------------------------------------------------------------*
      * 'Program 3' - Use of FMTADDR after province parameter was added.*
      *-----------------------------------------------------------------*
     C                   DO        2             X                 5 0
     C                   CALLP     FMTADDR (Address:Street#3(X):
     C                             StreetNam3(X):City3(X):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
```

To create these programs, follow these steps:

1. To create FMTADDR, using the source in Figure 70 on page 174, type:

   ```
   CRTRPGMOD MODULE(MYLIB/FMTADDR)
   ```

2. To create PRTADDR, using the source in Figure 72 on page 176, type:

   ```
   CRTRPGMOD MODULE(MYLIB/PRTADDR)
   ```

3. To create the program, PRTADDR, type:

   ```
   CRTPGM PGM(MYLIB/PRTADDR) MODULE(PRTADDR FMTADDR)
   ```

4. 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
   ```

Passing Less Data Than Required

When a parameter is prototyped, the compiler will check to see that the length is appropriate for the parameter. If the callee has indicated (through documentation or through that prototype) that a parameter can be shorter than the maximum length, you can safely pass shorter parameters.

Figure 73 on page 178 shows the prototype for QCMDEXC, where the first parameter is defined with OPTIONS(*VARSIZE) meaning that you can pass parameters of different lengths for the first parameter. Note that OPTIONS *VARSIZE can only be specified for a character field, a UCS-2 field, a graphic field, or an array.

```
     *-------------------------------------------------------------
     *  This prototype for QCMDEXC defines three parameters.  The
     *  first parameter can be passed character fields of
     *  different lengths, since it is defined with *VARSIZE.
     *-------------------------------------------------------------
    D qcmdexc         PR                  EXTPGM('QCMDEXC')
    D   cmd                       3000A   OPTIONS(*VARSIZE) CONST
    D   cmdlen                      15P 5 CONST
    D                                3A   CONST OPTIONS(*NOPASS)
```

*Figure 73. Prototype for System API QCMDEXC with *VARSIZE Parameter*

Passing File Parameters

You can use the LIKEFILE keyword to indicate that a prototyped parameter is a file. When make a call using the prototype, the file that you pass must either be the file that was specified on the LIKEFILE parameter of the prototype, or it must be a file that is related through LIKEFILE File-specification keywords to that file. For example, if you specify LIKEFILE(MYFILE) on the prototype, and you have another File specification that defines file OTHERFILE using LIKEFILE(MYFILE), then you can pass either MYFILE or OTHERFILE on the call. See

For more information *on file parameters* and *variables associated with files* see the chapter about general file considerations in the "WebSphere Development Studio ILE RPG Reference."

Order of Evaluation

There is no guaranteed order for evaluation of parameters on a prototyped call. This fact may be important when using parameters that cause side effects, as the results may not be what you would expect.

A **side effect** occurs if the processing of the parameter changes:

- The value of a reference parameter
- The value of a global variable
- An external object, such as a file or data area

If a side effect occurs, then, if the parameter is used elsewhere in the parameter list, then the value used for the parameter in one part of the list may not be the same as the value used in another part. For example, consider this call statement.

```
                 CALLP         procA (fld : procB(fld) : fld)
```

Assume that procA has all value parameters, and procB has a reference parameter. Assume also that *fld* starts off with the value 3, and that procB modifies *fld* to be 5, and returns 10. Depending on the order in which the parameters are evaluated, procA will receive either 3, 10, and 5 or possibly, 3, 10, and 3. Or possibly, 5, 10, and 3; or even 5, 10, and 5.

In short, it is important to be aware of the possibility of side effects occurring. In particular, if you are providing an application for third-party use, where the end user may not know the details of some of the procedures, it is important ensure that the values of the passed parameters are the expected ones.

Interlanguage Calls

When passing or receiving data from a program or procedure written in another language, it is important to know whether the other language supports the same parameter passing methods and the same data types as ILE RPG. "RPG Parameter Passing Methods" on page 179 shows the different parameter passing methods allowed by ILE RPG and, where applicable, how they would be coded in the other the ILE languages. The table also includes the OPM RPG/400 compiler for comparison.

RPG Parameter Passing Methods

Table 61. Passing By Reference

Calling mechanism	Example

ILE RPG – prototype	`D  proc      PR`
	`D    parm            1A`
	`C        CALLP  proc(fld)`

ILE C	`void proc(char *parm);`
	`proc(&fld);`

ILE COBOL	`CALL PROCEDURE "PROC" USING BY REFERENCE PARM`

RPG – non-prototyped	`C        CALL   'PROC'`
	`C        PARM                  FLD`

ILE CL	`CALL PROC (&FLD)`

Table 62. Passing By Value

Calling mechanism	Example

ILE RPG – prototype	`D  proc      PR`
	`D    parm            1A   VALUE`
	`C        CALLP  proc('a')`

ILE C	`void proc(char parm);`
	`proc('a');`

ILE COBOL	`CALL PROCEDURE "PROC" USING BY VALUE PARM`
RPG – non-prototyped	N/A
ILE CL	N/A

Table 63. Passing By Read-Only Reference

Calling mechanism	Example

ILE RPG – prototype	`D  proc      PR`
	`D    parm            1A     CONST`
	`C        CALLP  proc(fld)`

ILE C	`void proc(const char *parm);`
	`proc(&fld);`

ILE COBOL	N/A[1]
RPG – non-prototyped	N/A
ILE CL	N/A

Note:

1. Do not confuse passing by read-only reference with COBOL's passing BY CONTENT. In RPG terms, to pass Fld1 by content, you would code:

   ```
        C                    PARM      Fld1          TEMP
   ```

   Fld1 is protected from being changed, but TEMP is not. There is no expectation that the parameter will not be changed.

For information on the data types supported by different HLLs, consult the appropriate language manual.

Interlanguage Calling Considerations

1. To ensure that your RPG procedure will communicate correctly with an ILE CL procedure, code `EXTPROC(*CL:'procedurename')` on the prototype for the ILE CL procedure or on the prototype for the RPG procedure that is called by the ILE CL procedure.
2. To ensure that your RPG procedure will communicate correctly with an ILE C procedure, code `EXTPROC(*CWIDEN:'procedurename')` or `EXTPROC(*CNOWIDEN:'procedurename')` on the prototype for the ILE C procedure or on the prototype for the RPG procedure that is called by the ILE C procedure. Use *CNOWIDEN if the ILE C source contains `#pragma argument(procedure-name,nowiden)` for the procedure; otherwise, use *CWIDEN.
3. If you want your RPG procecure to be used successfully by every ILE language, do not specify any special value on the EXTPROC keyword. Instead, avoid the following types for parameters that are passed by value or return values:
   - Character of length 1 (1A or 1N)
   - UCS-2 of length 1 (1C)
   - Graphic of length 1 (1G)
   - 4-byte float (4F)
   - 1-byte or 2-byte integer or unsigned (3I, 3U, 5I, or 5U)
4. RPG procedures can interact with ILE C/C++ procedures that use 8-byte pointers. However, the ILE C/C++ procedures must use 16-byte pointers for parameters. See the *IBM Rational Development Studio for i: ILE C/C++ Compiler Reference*.
5. RPG file parameters prototyped with the LIKEFILE keyword are not interchangeable with file parameters from other languages. For example, you cannot pass an RPG file to a C function that is expecting a FILE or RFILE parameter. Similarly, a C function cannot pass a FILE or RFILE parameter to an RPG procedure if the RPG parameter was prototyped with the LIKEFILE keyword.

Using the Fixed-Form Call 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 to programs or procedures that are not prototyped. 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) or an LR indicator (positions 75 and 76) or both.

   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 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 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 or module is doing the call.

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

  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 begin with underscores or contain blanks and you do not have to be concerned with these.
- **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 character and graphic field and subfield.

  For more information on operational descriptors, see “Using Operational Descriptors” on page 170.
- There are further restrictions that apply when using the CALL or CALLB operation codes. For a detailed description of these restrictions, see the *IBM Rational Development Studio for i: ILE RPG Reference*.

Examples of CALL and CALLB

For examples of using the CALL operation, see:

- “Sample Source for Debug Examples” on page 286, for example of calling an RPG program.

For examples of using the CALLB operation, see:

- Figure 46 on page 137, for an example of calling a procedure in a service program.
- Figure 58 on page 156, for an example of calling bindable APIs.
- “CUSMAIN: RPG Source” on page 395, for an example of a main inquiry program calling various RPG procedures.

Passing Parameters Using PARM and PLIST

When you pass parameters using fixed-form call, you must pass parameters using the PARM and PLIST operations. All parameters are passed by reference. You can specify that an operational descriptor is to be passed and can also indicate that a parameter is omitted.

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 64 on page 182 shows how factor 1 and factor 2 are used.

Table 64. 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 either:

- Check %PARMS to determine the number of parameters passed. For an example using %PARMS, see "Checking for the Number of Passed Parameters" on page 172.
- 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, using %ADDR(parameter) = *NULL. For more information, refer to "Omitting Parameters" on page 171.

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.
- The following rules apply to *OMIT for non-prototyped parameters:
  - *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.
- There are other restrictions that apply when using the PARM operation code. For a detailed description of these restrictions, see the *IBM Rational Development Studio for i: ILE RPG Reference*.

For examples of the PARM operation see:

- Figure 48 on page 140
- Figure 43 on page 133
- Figure 141 on page 309

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 PLISTs can appear in a procedure. However, only one *ENTRY PLIST can be specified, and only in the main procedure.

For examples of the PLIST operation see Figure 48 on page 140 and Figure 141 on page 309.

Returning from a Called Program or Procedure

When a program or procedure returns, its call stack entry is removed from the call stack. (If it is a program, the program entry procedure is removed as well.) A procedure ends abnormally when something outside the procedure ends its invocation. For example, this would occur if an ILE RPG procedure X calls another procedure (such as a CL procedure) that issues an escape message directly to the procedure calling X. This would also occur if the procedure gets an exception that is handled by an exception handler (a *PSSR or error indicator) of a procedure further up the call stack.

Because of the cycle code associated with main procedures, their return is also associated with certain termination routines. This section discusses the different ways that main procedures and subprocedures can return, and the actions that occur with each.

Returning from a Main Procedure

A return from a main procedure causes the following to occur:

- If it is a cycle-main procedure, and LR is on, then global files are closed and other resources are freed.
- The procedure's call stack entry is removed from the call stack.
- If the procedure was called by the program entry procedure, then that program entry procedure is also removed from the call stack.

A cycle-main 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 cycle-main 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 *IBM Rational Development Studio for i: ILE RPG Reference*.

Normal End for a Cycle-Main Procedure

A cycle-main 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 cycle-main procedure also ends normally if:

- The RETURN operation (with a blank factor 2) 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 cycle-main procedure ends normally, the following occurs:

- The Factor-2-to-Result-field move of a *ENTRY 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 global files that are open are closed.
- A return code is set to indicate to the caller that the procedure has ended normally, and control then returns to the caller.

On the next call to the cycle-main procedure, with the exception of exported variables, a fresh copy is available for processing. (Exported variables defined with the EXPORT keyword. They are initialized only once, when the program is first activated in an activation group. They retain their last assigned value on a new call, even if LR was on for the previous call. If you want to re-initialize them, you have to reset them manually.)

TIP

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 for a Cycle-Main Procedure

A cycle-main 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 inquiry 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 307).
- An H1 through H9 indicator is on when a RETURN operation (with a blank factor 2) is processed.
- An H1 through H9 indicator is on when last record (LR) processing occurs in the RPG cycle.

When a cycle-main procedure ends abnormally, the following occurs:

- All global files that are open are closed.
- Any data areas locked by the procedure are unlocked.
- If the cycle-main procedure ended because of a cancel reply to an inquiry message, then it was a function check that caused the abnormal end. In this case, the function check is percolated to the caller. If it ended because of an error subroutine ending with '*CANCL', then escape message RNX9001 is issued directly to the caller. Otherwise the caller will see whatever exception caused the abnormal end.

On the next call to the procedure, a fresh copy is available for processing. (For more information on exception handlers, see "Using RPG-Specific Handlers" on page 298.)

Returning without Ending for a Cycle-Main Procedure

A cycle-main 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 (with a blank factor 2) 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 *IBM Rational Development Studio for i: ILE RPG Reference*)

If you call a cycle-main 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. However, there are three exceptions:

- 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.
- If another procedure in the same module was called in between, then the results are unpredictable.

You can use either the RETURN operation (with a blank factor 2) 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, and the H1 through H9 indicators. A return will cause an end if the LR indicator or any of the halt indicators is on and either of the following conditions is true:

- A RETURN operation is done
- The RT would cause a return without an end

Returning from a Subprocedure

This section applies to ordinary subprocedures and to linear-main procedures.

A subprocedure returns normally when a RETURN operation is performed successfully or when the last statement in the procedure (not a RETURN operation) is processed. If the subprocedure has any local files in automatic storage, they will be closed when the subprocedure ends. Otherwise, other than the removal of the subprocedure from the call stack no termination actions are performed until the cycle-main procedure, if any, of the program ends. In other words, all the actions listed for the normal end of a cycle-main procedure take place only for the main procedure.

A subprocedure ends abnormally and control returns to the calling procedure when an unhandled exception occurs. Any local files in automatic storage are closed. Other than that, no further actions occur until the cycle-main procedure ends.

If the module is a cycle module, and the main procedure is never called (and therefore cannot end) then any files, data areas, etcetera, will not be closed. If you think this might arise for a subprocedure, you should code a termination procedure that gets called when the subprocedure ends. This is especially true if the subprocedure is in a module with NOMAIN specified on the control specification.

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* termination processing as described above for main procedures and subprocedures. 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 then follow abnormal termination as described above.

Note: You cannot use either of these APIs in a program created with DFTACTGRP(*YES), since procedure calls are not allowed in these procedures.

Note that if the cycle-main procedure is not active, or if there is no cycle-main, then nothing will get closed or freed. In this case, you should enable an ILE cancel handler, using CEERTX. If the cancel handler is in the same module, it can close the files, unlock the data areas, and perform the other termination actions.

For more information on CEETREC and CEE4ABN, refer to the *CL and APIs* section of the *Programming* category in the **IBM i Information Center** at this Web site - http://www.ibm.com/systems/i/infocenter/.

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 to call procedures, that is, the CALLP operation or the CALLB operation. If the API returns a value and you want to use it, call the API in an expression. For the information required to define a prototype for an API, see the description of the API in the *CL and APIs* section of the *Programming* category in the **IBM i Information Center** at this Web site - http://www.ibm.com/systems/i/infocenter/. Figure 74 on page 186 shows a sample 'call' to a bindable API.

```
D CEExxxx         PR                  EXTPROC('CEExxxx')
D   parm1   ...
D   ...
C                   CALLP   CEExxxx( parm1 : parm2 : … :
                                     parmn : feedback)
 or
C                   CALLB   'CEExxxx'
C                   PARM                   parm1
C                   PARM                   parm2
                    ...
C                   PARM                   parmn
C                   PARM                   feedback
```

Figure 74. 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.

Note: Bindable APIs cannot be used if DFTACTGRP(*YES) is specified on the CRTBNDRPG command.

For more information on bindable APIs, refer to the *CL and APIs* section of the *Programming* category in the **IBM i Information Center** at this Web site - http://www.ibm.com/systems/i/infocenter/.

Examples of Using Bindable APIs

For examples of using bindable APIs, see:

- "Sample Service Program" on page 132, for an example of using CEEDOD
- "Managing Your Own Heap Using ILE Bindable APIs" on page 153. for an example of using CEEGTST, CEEFRST, and CEECZST.
- "Using a Condition Handler" on page 308, for an example of using CEEHDLR and CEEHDLU.

- “Using Cancel Handlers” on page 313, for an example of using CEERTX and CEEUTX.

Calling a Graphics Routine

ILE RPG supports the use of the CALL or CALLP operation to call IBM i 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 variable). 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 and cannot have a float format.

The procedure that processes the CALL does not implicitly start or end IBM i graphics routines.

For more information on IBM i Graphics, graphics routines and parameters, see the *GDDM Programming Guide* manual and the *GDDM Reference*.

Note: You can call IBM i Graphics using the CALL operation. You can also use CALLP if you define a prototype for the routine and specify the EXTPGM keyword on the prototype. 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 supports the use of the following special routines using the CALL and PARM operations or the CALLP operation:

- 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. You can use CALLP if you define a prototype for the 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.

Storage Model

The integrated language environment (ILE) offers two storage models, single-level and teraspace. Modules, programs and service programs can be created to use one of these storage models, or they can be created to inherit their caller's storage model.

All the programs and service programs called into an activation group must use the same storage model. If the first program or service program called into an activation group uses the single-level storage model, then all other programs and service programs in the same activation group must use either the single-level or the inherit storage model. Simiarly, if the first program or service program called into an activation group uses the teraspace storage model, then all other programs and service programs in the same activation group must use either the teraspace or the inherit storage model.

A program or service program may be created from modules that use the single-level storage model and the inherit storage model, or from modules that use the teraspace storage model and the inherit storage model. A program or service program cannot be created from modules that use both the single-level storage model and the teraspace storage model.

Considerations for the single-level storage model

- There is a limitation of 16MB of automatic storage for a single procedure.
- There is a limitation of a total of 16MB automatic storage for all the procedures on the call stack.

Considerations for the teraspace storage model

- There are no practical limits to automatic storage.
- Using the teraspace storage model provides access to service programs written in C and C++ that use faster 8 byte pointers. However, the C or C++ functions in the service program must use 16 byte pointers for parameters. See the *IBM Rational Development Studio for i: ILE C/C++ Compiler Reference*.

Considerations for the inherit storage model

- The activation group must be *CALLER.
- A program or service program with the inherit storage model can be called from a program or service program that uses any storage model.
- The actual storage model is determined at runtime by the storage model of the caller.
- There is a limit of 16MB automatic storage for a single procedure at compile time.
- The runtime limits on automatic storage depend on the actual storage model at runtime.

Recommendations for the storage model of programs and service programs

- Consider using STGMDL(*INHERIT) for ACTGRP(*CALLER) programs and service programs, unless the benefits of the teraspace storage model are always required by the program or service program.
- If programs and service programs are created with named activation groups, consider using a naming convention to identify teraspace activation groups. For example, you could end the teraspace activation group names with TS. This conforms to the way the activation group name is chosen when you specify ACTGRP(*STGMDL) for CRTBNDRPG or CRTPGM; in that case QILETS is used for teraspace storage model and QILE is chosen for single-level storage model.
- Avoid creating teraspace storage model and single-level storage model programs with the same activation group attribute. For example, assume that TERAPGM is a teraspace storage model program and SLSPGM is a single-level storage model program, and both TERAPGM and SLSPGM are compiled to use ACTGRP(MYACTGRP). If TERAPGM is called first, then activation group MYACTGRP would be created as a teraspace storage model activation group and any attempt to call SLSPGM would fail due to a storage model mismatch. Similarly, if SLSPGM is called first, then activation group MYACTGRP would be created as a single-level storage model activation group and any attempt to call TERAPGM would fail due to a storage model mismatch.

See *ILE Concepts* for more information.

Multithreading Considerations

When you run in a multithreaded environment, there are many thread-safety issues that you must consider. Before you embark on writing a multithreaded application, you should become familiar with the concepts of multithreading; see information on *multithreaded applications* at: http://www.ibm.com/systems/infocenter/. You should also become familiar with the common programming errors that are made when coding multithreaded applications; see *common multithreaded programming errors* at: http://www.ibm.com/systems/infocenter/. Pay special attention to the problems you may encounter using database files, and using commitment control in a multithreaded environment.

One important issue for thread safety is the handling of static storage. There are two ways of ensuring thread-safe handling of static storage in your ILE RPG modules. You can have separate static storage for each thread by specifying THREAD(*CONCURRENT), or you can limit access to the module to only one thread at a time by specifying THREAD(*SERIALIZE). See Multithreaded Applications for a comparison of the two modes of thread-safety.

The remainder of this discussion assumes that you are familiar with these concepts.

Running Concurrently in Multiple Threads

When you specify THREAD(*CONCURRENT), two or more threads could be running different procedures in the same module, the same procedure, or even the same statement. When two or more threads are running in the same module or in the same procedure, they each default to having their own instance of the static storage in the module and its procedures. This storage is referred to as thread-local storage. For example, if the two threads happen to be running in the loop below, one thread could be processing the "IF" statement after reading the twentieth record of the file, with variable "count" having a value of 7; the other thread could be processing the "READ" statement after reading the fourth record of the file, with variable "count" having a value of 1.

```
      read rec ds;
      count = 0;
      dow not %eof(file);
          if (ds.amtOwing > ds.max.Owing);
            handleAccount (ds);
            count += 1;
         endif;
         read rec ds;
      enddo;
```

If you have chosen to define a variable with STATIC(*ALLTHREAD), then all threads will use the same instance of that variable.

CAUTION: RPG does not provide any protection against two threads trying to change an all-thread static variable at the same time, or against one thread trying to change the variable while another thread is checking its value. See All- Thread Static Variables for more information.

If you want to ensure that some code is only used by one thread at a time, you can place the code in a serialized procedure (SERIALIZE keyword on the Procedure-Begin specification). Note that each serialized procedure has its own serialization mechanism; one thread can be running in one serialized procedure while another thread is running in a different serialized procedure in the same module.

Another way to ensure that the code is only used by one thread at a time is to put the code in a procedure to a thread-serialized module.

Running Serialized in Multiple Threads

Specifying THREAD(*SERIALIZE) will protect most of your variables and all your internal control structures from being accessed improperly by multiple threads. The thread-serialized module will be locked when a thread starts running any procedure in the module, and only unlocked when the thread is no longer running in the module. When the module is locked, no other thread can run a procedure in the module. If another thread is trying to call a procedure in the module, it must wait for the module to be unlocked before it can run the procedure. This serialized access ensures that only one thread is active in a thread-serialized module, within an activation group, at any one time.

Activation Group Considerations for the THREAD keyword

- Avoid running in the default activation group when the THREAD keyword is specified.
  - With THREAD(*CONCURRENT), a program compiled with DFTACTGRP(*YES) cannot return with LR on or return abnormally if the program is currently running in another thread. When the program tries to deactivate itself, the deactivation will fail with MCH4405.
  - With THREAD(*SERIALIZE), or THREAD(*CONCURRENT) where the SERIALIZE keyword is specified on a procedure specification, if the program runs in the default activation group, the RPG runtime cannot destroy the mutex used to serialize access to the module or the procedure. In some cases, after the RPG program has ended, the pointer in the module's static storage that points to the mutex will be deallocated or overwritten. This situation can cause a storage leak, because the system's storage associated with the mutex is not freed when the pointer to the mutex is lost. If the RPG program is called again, it will create a new mutex.

    This situation can arise in the following ways:

- The program is compiled with THREAD(*SERIALIZE) and DFTACTGRP(*YES), and the program ends with LR on, or it ends abnormally, which causes the program to be deactivated. When the program is deactivated, its static storage is deallocated.
- The RCLRSC command is used, and a program that uses a mutex to serialize a module or a procedure ran in the default activation group. A program will run in the default activation group if it is compiled with DFTACTGRP(*YES) or with ACTGRP(*CALLER), where the caller is in the default activation group.

• Avoid compiling with ACTGRP(*NEW) for programs that will run in a secondary thread. When a program compiled with ACTGRP(*NEW) ends in a secondary thread, the job will end with message CEE0200.

Storage that is Shared Among Multiple Threads

Two or more threads can access the same storage if any of the following are true:

• Variables are defined with the STATIC(*ALLTHREAD) keyword
• EXPORT/IMPORT keywords are used on the definition specifications in a thread-serialized module
• Data is based on a pointer where the pointer is available to more than one module
• Files are created or overridden with SHARE(*YES). In this case, it is the feedback areas that represent the shared storage. RPG always refers to the feedback areas during file operations, so you should synchronize access to the file itself.

It is up to the programmer to handle thread safety for storage that is shared across modules. This is done by adding logic in the application to synchronize access to the storage. To synchronize access to this shared storage, you can do one or both of the following:

• Structure the application such that the shared resources are not accessed simultaneously from multiple threads.
• If you are going to access resources simultaneously from separate threads, synchronize access using facilities such as semaphores or mutexes. For more information, see Using thread-related APIs.

How to Avoid Deadlock Between Modules

In some situations, it may be necessary for you to control the synchronization of modules using facilities other than a thread-serialized module or a serialized procedure. For example, consider the situation shown in Figure 75 on page 190, where two threads are each running a procedure in different thread-serialized modules: procedure PROC1 in module MOD1 and procedure PROC3 in module MOD2. MOD1 also has procedure PROC2 and MOD2 also has procedure PROC4. Even though there is no actual recursive calling; if PROC1 calls PROC4, it will wait for MOD2 to unlock; and if PROC3 calls PROC2, it will wait for MOD1 to unlock. The procedures will not be able to complete their calls, since each module will be locked by the thread in the other module. This type of problem can occur even with serialization of calls to a module, and is referred to as deadlock.

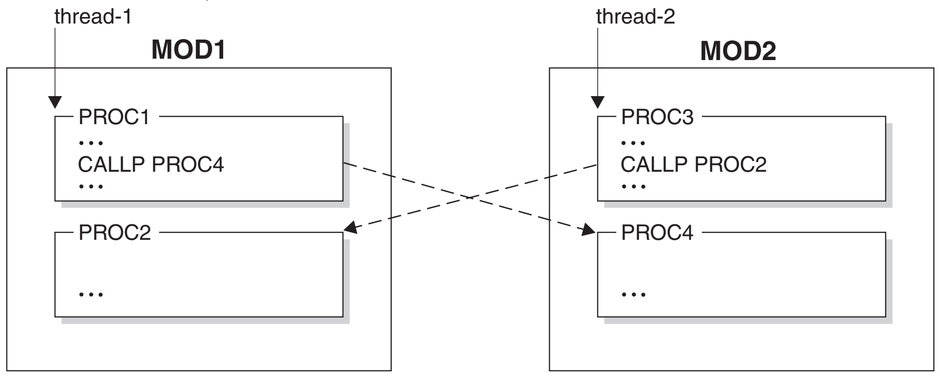


*Figure 75. Deadlock Example in a THREAD(*SERIALIZE) module*

This example shows how deadlock can occur if you try to access more than one procedure in the same thread-serialized module at the same time.

To avoid the problem in the above example and ensure thread safe applications, you can control the synchronization of modules using the techniques described in Using thread-related APIs. Any callers of PROC1 or PROC3 for each thread should do the following:

1. Restrict access to the modules for all threads except the current thread, always in the same order (MOD1 then MOD2)
2. In the current thread, call the required procedure (PROC1 or PROC3)
3. Relinquish access to the modules for all threads in the reverse order of step 1 (MOD2 then MOD1).

One thread would be successful in restricting access to MOD1. Since all users of MOD1 and MOD2 use the protocol of restricting access to MOD1 and MOD2 in that order, no other thread can call procedures in MOD1 or MOD2 while the first thread has restricted access to the modules. In this situation you have access to more than one procedure in the same module at the same time, but since it is only available to the current thread, it is thread safe.

This method should also be used to synchronize access to shared storage.

All-Thread Static Variables

When you define a variable with the STATIC(*ALLTHREAD) keyword, you are responsible for ensuring that the variable is used in a thread-safe way. Depending on the scope of the variable and usage of the variable, you may need to have additional variables to help synchonize access to the variables:

- If the variable is local to a serialized procedure, then only one thread can access the variable at one time due to the serialization, so you do not need to add any extra synchronization for it.
- If the variable is global to the module, and you can guarantee that it is changed in only one place in your code, and you can further guarantee that the code that changes the variable will run before any other thread can use the variable, then you do not need to add any synchronization for the variable.
- Otherwise, you must add an additional variable to be used with a synchronization technique such as a mutex or a semaphore. See information about *Threads* at: http://www.ibm.com/systems/infocenter/ and in Using thread-related APIs.

If you need to add a synchronization variable to synchronize access to another variable you must ensure the following:

- The synchronization variable must be initialized before the variable is ever accessed.
- Whenever you work with the variable, you must first gain access to it, by locking the semaphore or mutex; when you are finished working with the variable, you must unlock the semaphore or mutex.
- If the variable is exported from the module, you must ensure that all modules that import the variable can also use the the synchronization variable. You can do this by exporting the synchronization variable, or by adding exported lock and unlock procedures in your exporting module that can be called by any module that needs to use the variable.

  Tip: Establish a naming convention for your variables that require synchronization and for their synchronization variables or lock and unlock procedures. Your convention might be to prefix a variable that requires synchronization with SN_, and to use the same name for its synchronization variable or procedures, but with different prefixes. For example, variable SN_nextIndex might have lock and unlock procedures LOCK_nextIndex and UNLOCK_nextIndex. By using such a convention, and by rigidly enforcing its use, you can reduce the possibility that a programmer will use a variable that requires synchronization without observing the correct synchronization protocol.
- You must avoid deadlock situations. For example, if one thread has a lock for FLD1 and tries to obtain a lock for FLD2, while another thread has a lock on FLD2 and tries to obtained a lock on FLD1, then both threads will wait forever.

When to use a serialized procedure

You can use a serialized procedure to synchronize access to a shared resource. You may need to add additional manual synchronization if the shared resources are used elsewhere in your job.

In the following example, a global all-thread static variable is loaded from a file once, and all other uses in the application only refer to the value of the variable. Recall that it is necessary to control access to an allthread static variable if it might be changed by multiple threads at the same time, or if one thread might be changing the value while another thread is using the value. However, in the special case of a variable that is changed only once in "first-time-only setup" code, a serialized procedure is sufficient to control the access by multiple threads. All threads call the first-time-only setup procedure, and the procedure itself uses a local all-thread static variable to keep track of whether the setup has already been done. No manual synchronization is required to control the access to a local all-thread-static variable in a serialized procedure, because the procedure is serialized to allow only one thread to be running the procedure at one time

The getCustList procedure is an example of a first-time-only setup procedure; the shared resources that it is controlling are two global all-thread-static variables, ATS_custList and ATS_numCusts. The procedure is defined with the SERIALIZE keyword. It reads a file containing a list of customers and saves the list in an array. It uses a local all-thread static variable, ATS_done, to keep track of whether the list has already been obtained, and if it has been obtained already, it returns immediately. If more than one thread tries to call the procedure at the same time before the list has been obtained, one thread will get control and the other threads will wait until the first thread has completed the procedure. When the other threads finally get control, one at a time, they will return immediately because they will find that ATS_done has the value *ON.

```
 * !!! Warning !!!
 * These global ATS_xxx variables are in all-thread static storage.
 * They are setup in getCustList().
 * They should not be used before that procedure is called,
 * and they should not be changed after that procedure is called.
D ATS_custList    S            100A   VARYING DIM(500)
D                                     STATIC(*ALLTHREAD)
D ATS_numCusts    S             10I 0 INZ(0)
D                                     STATIC(*ALLTHREAD)
 /free
       // Ensure that the all-thread static variables ATS_custList
       // and ATS_numCusts have been set up
       getCustList();

       // Search for the customer name in the customer list
       if %lookup(custname : ATS_custList : 1 : ATS_numCusts);
         ...
 /end-free

P getCustList     B                   SERIALIZE
FcustList  IF   E             DISK
D custInfo        DS                  LIKEREC(custRec)
 * ATS_done is shared by all threads running this procedure.
 * It doesn't need special thread-control because the procedure
 * is serialized.
D ATS_done        S               N   INZ(*OFF)
D                                     STATIC(*ALLTHREAD)
 /free
      // Only load the customer array once
      if ATS_done;
         return;
      endif;
      // Fetch the list of customers into the ATS_custList array
      read custList custInfo;
      dow not %eof(custList);
         ATS_numCusts += 1;
         ATS_custList(ATS_numCusts) = %trim(custInfo.custName);

         read custList custInfo;
      enddo;

      // Set on the "first-time-only" indicator
      ATS_done = *ON;
 /end-free
P getCustList     E
```

When a serialized procedure does not provide sufficient protection

If you have a global all-thread static variable, it may seem like a good idea to control access to it by having serialized "get" and "set" procedures for the variable. Unfortunately, this does not give adequate protection, because the procedures are serialized independently, each having its own separate control mechanism. If one thread is running the "get" procedure, another could be running the "set" procedure at the same time.

If you want to use "get" and "set" procedures, you will need to add code to both procedures to manually synchronize access to the variable.

An alternative is to combine both "get" and "set" in one get-set procedure. It could have a separate parameter to indicate the required function, or it could have an optional parameter, which if passed, would provide the "set" function; the "get" function would always be provided since the procedure would always return a value.

However, even using a single "get-set" procedure may not provide adequate thread-safety for the variable. If you want to modify a variable using its previous value, such as adding one to the variable, you might think that getting the value of the variable, and then setting it to a new value in the same statement would work. However, another thread might call the procedure between your two calls to the procedure. Your second "set" call to the procedure would incorrectly overwrite the value that had been set by the other thread.

```
       // If myFld has the value 2 before this statement is run, the first call
       // would return 2.  The second call would set the value to 3.  If another
       // thread had set the value to 15 in between the calls, the second call
       // should logically set it to 16, not to 3.
       getSetMyFld                    // second call to getSetMyFld, to set the value
              (getSetMyFld() + 1);    // first call to getSetMyFld, to get the value
```

If you need to perform more than one access to a variable without another thread being able to get or set the variable while you are performing the operation, you must use some manual synchronization to control all access to the variable. All users of that variable must use the same synchronization mechanism.

Difficulty of manually synchronizing access to shared resources

It is very difficult to successfully control access to a shared resource. All users of the shared resource must agree to use the same synchronization mechanism. The wider the scope of visibility of the shared resource, the more difficult it is to control the access. For example, it is quite easy to have thread-safe access to a local all-thread static variable, since only that procedure can access it. It is very difficult to have thread-safe access to an exported all-thread static variable, since any procedure can access it.

Using thread-related APIs

You can call system APIs to start threads and wait for threads to complete, and to synchronize access to shared resources.

The following example creates several threads, and uses two different synchronization techniques to control access to some shared variables. To fully understand the examples, you should refer to the *Multithreaded Applications* at: http://www.ibm.com/systems/infocenter/. The examples are similar to the C examples that show how to use semaphores and mutexes.

The semaphore example shows how to pass a parameter to a thread-start procedure threadSem. Normally a thread-start procedure parameter is a data structure, whose subfields take the place of the parameters and return value that a normal procedure would use. The example has two subfields; the "val" subfield is input to the thread-start procedure, and the "result" subfield is output by the procedure. In the example, the thread-start procedure sets the result to the input value multiplied by two.

How to build the examples

1. Copy the source for the examples into a source member. The rest of these instructions assume that they are in file MYLIB/MYSRCFILE, in members THREADMTX and THREADSEM RPGLE.
2. Compile the programs.
   - CRTBNDRPG MYLIB/THREADMTX SRCFILE(MYLIB/MYSRCFILE)
   - CRTBNDRPG MYLIB/THREADSEM SRCFILE(MYLIB/MYSRCFILE)

How to run the examples

The sample programs must be run in a multithread-capable job. You can use SBMJOB to call the programs, specifying ALWMLTTHD(*YES) to allow multiple threads in the job:

- SBMJOB CMD(CALL MYLIB/THREADMTX) ALWMLTTHD(*YES)
- SBMJOB CMD(CALL MYLIB/THREADSEM) ALWMLTTHD(*YES)

```
 /UNDEFINE LOG_ALL_RESULTS

H THREAD(*CONCURRENT) MAIN(threadMtx)
H BNDDIR('QC2LE')
 /IF DEFINED(*CRTBNDRPG)
H DFTACTGRP(*NO)
 /ENDIF
H OPTION(*SRCSTMT : *NOUNREF)

 /COPY QSYSINC/QRPGLESRC,PTHREAD
D NUMTHREADS      C                   3

D threadMtx       PR                  EXTPGM('THREADMTX')

D mtxThread       PR              *   EXTPROC('mtxThread')
D    parm                         *   VALUE

D handleThreads   PR                  EXTPROC('handleThreads')

D checkResults    PR                  EXTPROC('checkResults')
D    string                   1000A   VARYING CONST
D    val                        10I 0 VALUE

D threadMsg       PR                  EXTPROC('threadMsg')
D    string                   1000A   VARYING CONST

D print           PR                  EXTPROC('print')
D    msg                      1000A   VARYING CONST

D CEETREC         PR
D    cel_rc_mod                 10I 0 OPTIONS(*OMIT)
D    user_rc                    10I 0 OPTIONS(*OMIT)

D sleep           PR                  EXTPROC(*CWIDEN:'sleep')
D    secs                       10I 0 VALUE

D fmtThreadId     PR            17A   VARYING
```

Figure 76. RPG source file THREADMTX

```
      *-----------------------------------------------------
      * Thread-scoped static variables (the STATIC keyword
      * is implied because the definition is global)
      *-----------------------------------------------------
     D psds            SDS
     D   pgmName                     10A   OVERLAY(psds : 334)

      *-----------------------------------------------------
      * Job-scoped static variables
      *-----------------------------------------------------

      * Shared data that will be protected by the mutex
     D sharedData      S             10I 0 INZ(0)
     D                                     STATIC(*ALLTHREAD)
     D sharedData2     S             10I 0 INZ(0)
     D                                     STATIC(*ALLTHREAD)

      * A mutex to control the shared data
     D mutex           DS                  LIKEDS(pthread_mutex_t)
     D                                     STATIC(*ALLTHREAD)

      // Program entry procedure
     P threadMtx       B
      /free
         print ('Enter ' + pgmName);
         handleThreads ();
         print ('Exit  ' + pgmName);
      /end-free
     P threadMtx       E


     P handleThreads   B
     D handleThreads   PI

     D thread          DS                  LIKEDS(pthread_t)
     D                                     DIM(NUMTHREADS)
     D rc              S             10I 0 INZ(0)
     D i               S             10I 0 INZ(0)
      /free

          print ('"handleThreads" starting');
          print ('Test using a mutex');

          // Initialize the mutex
          mutex = PTHREAD_MUTEX_INITIALIZER;

          print ('Hold Mutex to prevent access to shared data');
          rc = pthread_mutex_lock (mutex);

          checkResults('pthread_mutex_lock()' : rc);

          print ('Create/start threads');
          for i = 1 to NUMTHREADS;
             rc = pthread_create(thread(i) : *OMIT
                               : %paddr(mtxThread) : *NULL);
             checkResults ('pthread_create()' : rc);
          endfor;

          print ('Wait a bit until we are "done" with the shared data');
          sleep(3);
          print ('Unlock shared data');
          rc = pthread_mutex_unlock (mutex);
          checkResults('pthread_mutex_unlock()' : rc);

          print ('Wait for the threads to complete, '
               + 'and release their resources');
          for i = 1 to NUMTHREADS;
             rc = pthread_join (thread(i) : *OMIT);
             checkResults('pthread_join( ' + %char(i) + ')' : rc);
          endfor;
          print ('Clean up the mutex');
          rc = pthread_mutex_destroy (mutex);

          print ('"handleThreads" completed');
          return;

      /end-free
     P handleThreads   E

     P mtxThread       B
     D mtxThread       PI              *
     D    parm                         *   VALUE
     D rc              S             10I 0
     D
      /free

         threadMsg ('Entered');

         rc = pthread_mutex_lock (mutex);
         checkResults ('pthread_mutex_lock()' : rc);
         //********** Critical Section Begin *******************
         threadMsg ('Start critical section, holding lock');

         // Access to shared data goes here
         sharedData += 1;
         sharedData2 -= 1;

         threadMsg ('End critical section, release lock');
         //********** Critical Section End   *******************

         rc = pthread_mutex_unlock (mutex);
         checkResults ('pthread_mutex_unlock()' : rc);

         return *NULL;
      /end-free
     P mtxThread       E

     P checkResults    B                   EXPORT
     D checkResults    PI
     D    string                   1000A   VARYING CONST
     D    val                        10I 0 VALUE
     D msg             S           1000A   VARYING
      /FREE
         if val <> 0;
            print (string + ' failed with ' + %char(val));
            CEETREC (*OMIT : *OMIT);
         else;
            /if defined(LOG_ALL_RESULTS)
               print (string + ' completed normally with ' + %char(val));
            /endif
         endif;
      /END-FREE
     P checkResults    E
```

```
P threadMsg       B                   EXPORT
D threadMsg       PI
D    string                    1000A   VARYING CONST
 /FREE
    print ('Thread(' + fmtThreadId() + ') ' + string);
 /END-FREE
P threadMsg       E


P print           B                   EXPORT
D print           PI
D    msg                       1000A   VARYING CONST
D printf          PR               *   EXTPROC('printf')
D    template                      *   VALUE OPTIONS(*STRING)
D    string                        *   VALUE OPTIONS(*STRING)
D    dummy                         *   VALUE OPTIONS(*NOPASS)
D NEWLINE         C                   x'15'
 /free
    printf ('%s' + NEWLINE : msg);
 /end-free
P print           E

P fmtThreadId     B                   EXPORT
D fmtThreadId     PI             17A   VARYING
D pthreadId       DS                  LIKEDS(pthread_id_np_t)
D buf             S            1000A
D sprintf         PR               *   EXTPROC('sprintf')
D  buf                             *   VALUE
D  template                        *   VALUE OPTIONS(*STRING)
D  num1                          10U 0 VALUE
D  num2                          10U 0 VALUE
D  dummy                           *   OPTIONS(*NOPASS)
 /FREE
    pthreadId = pthread_getthreadid_np();
    // get the hex form of the 2 parts of the thread-id
    // in "buf", null-terminated
    sprintf (%addr(buf)
           : '%.8x %.8x'
           : pthreadId.intId.hi
           : pthreadId.intId.lo);
    return %str(%addr(buf));
 /END-FREE
P fmtThreadId     E
```

```
 /UNDEFINE LOG_ALL_RESULTS

H THREAD(*CONCURRENT) MAIN(threadSem)
H BNDDIR('QC2LE')
 /IF DEFINED(*CRTBNDRPG)
H DFTACTGRP(*NO)
 /ENDIF
H OPTION(*SRCSTMT : *NOUNREF)

 /COPY QSYSINC/QRPGLESRC,PTHREAD
 /COPY QSYSINC/QRPGLESRC,SYSSEM
 /COPY QSYSINC/QRPGLESRC,SYSSTAT
D NUMTHREADS      C                   3

D threadSem       PR                  EXTPGM('THREADSEM')

D semThreadParm_t...
D                 DS                  QUALIFIED TEMPLATE
D   val                         10I 0
D   result                      10I 0
D semThread       PR              *   EXTPROC('semThread')
D    parm                             LIKEDS(semThreadParm_t)

D handleThreads   PR                  EXTPROC('handleThreads')

D checkResults    PR                  EXTPROC('checkResults')
D    string                   1000A   VARYING CONST
D    val                        10I 0 VALUE

D checkResultsErrno...
D                 PR                  EXTPROC('checkResultsErrno')
D    string                   1000A   VARYING CONST
D    cond                         N   VALUE

D threadMsg       PR                  EXTPROC('threadMsg')
D    string                   1000A   VARYING CONST

D print           PR                  EXTPROC('print')
D    msg                      1000A   VARYING CONST
```

Figure 77. RPG program THREADSEM showing the use of a semaphore

```
D CEETREC         PR
D    cel_rc_mod                 10I 0 OPTIONS(*OMIT)
D    user_rc                    10I 0 OPTIONS(*OMIT)

D sleep           PR                  EXTPROC(*CWIDEN:'sleep')
D    secs                       10I 0 VALUE

D fmtThreadId     PR            17A   VARYING

 *-------------------------------------------------------
 * Thread-scoped static variables (the STATIC keyword
 * is implied because the definition is global)
 *-------------------------------------------------------
D psds           SDS
D   pgmName                     10A   OVERLAY(psds : 334)

 *-------------------------------------------------------
 * Job-scoped static variables
 *-------------------------------------------------------

 * Shared data that will be protected by the mutex
D sharedData      S             10I 0 INZ(0)
D                                     STATIC(*ALLTHREAD)
D sharedData2     S             10I 0 INZ(0)
D                                     STATIC(*ALLTHREAD)

 * A semaphore to control the shared data
D semaphoreId     S             10I 0 STATIC(*ALLTHREAD)

 * Simple lock operation. 0=which-semaphore, -1=decrement, 0=noflags
 * Will be set to { 0, -1, 0} in main procedure before threads are created
D lockOperation   DS                  LIKEDS(struct_sembuf)
D                                     DIM(1)
D                                     STATIC(*ALLTHREAD)
```

```
      * Simple unlock operation. 0=which-semaphore, 1=increment, 0=noflags
      * Will be set to { 0, 1, 0} in main procedure before threads are created
     D unlockOperation...
     D                 DS                  LIKEDS(struct_sembuf)
     D                                     DIM(1)
     D                                     STATIC(*ALLTHREAD)

      // Program entry procedure
     P threadSem       B
      /free
         print ('Enter ' + pgmName);
         handleThreads ();
         print ('Exit  ' + pgmName);
      /end-free
     P threadSem       E


     P handleThreads   B
     D handleThreads   PI

     D thread          DS                  LIKEDS(pthread_t)
     D                                     DIM(NUMTHREADS)
     D rc              S             10I 0 INZ(0)
     D i               S             10I 0 INZ(0)
     D parms           DS                  LIKEDS(semThreadParm_t)
     D                                     DIM(NUMTHREADS)
      /free

          print ('"handleThreads" starting');

          print ('Test using a semaphore');

          lockOperation(1).sem_num = 0;
          lockOperation(1).sem_op = -1;
          lockOperation(1).sem_flg = 0;

          unlockOperation(1).sem_num = 0;
          unlockOperation(1).sem_op = 1;
          unlockOperation(1).sem_flg = 0;
```

```
           // Create a private semaphore set with 1
           // semaphore that only I can use
           semaphoreId = semget(IPC_PRIVATE : 1 : 0 + S_IRUSR + S_IWUSR);
           checkResultsErrno ('semget' : semaphoreId >= 0);

           // Set the semaphore count to 1.
           // Simulate a mutex
           rc = semctl(semaphoreId : 0 : CMD_SETVAL : 1);
           checkResults('semctl(SETVAL)' : rc);

           print ('Wait on semaphore to prevent access to shared data');
           rc = semop(semaphoreId : lockOperation(1) : 1);
           checkResultsErrno('main semop(lock)': rc = 0);

           parms(1).val = 5;
           parms(2).val = -10;
           parms(3).val = 421;

           print ('Create/start threads');
           for i = 1 to NUMTHREADS;
              rc = pthread_create(thread(i) : *OMIT
                                  : %paddr(semThread) : %addr(parms(i)));
              checkResults ('pthread_create()' : rc);
           endfor;

           print ('Wait a bit until we are "done" with the shared data');
           sleep (3);
           print ('Unlock shared data');
           rc = semop (semaphoreId : unlockOperation(1) : 1);
           checkResultsErrno ('main semop(unlock)' : rc = 0);

           print ('Wait for the threads to complete, '
                + 'and release their resources');
           for i = 1 to NUMTHREADS;
              rc = pthread_join (thread(i) : *OMIT);
              checkResults('pthread_join( ' + %char(i) + ')' : rc);
           endfor;
           print ('Clean up the semaphore');
           rc = semctl(semaphoreId : 0 : IPC_RMID);
           checkResults ('semctl(removeID)' : rc);
```

```
          print ('Result(1) = ' + %char(parms(1).result));
          print ('Result(2) = ' + %char(parms(2).result));
          print ('Result(3) = ' + %char(parms(3).result));

          print ('"handleThreads" completed');
          return;

      /end-free
     P handleThreads   E

     P semThread       B
     D semThread       PI              *
     D    parm                             LIKEDS(semThreadParm_t)

     D rc              S             10I 0
     D
      /free

         threadMsg ('Entered + parm.val = ' + %char(parm.val));
         // Set the output subfields of the parameter
         parm.result = parm.val * 2;

         rc = semop (semaphoreId : lockOperation(1) : 1);
         checkResultsErrno ('thread semop(lock)' : rc = 0);

         //********** Critical Section Begin *******************
         threadMsg ('Start critical section, holding semaphore');

         // Access to shared data goes here
         sharedData += 1;
         sharedData2 -= 1;

         threadMsg ('End critical section, release semaphore');
         //********** Critical Section End   *******************

         rc = semop (semaphoreId : unlockOperation(1) : 1);
         checkResultsErrno ('thread semop(unlock)' : rc = 0);

         threadMsg ('Exiting');

         return *NULL;
      /end-free
     P semThread       E

     P checkResults    B                   EXPORT
     D checkResults    PI
     D    string                   1000A   VARYING CONST
     D    val                        10I 0 VALUE
     D msg             S           1000A   VARYING
      /FREE
         if val <> 0;
            print (string + ' failed with ' + %char(val));
            CEETREC (*OMIT : *OMIT);
         else;
            /if defined(LOG_ALL_RESULTS)
               print (string + ' completed normally with ' + %char(val));
            /endif
         endif;
      /END-FREE
     P checkResults    E

     P checkResultsErrno...
     P                 B
     D checkResultsErrno...
     D                 PI
     D    string                   1000A   VARYING CONST
     D    cond                        N    VALUE
     D getErrnoPtr     PR             *    EXTPROC('__errno')
     D errnoVal        S             10I 0 based(threadErrnoPtr)
      /FREE
         if not cond;
            threadErrnoPtr = getErrnoPtr();
            print (string + ' Errno(' + %char(errnoVal) + ')');
            CEETREC (*OMIT : *OMIT);
         else;
            /if defined(LOG_ALL_RESULTS)
               print (string + ' completed normally');
            /endif
         endif;
      /END-FREE
     P checkResultsErrno...
     P                 E
     P threadMsg       B                    EXPORT
     D threadMsg       PI
     D    string                   1000A    VARYING CONST
      /FREE
         print ('Thread(' + fmtThreadId() + ') ' + string);
      /END-FREE
     P threadMsg       E
     P print           B                    EXPORT
     D print           PI
     D    msg                      1000A    VARYING CONST
     D printf          PR                 *  EXTPROC('printf')
     D    template                        *  VALUE OPTIONS(*STRING)
     D    string                          *  VALUE OPTIONS(*STRING)
     D    dummy                           *  VALUE OPTIONS(*NOPASS)
     D NEWLINE         C                    x'15'
      /free
         printf ('%s' + NEWLINE : msg);
      /end-free
     P print           E

     P fmtThreadId     B                    EXPORT
     D fmtThreadId     PI            17A    VARYING
     D pthreadId       DS                   LIKEDS(pthread_id_np_t)
     D buf             S           1000A
     D sprintf         PR                 *  EXTPROC('sprintf')
     D  buf                               *  VALUE
     D  template                          *  VALUE OPTIONS(*STRING)
     D  num1                         10U 0 VALUE
     D  num2                         10U 0 VALUE
     D  dummy                             *  OPTIONS(*NOPASS)
      /FREE
         pthreadId = pthread_getthreadid_np();
         // get the hex form of the 2 parts of the thread-id
         // in "buf", null-terminated
         sprintf (%addr(buf)
                : '%.8x %.8x'
                : pthreadId.intId.hi
                : pthreadId.intId.lo);
         return %str(%addr(buf));
      /END-FREE
     P fmtThreadId     E
```

RPG and the eBusiness World

This chapter describes how you can use ILE RPG as part of an eBusiness solution. It includes:

- "RPG and XML" on page 204
- "RPG and MQSeries" on page 210
- "RPG and Java" on page 211

RPG and XML

The Extensible Markup Language (XML) is a subset of SGML that is developed by the World Wide Web Consortium (W3C). Its goal is to enable generic SGML to be served, received, and processed on the Web in the way that is now possible with HTML. XML has been designed for ease of implementation and for interoperability with both SGML and HTML.

For more information about XML, see `http://www.w3.org/XML`

You can use the XML-INTO and XML-SAX operation codes to process your XML documents. For more information, see "Processing XML Documents" on page 204.

XML Toolkit (5733-XT1) allows your ILE RPG programs to create new XML documents and parse existing ones. You can use XML as both a datastore and I/O mechanism.

Processing XML Documents

You can process XML documents from your RPG program by using the XML-INTO or XML-SAX statements. These statements are the RPG language interface to the high-speed XML parser. The parser currently being used by RPG is a non-validating parser, although it checks XML documents for many wellformedness errors. See the "XML Conformance" section in the "XML Reference Material" appendix of the *ILE COBOL Programmer's Guide* for more information on the XML parser.

The XML documents can be in a character or UCS-2 RPG variable, or they can be in an Integrated File System file.

The parser is a SAX parser. A SAX parser operates by reading the XML document character by character. Whenever it has located a fragment of the XML document, such as an element name, or an attribute value, it calls back to a handling procedure provided by the caller of the parser, passing it information about the fragment of XML that it has found. For example, when the parser has found an XML element name, it calls the handling procedure indicating that the "event" is a "start element" event and passing it the name of the element.

The handling procedure processes the information and returns to the parser which continues to read the XML document until it has enough information to call the handling procedure with another event. This process repeats until the entire XML document has been parsed, or until the handling procedure indicates that parsing should end.

For example, consider the following XML document:

```
<email type="text">
  <sendto>JohnDoe@there</sendto>
</email>
```

The following are the fragments of text that the parser would read, the events that it would generate, and the data associated with each event. **Note:** The term "whitespace" refers to end-of-line characters, tab characters and blanks.

Parsed text	Event	Event data
	start document	
<email	start element	"email"
type=	attribute name	"type"
"text"	attribute value	"text"
>whitespace	element content	the whitespace
<sendto>	start element	"sendto"
JohnDoe@there	element content	"JohnDoe@there"
</sendto>	end element	"sendto"
whitespace	element content	the whitespace
</email>	end element	"email"
	end document	

The XML-SAX and XML-INTO operation codes allow you to use the XML parser.

1. The XML-SAX operation allows you to specify an event handling procedure to handle every event that the parser generates. This is useful if you do not know in advance what an XML document may contain.

   For example, if you know that an XML document will contain an XML attribute with the name *type*, and you want to know the value of this attribute, your handling procedure can wait for the "attribute name" event to have a value of "type". Then the next time the handler is called, it should be an "attribute value" event, with the required data ("text" in the example above).

2. The XML-INTO operation allows you to read the contents of an XML document directly into an RPG variable. This is useful if you know the format of the XML document and you know that the names of the XML elements in the document will be the same as the names you have given to your RPG variables.

For example, if you know that the XML document will always have the form of the document above, you can define an RPG data structure with the name "email", and with subfields "type" and "sendto". Then you can use the XML-INTO operation to read the XML document directly into the data structure. When the operation is complete, the "type" subfield would have the value "text" and the "sendto" subfield would have the value "JohnDoe@there".

3. The XML-INTO operation also allows you to obtain the values of an unknown number of repeated XML elements. You provide a handling procedure that receives the values of a fixed number of elements each time the handling procedure is called. This is useful if you know that the XML document will contain a series of identical XML elements, but you don't know in advance how many there will be.

The XML data is always returned by the parser in text form. If the data is known to represent other data types such as numeric data, or date data, the XML-SAX handling procedure must use conversion functions such as %INT or %DATE to convert the data.

The XML-INTO operation will automatically convert the character data to the type of the field or subfield specified as the receiver.

Both the XML-SAX and XML-INTO operations allow you to specify a series of options that control the operation. The options are specified in a single character expression in the form

```
'opt1=val1 opt2=val2'
```

Each operation has its own set of valid options. The options that are common to both operation codes are

doc
The "doc" option specifies whether the XML document that you provide to the operation is the name of an Integrated File System file containing the document, or the document itself. The default is "doc=string" indicating that you have provided an actual XML document. You use the option "doc=file" to indicate that you have provided the name of a file containing the actual XML document.

ccsid
The "ccsid" option specifies the CCSID in which the XML parser will return data. For the XML-SAX operation, you can specify any CCSID that the parser supports. For the XML-INTO operation, you can only control whether the parsing will be done in single-byte character or UCS-2. See the information in the *ILE RPG Reference* for more information on the "ccsid" option for each of these operation.

XML Parser Error Codes

If the XML parser detects an error in the XML document during parsing, message RNX0351 will be issued. From the message, you can get the specific error code associated with the error, as well as the offset in the document where the error was discovered.

The following table shows the meaning of each parser error code:

XML Parser Error Code	Description
1	The parser found an invalid character while scanning white space outside element content.
2	The parser found an invalid start of a processing instruction, element, comment, or document type declaration outside element content.
3	The parser found a duplicate attribute name.
4	The parser found the markup character '<' in an attribute value.
5	The start and end tag names of an element did not match.
6	The parser found an invalid character in element content.
7	The parser found an invalid start of an element, comment, processing instruction, or CDATA section in element content.

XML Parser Error Code	Description
8	The parser found in element content the CDATA closing character sequence ']]>' without the matching opening character sequence '<![CDATA['.
9	The parser found an invalid character in a comment.
10	The parser found in a comment the character sequence '--' (two hyphens) not followed by '>'.
11	The parser found an invalid character in a processing instruction data segment.
12	A processing instruction target name was 'xml' in lowercase, uppercase or mixed case.
13	The parser found an invalid digit in a hexadecimal character reference (of the form &#xdddd;, for example &#x0eb1).
14	The parser found an invalid digit in a decimal character reference (of the form &#dddd;).
15	A character reference did not refer to a legal XML character.
16	The parser found an invalid character in an entity reference name.
17	The parser found an invalid character in an attribute value.
18	The parser found a possible invalid start of a document type declaration.
19	The parser found a second document type declaration.
20	An element name was not specified correctly. The first character was not a letter, '_', or ':', or the parser found an invalid character either in or following the element name.
21	An attribute was not specified correctly. The first character of the attribute name was not a letter, '_', or ':', or a character other than '=' was found following the attribute name, or one of the delimiters of the value was not correct, or an invalid character was found in or following the name.
22	An empty element tag was not terminated by a '>' following the '/'.
23	The element end tag was not specified correctly. The first character was not a letter, '_', or ':', or the tag was not terminated by '>'.
24	The parser found an invalid start of a comment or CDATA section in element content.
25	A processing instruction target name was not specified correctly. The first character of the processing instruction target name was not a letter, '_', or ':', or the parser found an invalid character in or following the processing instruction target name.
26	A processing instruction was not terminated by the closing character sequence '?>'.
27	The parser found an invalid character following '&' in a character reference or entity reference.
28	The version information was not present in the XML declaration.
29	The 'version' in the XML declaration was not specified correctly. 'version' was not followed by '=', or the value was missing or improperly delimited, or the value specified a bad character, or the start and end delimiters did not match, or the parser found an invalid character following the version information value closing delimiter in the XML declaration.
30	The parser found an invalid attribute instead of the optional encoding declaration in the XML declaration.

XML Parser Error Code	Description
31	The encoding declaration value in the XML declaration was missing or incorrect. The value did not begin with lowercase or uppercase A through Z, or 'encoding' was not followed by '=', or the value was missing or improperly delimited or it specified a bad character, or the start and end delimiters did not match, or the parser found an invalid character following the closing delimiter.
32	The parser found an invalid attribute instead of the optional standalone declaration in the XML declaration.
33	The 'standalone' attribute in the XML declaration was not specified correctly. 'standalone' was not followed by a '=', or the value was either missing or improperly delimited, or the value was neither 'yes' nor 'no', or the value specified a bad character, or the start and end delimiters did not match, or the parser found an invalid character following the closing delimiter.
34	The XML declaration was not terminated by the proper character sequence '?>', or contained an invalid attribute.
35	The parser found the start of a document type declaration after the end of the root element.
36	The parser found the start of an element after the end of the root element.
300	The parser reached the end of the document before the document was complete.
301	The %HANDLER procedure for XML-INTO or XML-SAX returned a non-zero value, causing the XML parsing to end.
302	The parser does not support the requested CCSID value or the first character of the XML document was not '<'.
303	The document was too large for the parser to handle. The parser attempted to parse the incomplete document, but the data at the end of the document was necessary for the parsing to complete.
500-999	Internal error in the external parser. Please report the error to your service representative.
10001-19999	Internal error in the parser. Please report the error to your service representative.

Limitations of the XML Parser

- An RPG character variable can only be 16773104 bytes long. If your program has a pointer to XML data that is longer than that, for example from an MQSeries®® call, you will have to write the XML data to a temporary file in the Integrated File System, and parse the XML data from your temporary file. See Figure 78 on page 209 for a sample procedure that does this.
- If the parsing is done in a single-byte character CCSID, the maximum number of characters that the parser can handle is 2147483408.
- If the parsing is done in UCS-2, the maximum number of UCS-2 characters that the parser can handle is 1073741704.
- The parser does not support every CCSID. If your job CCSID is one of the CCSIDs that the parser does not handle, you must parse your document in UCS-2.
  - The following EBCDIC CCSIDs are supported: 1047, 37, 1140, 273, 1141, 277, 1142, 278, 1143, 280, 1144, 284, 1145, 285, 1146, 297, 1147, 500, 1148, 871, and 1149.
  - The following ASCII CCSIDs are supported: 819, 813, 920.
  - The following Unicode CCSIDs are supported: 1200, 13488, 17584.

- The parser does not support entity references other than the five predefined references &amp;, &apos, &gt, &lt, and &quot. When it encounters an unknown entity reference, it generates either an "unknown reference" or "unknown attribute reference" event. The value of the event is the reference in the form "&name;".
- The parser does not parse the DOCTYPE declaration. The text of the DOCTYPE declaration is passed as the data value for the "DOCTYPE declaration" event.
- The parser does not support name spaces. It ignores the colons in XML element and attribute names.
- The parser does not generate "start prefix mapping" and "end prefix mapping" events. It ignores the colons in XML element and attribute names.

```
 * Parameters:
 * 1. path      : a pointer to a null-terminated string containing
 *                the path to the file to be written
 * 2. dataPtr   : a pointer to the data to be written
 * 3. dataLen   : the length of the data in bytes
 * 4. dataCcsid : the CCSID of the data
 * 5. fileCcsid : the desired CCSID of the file
 * Sample RPG coding:
 *   ifsWrite ('/home/mydir/temp.xml' : xmlPtr : xmlLen : 37 : 37);
 *   xml-into ds %xml('/home/mydir/temp.xml' : 'doc=file');
 * To delete the file, use the system command
 *   rmvlnk '/home/mydir/temp.xml'
```

Figure 78. Writing data to an Integrated File System file

```
 * Note: This module requires BNDDIR(QC2LE)
 P ifsWrite        B                   EXPORT
D ifsWrite        PI
D  path                           *   VALUE OPTIONS(*STRING)
D  dataPtr                        *   VALUE
D  dataLen                      10I 0 VALUE
D  dataCcsid                    10I 0 VALUE
D  fileCcsid                    10I 0 VALUE

D O_CREAT         C                   x'00000008'
D O_TRUNC         C                   x'00000040'
D O_WRONLY        C                   x'00000002'
D O_RDWR          C                   x'00000004'
D O_CCSID         C                   x'00000020'
D O_TEXT_CREAT    C                   x'02000000'
D O_TEXTDATA      C                   x'01000000'
D O_SHARE_NONE    C                   x'00080000'

D S_IRUSR         C                   x'0100'
D S_IROTH         C                   x'0004'
D S_IRGRP         C                   x'0020'
D S_IWUSR         C                   x'0080'
D S_IWOTH         C                   x'0002'
```

```
D ssize_t         S             10I 0
D size_t          S             10U 0

D open            PR            10I 0 EXTPROC('open')
D   path                          *   VALUE OPTIONS(*STRING)
D   flag                        10I 0 VALUE
D   mode                        10I 0 VALUE
D   fileCcsid                   10I 0 VALUE options(*nopass)
D   dataCcsid                   10I 0 VALUE options(*nopass)
D writeFile       PR                  LIKE(ssize_t)
D                                     EXTPROC('write')
D   handle                      10I 0 VALUE
D   data                          *   VALUE
D   len                               VALUE LIKE(size_t)
D closeFile       PR            10I 0 EXTPROC('close')
D   handle                      10I 0 VALUE

D oflag           S             10I 0
D omode           S             10I 0
D handle          S             10I 0
D rc              S             10I 0

D sysErrno        PR              *   EXTPROC('__errno')
D errno           S             10I 0 BASED(pErrno)
 /FREE
   pErrno = sysErrno();
   oflag = 0 + O_WRONLY + O_CREAT + O_TEXT_CREAT + O_TRUNC
         + O_CCSID + O_TEXTDATA + O_SHARE_NONE;
   omode = 0 + S_IRUSR + S_IWUSR + S_IRGRP + S_IROTH;

   handle = open(path : oflag : omode : fileCcsid : dataCcsid);
```

```
// insert error handling if handle is less than zero

   rc = writeFile (handle : dataPtr : dataLen);
   // insert error handling if rc is not zero

   rc = closeFile (handle);
   // insert error handling if rc is not zero

 /END-FREE
P ifswrite        E
```

RPG and other data formats such as JSON

Use the DATA-INTO operation code to import data from any hierarchical data format, such as JSON, or CSV.

For more information, see the section on the DATA-INTO operation in the *Rational Development Studio for i: ILE RPG Reference*.

When message RNX0357 is issued for a DATA-INTO operation, the message mentions an error code. The meaning for each error code is determined by each parser.

RPG and MQSeries

With MQSeries, a program can communicate with other programs on the same platform or a different platform using the same messaging product. MQSeries manages network interfaces, assures delivery, deals with communications protocols, and handles recovery after system problems. MQSeries is available on over 35 platforms.

RPG and Java

Introduction to Java and RPG

The Java programming language is a high-level object-oriented language developed by Sun Microsystems.

In object-oriented programming, a "method" is a programmed procedure that is defined as part of a "class", which is a collection of methods and variables. Java methods can be called from your RPG program. While most Java methods are written in Java, a method can also be written in another high-level language, such as RPG. This is known as a "native method". This section includes information on calling Java methods from RPG and on writing RPG native methods.

The Object Data Type and CLASS Keyword

Fields that can store objects are declared using the **O** data type. To declare a field of type O, code O in column 40 of the D-specification and use the CLASS keyword to provide the class of the object. The CLASS keyword accepts two parameters:

```
CLASS(*JAVA:class_name)
```

*JAVA identifies the object as a Java object. Class_name specifies the class of the object. It must be a character literal or named constant, and the class name must be fully qualified. The class name is case sensitive.

For example, to declare a field that will hold an object of type BigDecimal:

```
D bdnum           S               O   CLASS(*JAVA:'java.math.BigDecimal')
```

To declare a field that will hold an object of type String:

```
D string          S               O   CLASS(*JAVA:'java.lang.String')
```

Note that both class names are fully qualified and that their case exactly matches that of the Java class.

Fields of type O cannot be defined as subfields of data structures. It is possible to have arrays of type O fields, but pre-runtime and compile-time tables and arrays of type O are not allowed.

Prototyping Java Methods

Like subprocedures, Java methods must be prototyped in order to call them correctly. The ILE RPG compiler must know the name of the method, the class it belongs to, the data types of the parameters and the data type of the returned value (if any), and whether or not the method is a static method.

The extended EXTPROC keyword can be used to specify the name of the method and the class it belongs to. When prototyping a Java method, the expected format of the EXTPROC keyword is:

```
EXTPROC(*JAVA:class_name:method_name)
```

Both the class name and the method name must be character constants. The class name must be a fully qualified Java class name and is case sensitive. The method name must be the name of the method to be called, and is case sensitive.

Use *JAVA when creating a prototype for either a method written in Java or a native method written in RPG. Use the STATIC keyword to indicate that a method is static.

Java and RPG Definitions and Data Types

The data types of the parameters and the returned value of the method are specified in the same way as they are when prototyping a subprocedure, but the data types actually map to Java data types. The following table shows the mappings of ILE RPG data types to and from Java data types.

If you copy the JNI member in QSYSINC/QRPGLESRC, you can use LIKE to define your RPG variables and parameters like definitions in that file. For example, to define a variable like the Java "int" type, define it LIKE(jint). In the remainder of the discussion about RPG and Java, any definitions defined with

LIKE(jxxxx) are assumed to have a /COPY for QSYSINC/QRPGLESRC,JNI in the module. See the section "Additional RPG Coding for Using Java" on page 225 for more information about using this /COPY file.

Table 65. RPG definitions for Java data types

Java Data Type	ILE RPG Data Type	RPG Definitions
boolean	indicator	N
byte [1]	integer	3I 0
	character	1A
byte[]	character length > 1 (See "3" on page 212.)	nA
	array of character length=1 (See "4" on page 212.)	1A DIM(x)
	date	D
	time	T
	timestamp	Z
short	2–byte integer	5I 0
char	UCS-2 length=1	1C
char[]	UCS-2 length>1 (See "3" on page 212.)	nC
	array of UCS-2 length=1 (See "4" on page 212.)	1C DIM(x)
int	4–byte integer	10I 0
long	8–byte integer	20I 0
float	4–byte float	4F
double	8–byte float	8F
any object	object	O CLASS(x)
any array	array of equivalent type (See "4" on page 212.)	DIM(x)

Note:

1. When a Java byte type is converted to or from a character (1A) data type, ASCII conversion occurs. When a Java byte type is converted to or from an integer (3I) data type, ASCII conversion does not occur.
2. For arrays of any type in Java, you can declare an array of the equivalent type in RPG. However, note that you cannot use an array of character length greater than 1 or UCS-2 length greater than 1 data types.
3. For UCS-2 length greater than 1 and character length greater than 1 data types, the VARYING keyword is allowed. In general, it's recommended to use the VARYING keyword, since Java byte[] and char[] cannot be declared with a fixed length.
4. For RPG array data types, OPTIONS(*VARSIZE) should normally be coded for array parameters, since Java arrays cannot be declared with a fixed length.

Zoned, Packed, Binary, and Unsigned data types are not available in Java. If you pass a Zoned, Packed, Binary, or Unsigned field as a parameter, the compiler will do the appropriate conversion, but this may result in truncation and/or loss of precision.

When calling a method, the compiler will accept arrays as parameters only if the parameter is prototyped using the DIM keyword.

If the return value or a parameter of a method is an object, you must provide the class of the object by coding the CLASS keyword on the prototype. The class name specified will be that of the object being returned or the parameter being passed. (Use the EXTPROC keyword to specify the class of the method being called.)

If the method being called is a static method, then you must specify the STATIC keyword on the prototype. If the method is a constructor, you must specify *CONSTRUCTOR as the name of the method.

In Java, the following data types can only be passed by value:

```
boolean
byte
int
short
long
float
double
```

Parameters of these types must have the VALUE keyword specified for them on the prototype.

Note that objects can only be passed by reference. The VALUE keyword cannot be specified with type O. Since arrays are seen by Java as objects, parameters mapping to arrays must also be passed by reference. This includes character and byte arrays. The CONST keyword can be used.

Examples of Prototyping Java Methods

This section presents some examples of prototyping Java methods.

Example 1

The Java Integer class contains a static method called *toString*, which accepts an *int* parameter, and returns a String object. It is declared in Java as follows:

```
static String  Integer.toString(int)
```

This method would be prototyped as follows:

```
D tostring        PR              O   EXTPROC(*JAVA:
D                                      'java.lang.Integer':
D                                      'toString')
D                                     CLASS(*JAVA:'java.lang.String')
D                                     STATIC
D    num                        10I 0 VALUE
```

The EXTPROC keyword identifies the method as a Java method. It also indicates that the method name is 'toString', and that it is found in class 'java.lang.Integer'.

The O in column 40 and the CLASS keyword tell the compiler that the method returns an object, and the class of that object is 'java.lang.String'.

The STATIC keyword indicates that the method is a static method, meaning that an Integer object is not required to call the method.

The data type of the parameter is specified as 10I, which maps to the Java *int* data type. Because the parameter is an int, it must be passed by value, and the VALUE keyword is required.

Example 2

The Java Integer class contains a static method called *getInteger*, which accepts String and Integer objects as parameters, and returns an Integer object. It is declared in Java as follows:

```
static Integer Integer.getInteger(String, Integer)
```

This method would be prototyped as follows:

```
D getint          PR              O   EXTPROC(*JAVA:
D                                      'java.lang.Integer':
D                                      'getInteger')
D                                     CLASS(*JAVA:'java.lang.Integer')
```

```
D                                        STATIC
D   string                          O    CLASS(*JAVA:'java.lang.String') CONST
D   num                             O    CLASS(*JAVA:'java.lang.Integer') CONST
```

This method accepts two objects as parameters. O is coded in column 40 of the D-specification and the CLASS keyword specifies the class of each object parameter. Because both parameters are input-only, the CONST keyword is specified.

Example 3

The Java Integer class contains a method called *shortValue*, which returns the short representation of the Integer object used to invoke the method. It is declared in Java as follows:

```
short shortValue()
```

This method would be prototyped as follows:

```
D shortval        PR             5I 0 EXTPROC(*JAVA:
D                                      'java.lang.Integer':
D                                      'shortValue'
```

The STATIC keyword is not specified because the method is not a static method. The method takes no parameters, so none are coded. When you call this method, you will specify the Integer instance as the first parameter. The returned value is specified as 5I, which maps to the Java short data type.

Example 4

The Java Integer class contains a method called *equals*, which accepts an Object as parameter and returns a boolean. It is declared in Java as follows:

```
boolean equals(Object)
```

This method would be prototyped as follows:

```
D equals          PR              N   EXTPROC(*JAVA:
D                                      'java.lang.Integer':
D                                      'equals')
D  obj                            O   CLASS(*JAVA:'java.lang.Object')
```

The returned value is specified as N, which maps to the Java boolean data type. Because this is not a static method, a call to this method will have two parameters with the instance parameter coded first.

Calling Java Methods from ILE RPG

This section describes how to call Java methods from ILE RPG programs.

If the method is not a static method, then it is called an "instance method" and an object instance must be coded as an extra first parameter in order to call the method. For example, if an instance method is prototyped with one parameter, you must call it with two parameters, the first being the instance parameter.

The following steps describe the call from ILE RPG to a Java method:

1. Java methods can be called using existing operation codes CALLP (when no return value is expected) and EVAL (when a return value is expected). When your RPG procedure attempts to make call to a Java method, RPG will check to see if the Java Virtual Machine (JVM) has been started. If not, RPG will start the JVM for you. It is also possible to start JVM yourself using the JNI function described in "Creating the Java Virtual Machine (JVM)" on page 228
2. If you are using your own classes (or any classes outside the normal java.xxx classes), be sure to have your CLASSPATH environment variable setup before you call any Java methods. When RPG starts up the JVM for you, it will add the classes in your CLASSPATH environment variable to the standard

classpath, so when you use your own classes, Java will be able to find them. Set the CLASSPATH environment variable interactively like this:

```
===>ADDENVVAR ENVVAR(CLASSPATH)
              VALUE('/myclasses/:/xyzJava/classes/')
```

The directories must be separated by colons.

3. Normally, Java does its own garbage collection, detecting when an object is no longer needed. When you create objects by calling Java constructors from your non-native RPG procedure, Java has no way of knowing that the object can be destroyed, so it never destroys them. You can enable garbage collection for several objects at once by calling the JNI functions described in "Telling Java to free several objects at once" on page 225. If you know you are not going to need an object any more, you should tell this to Java by calling the JNI function described in "Telling Java you are finished with a temporary object" on page 226.

⚠ **CAUTION:** Since Java uses threads, the THREAD keyword must be coded in all modules that interact with Java. RPG relies heavily on static storage even in subprocedures that apparently only use automatic storage. THREAD keyword is necessary to ensure the correct handling of this static storage. This applies not only to modules that contain calls to Java methods, but also to any modules that might be called during interactions with Java, when the Java part of the application might be running with multiple threads.

See "Additional RPG Coding for Using Java" on page 225 for more information about the various JNI functions.

Example 1

In this example, the goal is to add two BigDecimal values together. In order to do this, two BigDecimal objects must be instantiated by calling the constructor for the BigDecimal class, fields must be declared to store the BigDecimal objects, and the add() method in the BigDecimal class must be called.

```
 *   Prototype the BigDecimal constructor that accepts a String
 *   parameter.  It returns a new BigDecimal object.
 *   Since the string parameter is not changed by the constructor, we will
 *   code the CONST keyword.  This will make it more convenient
 *   to call the constructor.
 *
D bdcreate1       PR              O    EXTPROC(*JAVA:
D                                       'java.math.BigDecimal':
D                                       *CONSTRUCTOR)
D    str                          O    CLASS(*JAVA:'java.lang.String')
D                                      CONST
 *
 *   Prototype the BigDecimal constructor that accepts a double
 *   parameter.  8F maps to the Java double data type and so must
 *   be passed by VALUE.  It returns a BigDecimal object.
 *
D bdcreate2       PR              O    EXTPROC(*JAVA:
D                                       'java.math.BigDecimal':
D                                       *CONSTRUCTOR)
D   double                       8F    VALUE
```

Figure 79. RPG Code Example Calling BigDecimal Java Class

```
 *   Define fields to store the BigDecimal objects.
 *
D bdnum1          S               O    CLASS(*JAVA:'java.math.BigDecimal')
D bdnum2          S               O    CLASS(*JAVA:'java.math.BigDecimal')
 *
 *   Since one of the constructors we are using requires a String object,
 *   we will also need to construct one of those.  Prototype the String
 *   constructor that accepts a byte array as a parameter.  It returns
 *   a String object.
 *
D makestring      PR              O    EXTPROC(*JAVA:
D                                       'java.lang.String':
D                                       *CONSTRUCTOR)
D    bytes                      30A    CONST VARYING

 *
 *  Define a field to store the String object.
 *
D string          S               O    CLASS(*JAVA:'java.lang.String')

 *
 *  Prototype the BigDecimal add method.  It accepts a BigDecimal object
 *  as a parameter, and returns a BigDecimal object (the sum of the parameter
 *  and of the BigDecimal object used to make the call).
 *
D add             PR              O    EXTPROC(*JAVA:
D                                       'java.math.BigDecimal':
D                                       'add')
D                                      CLASS(*JAVA:'java.math.BigDecimal')
D   bd1                           O    CLASS(*JAVA:'java.math.BigDecimal')
D                                      CONST

 *
 *  Define a field to store the sum. *
D sum             S               O    CLASS(*JAVA:'java.math.BigDecimal')
D
D double          S              8F    INZ(1.1)
D fld1            S             10A
 * Define a prototype to retrieve the String version of the BigDecimal
D getBdString     PR              O    CLASS(*JAVA:'java.lang.String')
D                                      EXTPROC(*JAVA:
D                                       'java.lang.BigDecimal':
D                                       'toString')
 * Define a prototype to retrieve the value of a String
D getBytes        PR         65535A    VARYING
D                                      EXTPROC(*JAVA:
D                                       'java.lang.String':
D                                       'getBytes')
 * Define a variable to hold the value of a BigDecimal object
D bdVal           S             63P 5
```

Here is the code that does the call.

```
 *  Call the constructor for the String class, to create a String
 *  object from fld1.  Since we are calling the constructor, we
 *  do not need to pass a String object as the first parameter.
 *
C                   EVAL      string = makestring('123456789012345678901234567890')
 *
 *  Call the BigDecimal constructor that accepts a String
 *  parameter, using the String object we just instantiated.
 *
C                   EVAL      bdnum1 = bdcreate1(string)

 *
 *  Call the BigDecimal constructor that accepts a double
 *  as a parameter.
 *
C                   EVAL      bdnum2 = bdcreate2(double)


 *
 *  Add the two BigDecimal objects together by calling the
 *  add method.  The prototype indicates that add accepts
 *  one parameter, but since add is not a static method, we
 *  must also pass a BigDecimal object in order to make the
 *  call, and it must be passed as the first parameter.
 *  bdnum1 is the object we are using to make the
 *  call, and bdnum2 is the parameter.
 *
C                   EVAL      sum = add(bdnum1:bdnum2)

 *  sum now contains a BigDecimal object with the value
 *  bdnum1 + bdnum2.
C                   EVAL      bdVal = %DECH(getBdString(sum) : 63 : 5)
 * val now contains a value of the sum.
 * If the value of the sum is larger than the variable "val" can
 * hold, an overflow exception would occur.
```

Example 2

This example shows how to perform a TRIM in Java by using the trim() method as an alternative to the ILE RPG %TRIM built-in function. The trim() method in the String class is not a static method, so a String object is needed in order to call it.

```
 *  Define a field to store the String object we wish to trim
 *
D str             S               O   CLASS(*JAVA:'java.lang.String')

D makestring      PR              O   EXTPROC(*JAVA:
D                                      'java.lang.String':
D                                      *CONSTRUCTOR)
D                                     CLASS(*JAVA:'java.lang.String')
D   parm                     65535A   CONST VARYING
 *
 *  Prototype the String method getBytes which converts a String to a byte
 *  array.  We can then store this byte array in an alpha field.
 *
D getBytes        PR         65535A   EXTPROC(*JAVA:
D                                       'java.lang.String':
D                                       'getBytes') VARYING

 *
 *  Prototype the String method trim.  It doesn't take any parameters,
 *  but since it is not a static method, must be called using a String
 *  object.
 *
D trimstring      PR              O   EXTPROC(*JAVA:
D                                      'java.lang.String':
D                                      'trim')
D fld             S             10A     INZ('   hello    ') VARYING
```

Figure 80. RPG Code Example Using trim() Java Method

The call is coded as follows:

```
 * Call the String constructor
 *
C                   EVAL      str = makestring(fld)

 *
 * Trim the string by calling the String trim() method.
 * We will reuse the str field to store the result.
 *
C                   EVAL      str = trimstring(str)

 *
 * Convert the string back to a byte array and store it
 * in fld.
 *
C                   EVAL      fld = getBytes(str)
```

Figure 81. RPG Call to the String constructor

Static methods are called in the same way, except that an object is not required to make a call. If the getBytes() method above was static, the call would look like the example below.

```
C                         EVAL      fld = getBytes()
```

If the method does not return a value, use the CALLP operation code.

Creating Objects

In order to call a non-static method, an object is required. The class of the object must be the same as the class containing the method. You may already have an object available, but you may sometimes need to instantiate a new object. You do this by calling a class constructor. A class constructor is neither a static method nor an instance method, and therefore it does not need an instance parameter. The special method name `*CONSTRUCTOR` is used when prototyping a constructor.

For example, class BigDecimal has a constructor that accepts a float parameter.

This constructor would be prototyped as follows:

```
D bdcreate        PR              O   EXTPROC(*JAVA:
D                                      'java.math.BigDecimal':
D                                      *CONSTRUCTOR)
D    dnum                       4F   VALUE
```

Note that the parameter must be passed by value because it maps to the Java float data type.

You would call this constructor like this:

```
D bd              S               O   CLASS(*JAVA:
D                                           'java.math.BigDecimal')
 /free
    bd = bdcreate(5.2E9);
 /end-free
```

The class of the returned object is the same as the class of the constructor itself, so the CLASS keyword is redundant for a constructor, but it may be coded.

Calling methods in your own classes

When you use your own Java classes, the class that you specify in the EXTPROC and CLASS keywords is simply the name of the class. If the class is part of a package, you include the package information in the keywords. For example, consider the following two classes:

```
class Simple
{
   static void method (void)
   {
      System.out.println ("Simple method");
   }
}
```

```
package MyPkg;

class PkgClass
{
   static void method (void)
   {
      System.out.println ("PkgClass method");
   }
}
```

If the Simple class file is /home/myclasses/Simple.class, you would specify the directory /home/myclasses in your CLASSPATH environment variable, and you would specify 'Simple' as the class name in your RPG keywords.

If the PkgClass class file is /home/mypackages/MyPkg/PkgClass.class, you would specify the directory /home/mypackages (the directory containing the package) in your CLASSPATH environment variable, and you would specify 'MyPkg.PkgClass' (the package-qualified Java class) as the class name in your RPG keywords.

The class name for your RPG keywords is the same name as you would specify in your import statements in your Java classes. You use the CLASSPATH environment variable to specify the location of the class files, or the location of the directory containing the package.

Note: Note: If you have classes in a jar file, you specify the jar file itself in your classpath.

```
===> ADDENVVAR CLASSPATH '/home/myclasses:/home/mypackages:/home/myjarfiles/j1.jar'
```

```
D simpleMethod    PR                   EXTPROC(*JAVA
D                                            : 'Simple'
D                                            : 'method')
D                                      STATIC
D pkgMethod       PR                   EXTPROC(*JAVA
D                                            : 'Pkg.PkgClass'
D                                            : 'method')
D                                      STATIC
```

Figure 82. Creating an RPG prototype for a Java method in a package

Controlling how the Java Virtual Machine is set up

When RPG starts the Java Virtual Machine (JVM), there are several options that control how the JVM is started. See the *Java System Properties* section in the IBM i Information Center.

- You can place these options in the SystemDefault.properties file.
- You can use the CLASSPATH environment variable to specify the classpath (see above).
- You can place these options in an environment variable called QIBM_RPG_JAVA_PROPERTIES. Any options placed in this environment variable will override the options in the SystemDefault.properties file. If you specify the java.class.path option in this environment variable, and you also specified the CLASSPATH environment variable, it is undefined which value will take precedence for the classpath.

To specify options in the QIBM_RPG_JAVA_PROPERTIES environment variable, you code the options in a string, one after the other, separated by any character that does not appear in any of the options. Then you end the string with the separator character. For example, if you want to specify the options

```
java.version=1.4
os400.stderr=file:stderr.txt
```

then you would add the environment variable using the following command:

```
ADDENVVAR ENVVAR(QIBM_RPG_JAVA_PROPERTIES)
VALUE('-Djava.version=1.4;-Dos400.stderr=file:stderr.txt;')
```

If the options string is not valid, Java may reject one of the options. Message JVAB55A will appear in the joblog indicating which option was not valid. If this happens, RPG will try to start the JVM again without any of the options, but still including the java.class.path option if it came from the CLASSPATH environment variable.

Some parameters and return values require conversion between the job CCSID and the CCSID that Java uses for byte arrays. The file.encoding Java property is used by RPG to obtain the CCSID that Java uses. Ensure that the file.encoding property is set correctly for your job CCSID. You can allow Java to set the property implicitly using attributes of your job, or you can set the property explicitly using one of the mechanisms above. For example, you could add '-Dfile.encoding=ISO8859_1' or '-Dfile.encoding=Cp948' to your QIBM_RPG_JAVA_PROPERTIES environment variable. For more information about the file.encoding property, see the IBM Developer Kit for Java topic in the Information Center.

For more information about the properties you can specify to control the Java virtual machine, and other environment variables that may be required for Java to operate correctly, such as the QIBM_USE_DESCRIPTOR_STDIO environment variable, see the see the IBM Developer Kit for Java topic in the Information Center. You may also need to consult the documentation for the specific Java classes that you will be using.

RPG Native Methods

To define an RPG native method, you code the prototype the same way as you would code the prototype for an ordinary Java method. Then, you write the RPG subprocedure normally. You must code the EXPORT keyword on the Procedure-Begin Specification for the native method.

You must have your native methods in a service program in your library list. In your Java class that is calling your native methods, you must have a static statement like this:

```
        static
        {
             System.loadLibrary ("MYSRVPGM");
        }
```

This will enable Java to find your native methods. Aside from adding *JAVA and the class to the EXTPROC keyword for the prototype of a native method, you write your native method like any subprocedure. Figure 83 on page 221 is an example of a Java class that calls a native method.

CAUTION: If you are using environment variables to control how the JVM is started, you must be sure that the environment variables exist in the job before any RPG programs call Java methods. If you use ADDENVVAR LEVEL(*SYS), the environment variable will be added at the system level, and by default, every job will start with that environment variable set. If you do this, be sure that the classpath includes all the directories containing the Java classes that may be needed by any application on the system.

```
        class MyClass
        {

           static
           {
             System.loadLibrary ("MYSRVPGM");
           }

          native boolean checkCust (byte custName[]);

           void anotherMethod ()
           {
              boolean found;
              // call the native method
              found = checkCust (str.getBytes());
            }
         }
```

Figure 83. Java Class Calling a Native Method

Figure 84 on page 221 is a prototype of an RPG native method.

```
D checkCust       PR              N   EXTPROC(*JAVA
D                                           : 'MyClass'
D                                           : 'checkCust')
D   custName                   100A   VARYING CONST
```

Figure 84. RPG Native Method Prototype

The native method itself is coded just like any subprocedure. Figure 85 on page 221 is an example of a native method coded in RPG.

```
P checkCust       B                   EXPORT
D checkCust       PI              N
D   custName                   100A   VARYING CONST
 /free   chain custName  rec;
   return %found;
 /end-free
P checkCust       E
```

Figure 85. Native Method Coded in RPG

Java calls your service program from the default activation group. If your service program is created with activation group *CALLER, it will run in the default activation group. This can sometimes cause problems:

- If you are debugging your native methods, and you want to make a change to the code, you will have to sign off and sign back on again before Java will see the new version.
- If you are calling other procedures in the service program from other RPG code that is not running in the default activation group, then you will not be able to share any global variables between the "ordinary procedures" and the native methods. This scenario can arise if a procedure in your RPG service program sets up some global variables, and then calls a Java class which then calls a native method in that service program. Those native methods will not see the same data that the first procedure set up.

If you create any Java objects in your native methods, by default they will be destroyed by Java when the native method returns. If you want the object to be available after the native method returns (for example, if you want to use it from another native method later), then you must tell Java that you want to make a global reference, by calling the JNI wrapper procedure getNewGlobalRef . When you are finished with the global reference, you will call JNI wrapper procedure freeGlobalRef, so Java can reclaim the object. See “Telling Java you want an object to be permanent” on page 226 and “Telling Java you are finished with a permanent object” on page 227 for more information about these wrapper procedures.

If your RPG native method ends abnormally with an unhandled exception, the RPG compiler will throw an exception to Java. The exception is of class java.lang.Exception, and has the form RPG nnnnn, where nnnnn is the RPG status code.

```
               try
               {
                  nativeMethod ();
               }
               catch (Exception exc)
               {
                  …
               }
```

Getting the Instance Parameter in Non-Static Native Methods

When a non-static native method is called, one of the parameters that Java passes to the native method is the object that the method applies to. This is called the "instance parameter", referred to as "this" in a Java method. Within the native method itself, you can use the built-in function %THIS to get the instance parameter. You do not code this parameter in your Procedure Interface.

Passing Character Parameters from Java to Native Methods

You have two choices when dealing with character parameters:

- If you want your Java code to be a simple as possible, define the parameter as a String in your Java native method declaration. Your RPG code would have to retrieve the value of the string itself (see “Using String Objects in RPG” on page 222).
- If you want the character data to be immediately available to your RPG program, code the parameter in the Java native method declaration as a byte array or a char array, and code it in your RPG prototype as a character field, UCS-2 field, or a Date, Time or Timestamp. That way, RPG will handle the conversion for you.

Using String Objects in RPG

If you have a String object in your RPG code, you can retrieve its length and contents using the code in Figure 86 on page 222.

```
D stringBytes     PR           100A   VARYING
D                                     EXTPROC(*JAVA
D                                           : 'java.lang.String'
D                                           : 'getBytes')
D stringLength    PR                  like(jint)
D                                     EXTPROC(*JAVA
D                                           : 'java.lang.String'
D                                           : 'length')
D string          S                   like(jstring)
D len             S                   like(jint)
D data            S            100A   VARYING
 /free      len = stringLength (string);
      data = stringBytes (string);
      if (len > %len(data));
            error ('Actual string was too long');
      endif;
 /end-free
```

Figure 86. Retrieving String object length and contents from Java

You can define the returned value from the getBytes method as character data of any length, either varying or non-varying, choosing the length based on your own knowledge of the length of data in the Java String. You can also define the return value as a Date, Time or Timestamp, if you are sure that the String object will have the correct format.

Alternately, you can retrieve the string value as a UCS-2 value, by calling the getChars method instead of getBytes.

Coding Errors when calling Java from RPG

Incorrectly specifying the method parameters in the RPG prototype

When coding the prototype for a Java method, if you do not specify the types of the return value and parameters correctly, the RPG compiler will build the method signature incorrectly. When the program is run, either the wrong method will be called, or the call will fail with a NoSuchMethodError Java exception.

If the call fails with a NoSuchMethodError Java exception, the RPG error message will indicate the signature that was used for the method call. The following table shows the mappings between Java types and method signature values. Refer to Table 65 on page 212 to see the mapping between Java types and RPG types.

Java type	Signature
boolean	Z
byte	B
char	C
short	S
int	I
long	J
float	F
double	D
any object	Lclass;
any array	[type

To see the list of valid signatures for the methods in the Java class, use the QSH command

```
 javap -s classname
```

where classname is specified with the package, for example java.lang.String. If the class is not in the standard classpath, you can specify a classpath option for javap:

```
javap -s classname -classpath classlocation
```

By comparing the valid signatures for the method with the signature being used by RPG for your method call, and working from the mapping tables, you should be able to determine the error in your prototype.

Failure to free Java resources

When you create a Java object by calling a constructor, or by calling a method that returns an object, that object will remain in existence until it is freed. It is freed when:

1. The RPG program calls a JNI function to free the object (see "Additional RPG Coding for Using Java" on page 225).
2. When the native method returns, if the object was created during a call from Java to a native method.
3. When the JVM ends.

If the RPG procedure calling the Java method is not itself an RPG native method, and the RPG procedure does not take care to free objects it has created, then the job may eventually be unable to create any more objects.

Consider the following code fragment:

```
           strObject = newString ('abcde');
           strObject = trim (strObject);
           data = getBytes (strObject);
           freeLocalRef (strObject);
```

It appears that this code is taking care to free the object, but in fact this code creates two objects. The first object is created by the called to newString(), and the second is created by the call to trim(). Here are two ways to correct this code fragment:

1. By freeing several objects at once:

   ```
           beginObjGroup();
           strObject = newString ('abcde');
           strObject = trim (strObject);
           data = getBytes (strObject);
           endObjGroup();
   ```

2. By keeping track of all objects used, and freeing them individually:

   ```
           strObject = newString ('abcde');
           trimmedStrObject = trim (strObject);
           data = getBytes (trimmedStrObject);
           freeLocalRef (strObject);
           freeLocalRef (trimmedStrObject);
   ```

Another problem can be created by calling Java methods as parameters to other Java methods. In the following example, the program is creating a BigDecimal object from the constructor that takes a String parameter:

```
           bigDec = newBigDecimal (newString ('12.345'));
           …
           freeLocalRef (bigDec);
```

The problem with this code is that a String object has been created for the parameter, but it can never be freed by the RPG procedure. This problem can be corrected by calling **`beginObjGroup()`** before the RPG code that calls Java and calling **`endObjGroup()`** after, or by coding as follows:

```
           tempObj = newString ('12.2345');
           bigDec = newBigDecimal (tempObj);
           freeLocalRef (tempObj);
           …
           freeLocalRef (bigDec);
```

Using objects that no longer exist

If you have static Object variables in your native method (STATIC keyword on the definition), or your native method uses static global Object variables (variables declared in the main source section), then the Object variables will retain their values between calls to the native method. However, by default, Java will free any objects created during a call to a native method. (See "Additional RPG Coding for Using Java" on page 225 to see how to prevent Java from freeing objects.)

An RPG "Object" is really a numeric object reference. When a Java object is freed, the numeric object reference can be reused. If the RPG native method refers to a static Object variable that has not been explicitly protected from being freed, one of two things can happen:

1. The object reference may be invalid, if the numeric object reference has not been reused.
2. The object reference may have been reused, but since it refers to a different object, any attempt to use it in the RPG native method will probably be incorrect.

To prevent problems with attempting to reuse objects illegally, the RPG programmer may do one or more of the following:

- Avoid declaring any Object variables in static storage. Instead, declare all Object variables in local storage of subprocedures, without using the STATIC keyword.
- Before returning from a native method, explicitly set all static object references to *NULL.
- Upon entering a native method, explicitly set all static object references to some initial values.

Additional RPG Coding for Using Java

When you are using ILE RPG with Java, there are some functions normally handled by Java that must be handled by your RPG code. The RPG compiler takes care of some of these for you, but you must handle some of them yourself. This section shows you some sample RPG wrappers to do this work, explains how and when to call them, and suggests how to handle JNI exceptions.

The module that you create to hold these JNI wrapper functions should begin with the following statements:

```
     H thread(*serialize)
     H nomain
     H bnddir('QC2LE')
      /define OS400_JVM_12
      /copy qsysinc/qrpglesrc,jni
      /copy JAVAUTIL
```

The following RPG wrappers for JNI functions are described. See Figure 92 on page 230 below for a complete working example. See Copy-file JAVAUTIL. for the copy file containing the prototypes and constants for the wrapper functions.

- “Telling Java to free several objects at once” on page 225
- “Telling Java you are finished with a temporary object” on page 226
- “Telling Java you want an object to be permanent” on page 226
- “Telling Java you are finished with a permanent object” on page 227
- “Creating the Java Virtual Machine (JVM)” on page 228
- “Obtaining the JNI environment pointer” on page 228

Telling Java to free several objects at once

You can free many local references at once by calling the JNI function PushLocalFrame before a section of RPG code that uses Java and then calling PopLocalFrame at the end of the section of RPG code. When you call PopLocalFrame, any local references created since the call to PushLocalFrame will be freed. For more information about the parameters to these JNI functions, see the JNI documentation at http://java.sun.com.

```
  *-----------------------------------------------------------------
  * beginObjGroup - start a new group of objects that can all
  *                 be deleted together later
  *-----------------------------------------------------------------
 P beginObjGroup   b                   export
 D beginObjGroup   pi            10i 0
 D   env                           *   const
 D   capacityParm                10i 0 value options(*nopass)
 D rc              s             10i 0
 D capacity        s             10i 0 inz(100)
  /free
         JNIENV_p = env;
         if (%parms >= 2);
             capacity = capacityParm;
         endif;
         rc = PushLocalFrame (JNIENV_p : capacity);
         if (rc <> 0);
             return JNI_GROUP_NOT_ADDED;
         endif;
         return JNI_GROUP_ADDED;
  /end-free
 P beginObjGroup   e

  *-----------------------------------------------------------------
  * endObjGroup - end the group of objects that was started
  *               most recently
  *-----------------------------------------------------------------
 P endObjGroup     b                   export
 D endObjGroup     pi            10i 0
 D   env                           *   const
 D   refObjectP                    o   class(*java:'java.lang.Object')
 D                                     const
 D                                     options(*nopass)
 D   newObjectP                    o   class(*java:'java.lang.Object')
```

```
D                                      options(*nopass)
D retVal          s               o   class(*java:'java.lang.Object')
D refObject       s                    like(refObjectP) inz(*null)
D newObject       s                    like(newObjectP)
 /free
        JNIENV_p = env;
        if %parms() >= 2;
            refObject = refObjectP;
        endif;
        newObject = PopLocalFrame (JNIENV_p : refObject);
        if %parms() >= 3;
             newObjectP = newObject;
        endif;
        return JNI_GROUP_ENDED;
 /end-free
P endObjGroup     e
```

Note: You need the JNI environment pointer (described in "Obtaining the JNI environment pointer" on page 228 below) to call this wrapper.

Telling Java you are finished with a temporary object

If you have created an object using a Java constructor, or if you have called a Java method that returned an object to you, this object will only be available to be destroyed by Java's garbage collection when it knows you do not need the object any more. This will happen for a native method (called by java) when the native method returns, but otherwise it will never happen unless you explicitly inform Java that you no longer need the object. You do this by calling the RPG wrapper procedure freeLocalRef.

```
CALLP  freeLocalRef (JNIEnv_P : string);
```

Figure 87 on page 226 contains the sample source code for freeLocalRef.

```
/*------------------------------------------------------*/
/* freeLocalRef                                         */
/*------------------------------------------------------*/

P freeLocalRef...
P                 B                   EXPORT
D freeLocalRef...
D                 PI
D   env                          *    VALUE
D   localRef                     O    CLASS(*JAVA
D                                         : 'java.lang.Object')
D                                     VALUE

 /free
     jniEnv_P = env;
     DeleteLocalRef (env : localRef);

 /end-free

P freeLocalRef...
P                 E
```

Figure 87. Source Code for freeLocalRef

Note: You need the JNI environment pointer (described in "Obtaining the JNI environment pointer" on page 228 below) to call this wrapper.

Telling Java you want an object to be permanent

If you have a reference to a Java object that was either passed to you as a parameter or was created by calling a Java method or constructor, and you want to use that object after your native method returns, you must tell Java that you want the object to be permanent, or "global". Do this by calling the RPG wrapper procedure getNewGlobalRef and saving the result in a global variable.

```
EVAL   globalString = getNewGlobalRef (JNIENV_P : string);
```

Figure 88 on page 227 contains the sample source code for getNewGlobalRef.

```
/*------------------------------------------------------*/
/* getNewGlobalRef                                      */
/*------------------------------------------------------*/

P getNewGlobalRef...
P                 B                   EXPORT
D getNewGlobalRef...
D                 PI              O    CLASS(*JAVA
D                                          : 'java.lang.Object')
D   env                           *    VALUE
D   localRef                      O    CLASS(*JAVA
D                                          : 'java.lang.Object')
D                                      VALUE

 /free
     jniEnv_P = env;
     return NewGlobalRef (env : localRef);
 /end-free
P getNewGlobalRef...
P                 E
```

Figure 88. Source Code for getNewGlobalRef

Note: You need the JNI environment pointer (described in "Obtaining the JNI environment pointer" on page 228 below) to call this wrapper.

Telling Java you are finished with a permanent object

If you have created a global reference, and you know that you no longer need this object, then you should tell Java that as far as you are concerned, the object can be destroyed when Java next performs its garbage collection. (The object will only be destroyed if there are no other global references to it, and if there are no other references within Java itself.) To tell Java that you no longer need the reference to the object, call the RPG wrapper procedure freeGlobalRef .

```
CALLP  freeGlobalRef (JNIEnv_P : globalString);
```

Figure 89 on page 228 contains sample source code for freeGlobalRef.

```
/*-----------------------------------------------------*/
/* freeGlobalRef                                       */
/*-----------------------------------------------------*/
P freeGlobalRef...
 P                 B                     EXPORT
 D freeGlobalRef...
 D                 PI
 D   env                           *     VALUE
 D   globalRef                     O     CLASS(*JAVA
 D                                           : 'java.lang.Object')
 D                                       VALUE

 /free
    jniEnv_P = env;
    DeleteGlobalRef (env : globalRef);
    /end-free

P freeGlobalRef...
 P                 E
```

Figure 89. Source Code for freeGlobalRef

Note: You need the JNI environment pointer (described in "Obtaining the JNI environment pointer" on page 228 below) to call this wrapper.

Creating the Java Virtual Machine (JVM)

If the JVM has not already been created when your RPG code is ready to call a Java method, RPG will create the JVM for you. See Controlling how the Java Virtual Machine is set upfor information on the ways you can control the Java class path and other Java properties that will be set when the JVM is started.

Obtaining the JNI environment pointer

If you need to call any JNI functions, use the /COPY file JNI from QSYSINC/QRPGLESRC. Most of the JNI functions are called through a procedure pointer. The procedure pointers are part of a data structure that it itself based on a pointer called the "JNI environment pointer". This pointer is called JNIEnv_P in the JNI /COPY file. To obtain this pointer, call the JNI wrapper procedure getJniEnv.

```
EVAL   JNIEnv_P = getJniEnv();
```

Figure 90 on page 229 contains sample source code for getJniEnv.

```
 *-----------------------------------------------------------------
 * getJniEnv - get the JNI environment pointer
 * Note: This procedure will cause the JVM to be created if
 *       it was not already created.
 *-----------------------------------------------------------------
P getJniEnv       b                   export
D getJniEnv       pi              *

D attachArgs      ds                  likeds(JavaVMAttachArgs)
D env             s               *   inz(*null)
D jvm             s                   like(JavaVM_p) dim(1)
D nVms            s                   like(jsize)
D rc              s             10i 0
D obj             s               o   class(*java
D                                         : 'java.lang.Integer')
D newInteger      pr              o   extproc(*java
D                                           : 'java.lang.Integer'
D                                           : *constructor)
D   value                       10i 0 value
 /free
    monitor;
       // Get the current JVM
       rc = JNI_GetCreatedJavaVMs(jvm : 1 : nVms);
       if (rc <> 0);
          // Some error occurred
          return *null;
       endif;  if (nVms = 0);
          // The JVM is not created yet.  Call a Java
          // method to get the RPG runtime to start the JVM
          obj = newInteger(5);

          // Try again to get the current JVM
          rc = JNI_GetCreatedJavaVMs(jvm : 1 : nVms);
          if (rc <> 0
          or  nVms = 0);
             // Some error occurred
             return *null;
          endif;
       endif;
       // Attach to the JVM
       JavaVM_P = jvm(1);
       attachArgs = *allx'00';
       attachArgs.version = JNI_VERSION_1_2;
       rc = AttachCurrentThread (jvm(1) : env
                               : %addr(attachArgs));
       if (rc <> 0);
          return *null;
       endif;

       // Free the object if we created it above while
       // getting the RPG runtime to start the JVM
       if obj <> *null;
          freeLocalRef (env : obj);
       endif;
    on-error;
       return *null;
    endmon;
    return env;
 /end-free
P getJniEnv       e
```

Figure 90. Source Code for getJniEnv

```
 *-----------------------------------------------------------------
 * Copy file JAVAUTIL
 *-----------------------------------------------------------------
 /if defined(JAVAUTIL_COPIED)
 /eof
 /endif
 /define JAVAUTIL_COPIED
D JNI_GROUP_ADDED...
D                c                   0
D JNI_GROUP_NOT_ADDED...
D                c                   -1
D JNI_GROUP_ENDED...
D                c                   0
D beginObjGroup   pr            10i 0 extproc('beginObjGroup')
D   env                          *   const
D   capacityParm                10i 0 value options(*nopass)
D endObjGroup     pr            10i 0 extproc('endObjGroup')
D   env                          *   const
D   refObjectP                   o   class(*java:'java.lang.Object')
D                                    const
D                                    options(*nopass)
D freeLocalRef...
D                pr                  extproc('freeLocalRef')
D   env                          *   value
D   localRef                     o   CLASS(*JAVA
D                                        : 'java.lang.Object')
D                                    value
D getNewGlobalRef...
D                pr              o   class(*JAVA
D                                        : 'java.lang.Object')
D                                    extproc('getnewGlobalRef')
D   env                          *   value
D   localRef                     o   class(*JAVA
D                                        : 'java.lang.Object')
D                                    value
D freeGlobalRef...
D                pr                  extproc('freeGlobalRef')
D   env                          *   value
D   globalRef                    O   class(*JAVA
D                                        : 'java.lang.Object')
D                                    value
D getJniEnv       pr              *   extproc('getJniEnv')
```

Figure 91. Copy-file JAVAUTIL

```
Java class


class TestClass{
   String name = "name not set";

   TestClass (byte name[]) {
      this.name = new String(name);
   }

   void setName (byte name[]) {
      this.name = new String(name);
   }

   String getName () {
      return this.name;
   }
}
```

Figure 92. Using the wrappers for the JNI functions

```
RPG program


H THREAD(*SERIALIZE)
H BNDDIR('JAVAUTIL')
 // (JAVAUTIL is assumed to the binding directory that lists
 // the service program containing the procedures described
 // below)

 /copy JAVAUTIL
 // (JAVAUTIL is assumed to be the source member containing the
 // prototypes for the procedures described below)


D TestClass       C                   'TestClass'
D StringClass     C                   'java.lang.String'
D newTest         PR              O   EXTPROC(*JAVA : TestClass
D                                           : *CONSTRUCTOR)
D   name                        25A   VARYING CONST

D getName         PR              O   CLASS(*JAVA : StringClass)

D                                     extproc(*JAVA : TestClass
D                                           : 'getName')

D setName         PR                  extproc(*JAVA : TestClass
D                                           : 'setName')
D   newName                     25A   VARYING CONST

D newString       PR              O   EXTPROC(*JAVA : StringClass
D                                           : *CONSTRUCTOR)
D   value                    65535A   VARYING CONST

D nameValue       PR            25A   VARYING
D                                     extproc(*JAVA : StringClass
D                                           : 'getBytes')

D myTestObj       S                   LIKE(newTest)
D myString        S                   LIKE(newString)
D env             S                   LIKE(getJniEnv)
 /free

    // Get the JNI environment pointer so that JNI functions
    // can be called.
    env = getJniEnv();


    // Set the beginning marker for an "object group"
    // so that any objects created between now and the
    // "end object group" can be freed all at once.

    beginObjGroup (env);

    // Create a Test object to work with
    // We do not want this object to be freed with the
    // other objects in the object group, so we make it
    // a permanent object

    myTestObj = newTest ('RPG Dept');
    myTestObj = getNewGlobalRef (env : myTestObj);

    // Get the current "name" from the Test object
    // This creates a local reference to the Name object

    myString = getName (myTestObj);
    dsply (nameValue(myString));

    // Change the name

    setName (myTestObj : 'RPG Department');

    // Get the current "name" again.  This will cause
    // access to the previous local reference to the old name
    // to be lost, making it impossible for this RPG
    // program to explicitly free the object.  If the object
    // is never freed by this RPG program, Java could never
    // do garbage-collection on it, even though the old String
    // object is not needed any more.  However, endObjGroup
    // will free the old reference, allowing garbage collection

    myString = getName (myTestObj);
    dsply (nameValue(myString));

    // End the object group.  This will free all local
    // references created since the previous beginObjGroup call.
    // This includes the two references created by the calls
    // to getName.

    endObjGroup (env);

    // Since the original Test object was made global, it can
    // still be used.

    setName (myTestObj : 'RPG Compiler Dept');

    // The original Test object must be freed explicitly
    // Note: An alternative way to handle this situation
    //       would be to use nested object groups, removing
    //       the need to create a global reference
    //         beginObjGroup ------------.
    //         create myTestObj          |
    //         beginObjGroup  ---------. |
    //         ...                     | |
    //         endObjGroup    ---------' |
    //         use myTestObj again       |
    //         endObjGroup   ------------'

    freeGlobalRef (env : myTestObj);

    return;

 /end-free
```

Handling JNI Exceptions

In ILE RPG, an exception causes an exception message to be signaled. Programs do not need to check explicitly for exceptions; instead, you can code exception handlers to get control when an exception occurs. You only have to handle JNI exceptions yourself when you are making your own JNI calls. When a call to a JNI function results in an unhandled Java exception, there is no accompanying exception message. Instead, the JNI programmer must check whether an exception occurred after each call to a JNI function. This is done by calling the ExceptionOccurred JNI function, which returns a Java Exception object (or the Java null object which has a value of 0 in the JNI). Once you have determined that an exception has occurred, the only JNI calls you can make are ExceptionClear and ExceptionDescribe. After you have called ExceptionClear, you are free to make JNI calls again. If you make a non-exception JNI call before calling ExceptionClear, the exception will disappear, and you will not be able to get any further details. RPG always converts a JNI exception into an RPG exception (it signals one of the RNX030x messages, depending on the RPG function that was being done at the time).

Tip!

You may want to include this type of exception-handling code in your versions of the JNI wrapper procedures above.

Additional Considerations

Common Runtime Errors

The compiler will not attempt to resolve classes at compile time. If a class cannot be located at run time, a runtime error will occur. It will indicate that an *UnresolvedLinkException* object was received from the Java environment.

The compiler does no type checking of parameters at compile time. If there is a conflict between the prototype and the method being called, an error will be received at run time.

Debugging Hints

A Java object is viewed as an object reference in RPG. This object reference is an integer value, which behaves like a pointer. Normal object references are positive values, assigned in increasing order from 1. Global references, which can be created using JNI function NewGlobalRef , are negative values. These values are assigned in increasing order from the smallest negative number (-2147483647).

Normally, these values are not visible within the RPG code. However, this information may be useful when debugging RPG code.

Creating String objects in RPG

If you need a String object to pass to a Java method, you can create it like this:

```
D newString       PR               O   EXTPROC(*JAVA
D                                            : 'java.lang.String'
D                                            : *CONSTRUCTOR)
D   value                     65535A   CONST VARYING
D string          S                    like(jstring)
  /free
     string = newString ('abcde');
     …
  /end-free
```

If you want to create a string with UCS-2 data or graphic data, use this code:

```
D newStringC      PR               O   EXTPROC(*JAVA
D                                            : 'java.lang.String'
D                                            : *CONSTRUCTOR)
D   value                     16383C   CONST VARYING
D string          S                    like(jstring)
D graphicData     S             15G
D ucs2Data        S            100C
  /free
     string = newStringC (%UCS2(graphicData));
     …
     string = newStringC (ucs2Data);
 /end-free
```

Getting information about exceptions thrown by called Java methods

When RPG calls a Java method that ends with an exception, RPG handles the Java exception and signals escape message RNX0301. This message has the string value of the Exception, but it does not have the trace information that is normally available when Java calls a method that ends with an exception.

If you want to see the Java exception trace information, do the following:

1. ADDENVVAR ENVVAR(QIBM_USE_DESCRIPTOR_STDIO) VALUE('Y')

   Note: This step must be done before the JVM is started.
2. Ensure that the os400.stderr option in your SystemProperties.default file is set to file:myfilename, for example os400.stderr=file:/home/mydir/stderr.txt. See Controlling how the Java Virtual Machine is set up.

   Note: This step must be done before the JVM is started.
3. ADDENVVAR ENVVAR(QIBM_RPG_JAVA_EXCP_TRACE) VALUE('Y')

   Note: This step can be done at any time. To stop the exception trace being done by RPG, you can remove the environment variable, or set it to a value other than 'Y'.
4. After the exception has occurred, the trace information will be in the file that you specified in the os400.stderr option.

Advanced JNI Coding

The RPG IV compiler support for calling Java methods and for writing RPG native methods hides almost all the JNI coding from the RPG programmer. However, RPG's support is not necessarily the most efficient. For example, it always converts arrays between RPG and Java on calls and on entry and exit from native methods, but you may want to handle your own array conversions to improve performance.

The RPG support only gives you access to Java methods. If you want to access the fields in a class, you would have to add "get" and "set" methods to the Java class, or do JNI coding (see “Accessing Fields in Java Classes ” on page 236).

Figure 93 on page 234 is an example of a JNI call in RPG.

```
 /COPY JNI
D objectId        s                   like(jobject)
D methodId        s                   like(jmethodID)
D string          s                   like(jstring)
D parms           ds                   likeds(jvalue) dim(3)

 /free
  parms(1).i = 10;                // parameter 1 is an int
  parms(2).l = refToInt(string);  // parameter 2 is an object
  parms(3).d = 2.5e3;             // parameter 3 is a double
  CallVoidMethodA (JNIEnv_P : objectId : methodId : parms);
 /end-free
```

Figure 93. JNI Call in RPG

Note that the pointer JNIEnv_P is defined in the JNI /COPY file.

Setting an Object Reference in the jvalue Structure

The jvalue structure looks like this:

```
            D jvalue          DS                    QUALIFIED
            D                                       BASED(jvalue_P)
             ... more subfields ...
            D   l                                   LIKE(jint)
            D                                       OVERLAY(jvalue:1)
```

The "l" subfield of the jvalue structure represents an Object reference, but RPG does not support subfields of type Object. Since Object references are actually 4-byte integers, the "l" subfield of the "jvalue" data structure is defined as a 4-byte integer rather than as an Object. To assign an RPG Object type to the jvalue.l subfield, you must write a procedure and a "spoofing" prototype that will trick the RPG compiler into treating an object reference as an integer. You create one procedure that simply takes an integer parameter and returns an integer (procedure "refIntConv" in the example below. Then you create two prototypes that call this procedure using a procedure pointer; one procedure defines the return type as type Object (procedure "intToRef" in the example below), and the other procedure defines the parameter as type Object (procedure "refToInt" in the example below). To convert between Object and Integer types, you call either prototype refToInt or IntToRef.

```
 *-----------------------------------------------------------------
 * refIntConv_procptr:
 *    This procedure pointer is set to the address of a
 *    procedure that takes an integer parameter and returns
 *    an integer.
 *    Since an object refererence is actually an integer, we
 *    can define prototypes that define either the return value
 *    or the parameter as an object reference, to trick the RPG
 *    compiler into allowing the Object reference to be passed
 *    to or returned from the procedure.
 *    Note: This type of trickery is not normally recommended,
 *       but it is necessary in this case to circumvent the RPG
 *       restriction against Object subfields.
 *-----------------------------------------------------------------
D refIntConv_name...
D                 c                   'refIntConv'
D refIntConv_procptr...
D                 s               *   procptr
D                                     inz(%paddr(refIntConv_name))
 *-----------------------------------------------------------------
 * refToInt - convert an object reference to an integer
 *-----------------------------------------------------------------
D refToInt        pr            10i 0 extproc(refIntConv_procptr)
D   ref                           o   class(*java:'java.lang.Object')
D                                     value

 *-----------------------------------------------------------------
 * intToRef - convert an integer to an object reference
 *-----------------------------------------------------------------
D intToRef        pr              o   class(*java:'java.lang.Object')
D                                     extproc(refIntConv_procptr)
D   int                         10i 0 value
```

Figure 94. /COPY JNICONV_PR with prototypes for spoofing procedures to convert between Object and integer types

```
H NOMAIN

 /COPY JNICONV_PR

 *-----------------------------------------------------------------
 * refIntConv is used with prototypes refToInt and intToRef
 * to convert between Object and integer types in RPG.
 * See JNICONV_PR for more details.
 *-----------------------------------------------------------------
D refIntConv      pr            10i 0 extproc(refIntConv_name)
D    parm                       10i 0 value

 *-----------------------------------------------------------------
 * The procedure simply returns its parameter.
 *-----------------------------------------------------------------
P refIntConv      B                   export
D refIntConv      pi            10i 0
D    parm                       10i 0 value
 /free
    return parm;
 /end-free
P refIntConv      E
```

Figure 95. Procedure to convert between Object and integer types

```
 /copy QSYSINC/QRPGLESRC,JNI
 /copy JNICONV_PR
D jvals           ds                  likeds(jvalue) dim(5)
D myString        s               o   class(*java:'java.lang.String')
D newString       pr              o   extproc(*java:'java.lang.String'
D                                           : *constructor)
D   val                        100a   const varying
 /free
      myString = newString('Hello');
      // Set the myString reference in the first jvalue element
      jvals(1).l = refToInt (myString);
      . . .
      // Set the myString reference from the second jvalue element
      myString = intToRef(jvals(2).l);
      . . .
      return;
```

Figure 96. Using the conversion prototypes

Converting Java Character Data

In Java, character data is ASCII rather than EBCDIC, so you will have to ensure that class names, method names, and field names are in ASCII for calls to JNI functions like FindClass. Character data that comes from Java is ASCII. To use it in your RPG program, you will probably want to convert it to EBCDIC. The RPG compiler handles these conversions for you, but if you are making the JNI calls yourself, you will have to do the conversions between ASCII and EBSDIC.

Accessing Fields in Java Classes

RPG only supports calling Java methods; it does not support accessing Java fields. Normally, fields can be accessed through "get" and "set" methods, but it is also possible to access fields using JNI calls. Here is an example showing JNI calls necessary to access the fields of a Java class or object.

Note: This example is intended to be an example of using the JNI. It is not intended to be a recommendation to access fields directly rather than using "get" and "set" methods.

```
 *------------------------------------------------------------------
 * This example shows how to use JNI to access the fields of a
 * class or an object.
 *
 * This program creates a Rectangle object and accesses the
 * width and height variables directly, using JNI calls.
 *
 * In this particular case, the getWidth(), getHeight,
 * setWidth() and setHeight() methods could have been used
 * to access these fields, avoiding the use of JNI calls.
 *------------------------------------------------------------------
H THREAD(*SERIALIZE)
 /DEFINE JNI_COPY_FIELD_FUNCTIONS
 /COPY JNI
 /COPY JAVAUTIL

 *------------------------------------------------------------------
 * JAVA classes and methods
 *------------------------------------------------------------------
```

Figure 97. Using JNI to Access Fields of Java Classes and Objects

```
D Rectangle       C                   'java.awt.Rectangle'
D NewRectangle    PR              O   EXTPROC(*JAVA
D                                           : Rectangle
D                                           : *CONSTRUCTOR)
D     x                         10I 0 VALUE
D     y                         10I 0 VALUE
D     width                     10I 0 VALUE
D     height                    10I 0 VALUE
 *------------------------------------------------------------------
 * Constants with ASCII representations of Java names
 *------------------------------------------------------------------
 * One way to determine these values is to use %UCS2 to convert
 * a character value to UCS-2, and display the result in hex
 * in the debugger.
 *
 * The ASCII value is in every second byte of the UCS-2 characters.
 *
 *     For example, %UCS2('abc') = X'006100620063'
 *                                     --  --  --
 *     The ASCII representation of 'abc' is X'616263'
 *------------------------------------------------------------------
D ASCII_I         C                   x'49'
D ASCII_x         C                   x'78'
D ASCII_y         C                   x'79'
D ASCII_width     C                   X'7769647468'
D ASCII_height    C                   X'686569676874'

 * Note that this is 'java/awt/Rectangle', not 'java.awt.Rectangle'
 * because the JNI uses slash as a separator.
D ASCII_Rectangle...
D                 C                   X'6A6176612F6177742F52656-
D                                     374616E676C65'

 *------------------------------------------------------------------
 * Cancel handling
 *------------------------------------------------------------------
D EnableCanHdlr   PR                  EXTPROC('CEERTX')
D   Handler                       *   CONST PROCPTR
D   CommArea                      *   CONST OPTIONS(*OMIT)
D   Feedback                    12A   OPTIONS(*OMIT)
D CanHdlr         PR
D   CommArea                      *   CONST

 *------------------------------------------------------------------
 * Variables and procedures
 *------------------------------------------------------------------
D rect            s               O   CLASS(*JAVA : Rectangle)
D x               S             10I 0
D y               S             10I 0
D rectClass       S                   LIKE(jclass)
D fieldId         S                   LIKE(jfieldID)
D msg             S             52A
D Cleanup         PR

 *------------------------------------------------------------------
 * Enable the cancel handler to ensure cleanup is done
 *------------------------------------------------------------------
C                   CALLP     EnableCanHdlr (%PADDR(CanHdlr)
C                                          : *OMIT : *OMIT)

 *------------------------------------------------------------------
 * Create a new rectangle with x,y co-ordinates (5, 15),
 * width 100 and height 200.
 *------------------------------------------------------------------
C                   EVAL      rect = NewRectangle (5 : 15 : 100 : 200)

 *------------------------------------------------------------------
 * Prepare to call JNI functions to access the Rectangle's fields
 *------------------------------------------------------------------
```

```
C                   EVAL      JNIEnv_P = getJniEnv ()
C                   EVAL      rectClass = FindClass (JNIEnv_P
C                                                    : ASCII_Rectangle)
 *-----------------------------------------------------------------
 * Call JNI functions to retrieve the Rectangle's width and height
 *-----------------------------------------------------------------
C                   eval      fieldId = GetFieldID (JNIEnv_P
C                                                   : rectClass
C                                                   : ASCII_width
C                                                   : ASCII_I)
C                   eval      width = GetIntField (JNIEnv_P
C                                                   : rect
C                                                   : fieldId)
C                   eval      fieldId = GetFieldID (JNIEnv_P
C                                                   : rectClass
C                                                   : ASCII_height
C                                                   : ASCII_I)
C                   eval      height = GetIntField (JNIEnv_P
C                                                    : rect
C                                                    : fieldId)
C                   eval      msg = 'The rectangle has dimensions ('
C                                 + %trim(%editc(width : '1'))
C                                 + ', '
C                                 + %trim(%editc(height : '1'))
C                                 + ')'
C     msg           dsply

 *-----------------------------------------------------------------
 * Call the Cleanup procedure
 *-----------------------------------------------------------------
C                   callp     Cleanup()
C                   eval      *INLR = '1'

 *-----------------------------------------------------------------
 * Cleanup. * - Free objects if necessary
 *-----------------------------------------------------------------
P Cleanup         B
C                   if        rect <> *NULL and
C                             JNIEnv_P <> *NULL
C                   callp     DeleteLocalRef(JNIEnv_P : rect)
C                   endif
C                   eval      rect = *NULL
C                   eval      JNIEnv_P = *NULL
P Cleanup         E

 *-----------------------------------------------------------------
 * Cancel handler.  Ensures that cleanup is done.
 *-----------------------------------------------------------------
P CanHdlr         B
D CanHdlr         PI
D   CommArea                      *   CONST
C                   callp     Cleanup()
P CanHdlr         E
```

Calling Java Methods Using the JNI Rather than RPG *JAVA Prototypes

The first three parameters are always the same:

1. the JNI environment pointer
2. the object (for instance methods) or the class (for static methods)
3. the method

The method-specific parameters are coded after these three parameters, in one of three different ways. For example, if the method does not return a value (the return type is "void"),

CallVoidMethod:
Choose this way if you are going to call the same method many times, since it makes the method very easy to call. This expects the parameters to be passed normally. To call this JNI function, an RPG programmer would copy the CallVoidMethod prototype from the JNI /COPY file, and code additional parameters. These functions require at least one parameter to be coded with OPTIONS(*NOPASS). If

you don't want to make the method parameters optional, add an extra “dummy” parameter with OPTIONS(*NOPASS). For example, for the method

```
void mymethod (int len, String str);
```

you could code the following prototype for CallVoidMethod:

```
D CallMyMethod    PR                  EXTPROC(*CWIDEN
D                                     : JNINativeInterface.
D                                       CallVoidMethod_P)
D env                                 LIKE(JNIEnv_P) VALUE
D obj                                 LIKE(jobject) VALUE
D methodID                            LIKE(jmethodID) VALUE
D len                                 LIKE(jint) VALUE
D str                                 LIKE(jstring) CONST
D dummy                          1a   OPTIONS (*NOPASS)

…

CallMyMethod (JNIEnv_P : objectId : methodId : 10 : string);
```

Figure 98. Sample RPG Code for Calling CallVoidMethod

CallVoidMethodA:
Choose this way if you do not want to create a separate prototype for calling a method. This expects an array of jvalue structures, with each element of the array holding one parameter. Figure 93 on page 234 above is an example of this.

CallVoidMethodV:
Do not use this in RPG code. It expects a C construct that is extremely awkward to code in RPG.

The actual function to call depends on the type of the return value. For example, if the method returns an integer, you would use CallIntMethodA. To get the class and methodID parameters for these functions, use the FindClass and GetMethodID or GetStaticMethodID.

Note: When calling the JNI directly, the class names must be specified with a slash (/) rather than a period (.) as the separator. For example, use 'java/lang/String' rather than 'java.lang.String'.

Calling RPG programs from Java using PCML

An RPG program or procedure can be called from Java using a Program Call Markup Language (PCML) source file that describes the parameters for the RPG program or procedure. The Java application can use PCML by constructing a ProgramCallDocument object with a reference to the PCML source file.

The ILE RPG compiler will generate PCML information for your ILE RPG program or module when you specify the PGMINFO(*PCML) compiler parameter on your command or Control specification. You can have the PCML information generated into a stream file if you specify the *STMF or *ALL for the Location part of the PGMINFO parameter on the command; you specify the name of the stream file in the INFOSTMF command parameter. You can have the PCML information generated directly into the module if you specify *MODULE or *ALL for the Location part of the PGMINFO parameter on the command, or if you specify the PGMINFO keyword on the Control specification; you can later retrieve the information using the QBNRPII API.

For CRTBNDRPG, PCML is generated based on the contents of the *ENTRY PLIST or the Procedure Interface of the main procedure. For CRTRPGMOD, PCML is also generated based on the Procedure Interfaces of any exported subprocedures (except Java native methods).

When you use CRTRPGMOD, and create a service program, you specify the service program in your Java code using the setPath(String) method of the ProgramCallDocument class. For example:

```
    AS400 as400;
    ProgramCallDocument pcd;
    String path = "/QSYS.LIB/MYLIB.LIB/MYSRVPGM.SRVPGM";
    as400 = new AS400 ();
    pcd = new ProgramCallDocument (as400, "myModule");
    pcd.setPath ("MYFUNCTION", path);
    pcd.setValue ("MYFUNCTION.PARM1", "abc");
    rc = pcd.callProgram("MYFUNCTION");
```

Obtaining mixed-case names in the PCML

If you specify *DCLCASE as one of the parameters of the PGMINFO keyword of your Control specification, the PCML will be generated with the names in the same case as you coded them in your program. For example, for this source, the names in the PCML would be "myFunction", and "parm1", since PGMINFO(*DCLCASE) is specified.

```
      CTL-OPT PGMINFO(*DCLCASE) NOMAIN;
      DCL-PROC myFunction export;
         DCL-PI *N;
            parm1 CHAR(10);
         END-PI;
      END-PROC;
```

The Java code uses the same case for the names:

```
      AS400 as400;
      ProgramCallDocument pcd;
      String path = "/QSYS.LIB/MYLIB.LIB/MYSRVPGM.SRVPGM";
      as400 = new AS400 ();
      pcd = new ProgramCallDocument (as400, "myModule");
      pcd.setPath ("myFunction", path);
      pcd.setValue ("myFunction.parm1", "abc");
      rc = pcd.callProgram("myFunction");
```

PCML Restrictions

The following are restrictions imposed by PCML regarding parameter and return value types.

- The following data types are not supported by PCML:
  - Pointer
  - Procedure Pointer
  - 1-Byte Integer
- Return values and parameters passed by value can only be 4 byte integers (10i 0).
- Varying-length arrays, and data structures containing varying-length subfields are not supported.

  Note: This restriction does not apply when the PCML version is 7.0. For information on how to control the PCML version, see the section on the PGMINFO Control specification keyword in the *IBM Rational Development Studio for i: ILE RPG Reference*.
- When a data structure is used as a parameter for a *ENTRY PLIST, or a prototyped parameter is defined with LIKEDS, some PCML restrictions apply:
  - The data structure may not have any overlapping subfields.
  - The subfields must be coded in order; that is, the start position of each subfield must follow the end position of the previous subfield.
  - If there are gaps between the subfields, the generated PCML for the structure will have subfields named "_unnamed_1", "_unnamed_2" etc, of type "char".
- RPG does not have the concept of output-only parameters. Any parameters that do not have CONST or VALUE coded have a usage of "inputoutput". For inputoutput parameters, the ProgramCallDocument class requires the input values for the parameter to be set before the program can be called. If the parameter is truly an output parameter, you should edit the PCML to change "inputoutput" to "output".

The compile will fail if you generate PCML for a program or module that violates one of the restrictions. The PCML will be generated, but it will contain error messages as comments. For example, if you use a pointer as a parameter, the PCML for that parameter might look like this:

```
<data name="PTR" type="   " length="16" usage="input" />
<!-- Error: unsupported data type -->
```

Chapter 5. Debugging and Exception Handling

This section describes how to:

- Debug an Integrated Language Environment application by using the Integrated Language Environment source debugger
- Write programs that handle exceptions
- Obtain a dump

Debugging Programs

Debugging allows you to detect, diagnose, and eliminate run-time errors in a program. You can debug ILE and OPM programs using the ILE source .

Use the debugger in Rational Developer for i. This is the recommended method and documentation about debugging programs appears in that product's online help. You can debug your program running on the IBM i from a graphical user interface on your workstation. You can also set breakpoints directly in your source before running the debugger. The integrated Rational Developer for i user interface also enables you to control program execution. For example, you can run your program, set line, watch, and service entry point breakpoints, step through program instructions, examine variables, and examine the call stack. You can also debug multiple applications, even if they are written in different languages, from a single window. Each session you debug is listed separately in the Debug view.

This chapter describes how to use the ILE source to:

- Prepare your ILE RPG 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 breakpoints and watch conditions
- Step through a program
- Display and 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 and specify the *NO value of the UPDPROD parameter
- Use the SET debug command in the Display Module Source display and specify UPDPROD NO

See the chapter on debugging in *ILE Concepts* for more information on the ILE source (including authority required to debug a program or service program and the effects of optimization levels).

If you are unfamiliar with using the , follow these steps to create and debug a program. The source for the program PROOF is available in QGPL on all systems.

1. ===> CRTBNDRPG QTEMP/PROOF DBGVIEW(*ALL)
2. ===> STRDBG QTEMP/PROOF
3. Set a breakpoint on one of the calculation lines by putting your cursor on the line and pressing F6

4. Exit the DSPMODSRC screen with F12
5. ===> CALL QTEMP/PROOF

   You will see the source again, with your breakpoint line highlighted .
6. Move your cursor over one of the variables in the program source (Definition, Input, Calculation or Output Specifications) and press F11. The value of the variable will appear at the bottom of the screen
7. Step through the rest of the program by pressing F10, or run to the end with F12
8.

After setting breakpoints, you do not have to call the program directly. You can start an application that will eventually call the program.

If you step through the whole program, it will step through the Input and Output specifications. If you prefer to skip over Input and Output specifications, you can specify OPTION(*NODEBUGIO) in your Header specification or when you compile your program.

More details on these steps will be given in the rest of this chapter.

The ILE Source

The ILE source is used to detect errors in and eliminate errors from program objects and service programs. Using debug commands with any ILE program that contains debug data you can:

- View the program source or change the debug view
- Set and remove breakpoints and watch conditions
- 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 , you must select a debug view when you create a module object or program object using CRTRPGMOD or CRTBNDRPG. After starting the 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 128.

Debug Commands

Many debug commands are available for use with the ILE source . 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 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 job breakpoint at a position in the program being tested. Use BREAK *line-number* WHEN *expression* to enter a conditional job breakpoint.

CLEAR
: Permits you to remove conditional and unconditional breakpoints, or to remove one or all active watch conditions.

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 or WATCH commands. Currently, it does not apply to ILE RPG.

SET
: Allows you to change debug options, such as the ability to update production files, specify if find operations are to be case sensitive, or to enable OPM source debug support.

STEP
: Allows you to run one or more statements of the procedure being debugged.

TBREAK
: Permits you to enter either an unconditional or conditional breakpoint in the current thread at a position in the program being tested.

THREAD
: Allows you to display the Work with Debugged Threads display or change the current thread.

WATCH
: Allows you to request a breakpoint when the contents of a specified storage location is changed from its current value.

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 by the number of columns entered.

RIGHT
: Moves the displayed window of source to the right by the number of columns 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 commands.

The online help for the ILE source 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 .

Follow these steps to access the online help information for ILE RPG:

1. Enter `STRDBG library-name/program-name` where *program-name* is any ILE program with debug data in library *library-name*.
2. Enter `DSPMODSRC` to show the source view if this screen does not appear following step 1.
3. Enter `PF1 (Help)`
4. Put your cursor on EVAL and press enter to bring up the EVAL command help.
5. Put your cursor on `Expressions` and press enter to bring up help for expressions.
6. Put your cursor on `RPG language` and press enter to bring up RPG language examples.
7. From the help panel which appears, you can select a number of topics pertaining to RPG, such as displaying variables, displaying table, and displaying multiple-occurrence data structures.

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, a formatted dump will not list the values of program variables 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.

Note: An OPM program must be compiled with OPTION(*SRCDBG) or OPTION(*LSTDBG) in order to debug it using the ILE source . For more information, see "Starting the ILE Source " on page 247

The debug views are summarized in the following table:

Table 66. Debug Views

Debug View	Debug Data	DBGVIEW Parameter Value
None	No debug data	*NONE
Statement view (default)	No source displayed (use statement numbers in source section of compiler listing)	*STMT
Root source view	Root source member information	*SOURCE
COPY source view	Root source member and /COPY members information	*COPY
Listing view	Compiler listing (dependent on OPTION parameter)	*LIST
All	Data from root source, COPY source, and listing views	*ALL

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 that is produced by the ILE RPG compiler. The information contained in the listing view is dependent on whether OPTION(*SHOWCPY), OPTION(*EXPDDS), and OPTION(*SRCSTMT) are specified for either create command. OPTION(*SHOWCPY) includes /COPY members in the listing; OPTION(*EXPDDS) includes externally described files. OPTION(*SRCSTMT) allows the program object to be debugged using the Statement Numbers instead of the Line Numbers of the compiler listing.

Note: Some information that is available in the compiler listing will not appear on the listing view. For example, if you specify indentation in the compiler listing (via the INDENT parameter), the indentation will not appear in the listing view. If you specify OPTION(*SHOWSKP) in the compiler listing, the skipped statements 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.

You can encrypt the listing view so that the listing information cannot be viewed during a debug session unless the person knows the encryption key. You specify the encryption key using the DBGENCKEY parameter of the CRTBNDRPG, CRTRPGMOD, or CRTSQLRPGI command. You specify the same key when debugging the program to view the text of the listing view.

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 statement numbers and the debug commands. Since the source will not be displayed, you must make use of statement numbers which are shown in the source section of the compiler listing. In other words, to effectively use this view, you will need a compiler listing. In addition, the statement numbers generated for debugging are dependent on whether *SRCSTMT or *NOSRCSTMT is specified for the OPTION parameter. *NOSRCSTMT means that statement numbers are assigned sequentially and are displayed as Line Numbers on the left-most column of the source section of the compiler listing. *SRCSTMT allows you to request that the compiler use SEU sequence numbers and source IDs when generating statement numbers for debugging. The Statement Numbers are shown on the right-most column of the source section of the 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 , 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

Once you have created the debug view (statement, source, COPY, or listing), you can begin debugging your application. To start the ILE source , use the Start Debug (STRDBG) command. Once the is started, it remains active until you enter the End Debug (ENDDBG) command.

Initially you can add as many as 20 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. (Depending on how the OPM programs were compiled and also on the debug environment settings, you may be able to debug them by using the ILE source .) In addition, you can initially add as many as 20 service program objects to a debug session by using the Service Programs (SRVPGM) parameter on the STRDBG command. The rules for debugging a service program are the same as those for debugging a program:

- The program or service program must have debug data.
- You must have *CHANGE authority to a program or service 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.

For an ILE program, the entry module is shown if it has debug data; otherwise, the first module bound to the ILE program with debug data is shown.

For an OPM program, the first program specified on the STRDBG command is shown if it has debug data, and the OPMSRC parameter is *YES. That is, if an OPM program is in a debug session, then you can debug it using the ILE source if the following conditions are met:

1. The OPM program was compiled with OPTION(*LSTDBG) or OPTION(*SRCDBG). (Three OPM languages are supported: RPG, COBOL, and CL. RPG and COBOL programs can be compiled with *LSTDBG or *SRCDBG, but CL programs must be compiled with *SRCDBG.
2. The ILE debug environment is set to accept OPM programs. You can do this by specifying OPMSRC(*YES) on the STRDBG command. (The system default is OPMSRC(*NO).)

If these two conditions are not met, then you must debug the OPM program with the OPM system .

If an OPM program compiled without *LSTDBG or *SRCDBG is specified and a service program is specified, the service program is shown if it has debug data. If there is no debug data, then the DSPMODSRC screen will be empty. If an ILE program and a service program are specified, then the ILE program will be shown.

STRDBG Example

To start a debug session for the sample debug program DEBUGEX and a called OPM program RPGPGM, type:

```
STRDBG PGM(MYLIB/DEBUGEX MYLIB/RPGPGM) OPMSRC(*YES)
```

The Display Module Source display appears as shown in Figure 99 on page 248. DEBUGEX consists of two modules, an RPG module DBGEX and a C module cproc. See "Sample Source for Debug Examples" on page 286 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.

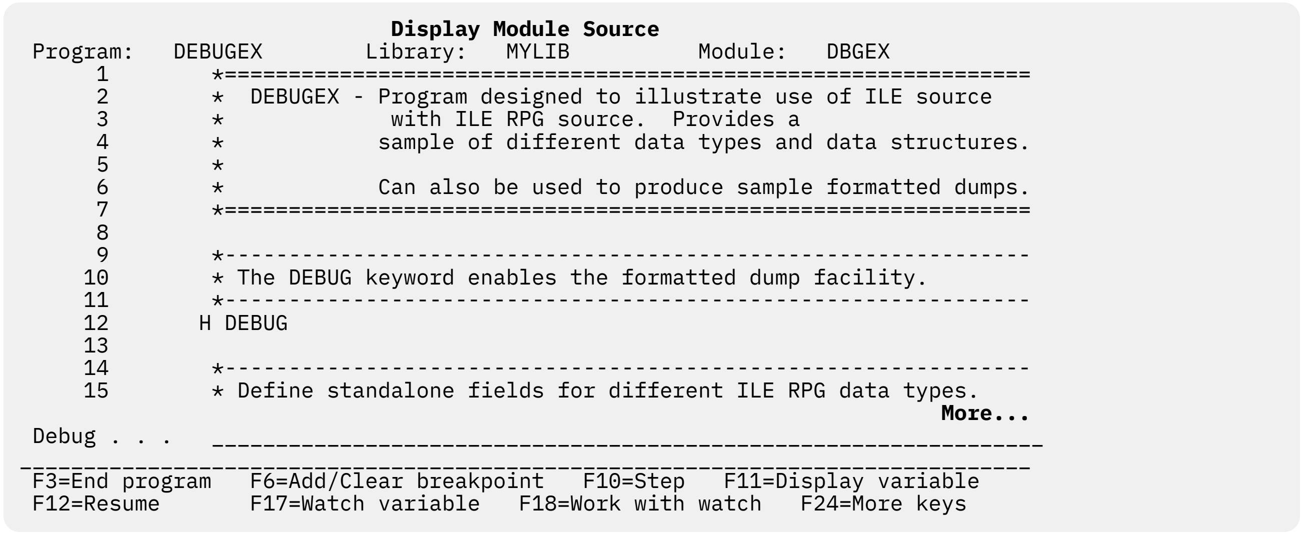

```
                              Display Module Source
 Program:   DEBUGEX        Library:   MYLIB          Module:   DBGEX
      1        *===============================================================
      2        *  DEBUGEX - Program designed to illustrate use of ILE source
      3        *             with ILE RPG 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 data types.
                                                                        More...
 Debug . . .   _________________________________________________________________
_______________________________________________________________________________
 F3=End program   F6=Add/Clear breakpoint   F10=Step   F11=Display variable
 F12=Resume       F17=Watch variable   F18=Work with watch   F24=More keys
```

Figure 99. Display Module Source display for program DEBUGEX

Note: Up to 20 service programs can initially be added to the debug session by using the Service Program (SRVPGM) parameter on the STRDBG command. You can also 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 add it as part of a STEP INTO debug command.

Setting Debug Options

After you start a debug session, you can set or change the following debug options:

- Whether database files can be updated while debugging your program. (This option corresponds to the UPDPROD parameter of the STRDBG command.)
- Whether text searches using FIND are case-sensitive.
- Whether OPM programs are to be debugged using the ILE source . (This option corresponds to the OPMSRC parameter.)

Changing the debug options using the SET debug command affects the value for the corresponding parameter, if any, specified on the STRDBG command. You can also use the Change Debug (CHGDBG) command to set debug options. However, the OPMSRC option can not be changed by the CHGDBG command. OPMSRC can only be changed by the debug SET command.

Suppose you are in a debug session working with an ILE program and you decide you should also debug an OPM program that has debug data available. To enable the ILE source to accept OPM programs, 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. Type

   ```
   SET
   ```

3. The Set Debug Options display appears. On this display type Y (Yes) for the *OPM source debug support* field, and press Enter to return to the Display Module Source display.

You can now add the OPM program, either by using the Work with Module display, or by processing a call statement to that program.

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 have two choices depending on the value specified for OPMSRC. If you specified OPMSRC(*YES), by using either STRDBG, the SET debug command, or CHGDBG, then you add or remove an OPM program using the Work With Module Display. (Note that there will not be a module name listed for an OPM program.) There is no limit to the number of OPM programs that can be included in a debug session when OPMSRC(*YES) is specified.

If you specified OPMSRC(*NO), then you must use the Add Program (ADDPGM) command or the Remove Program (RMVPGM) command. Only 20 OPM programs can be in a debug session at one time when OPMSRC(*NO) is specified.

Note: You cannot debug an OPM program with debug data from both an ILE and an OPM debug session. If OPM program is already in an OPM debug session, you must first remove it from that session before adding it to the ILE debug session or stepping into it from a call statement. Similarly, if you want to debug it from an OPM debug session, you must first remove it from an ILE debug session.

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 132 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 100 on page 250.
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
         DEBUGEX            MYLIB          *PGM
           DBGEX                           *MODULE     Selected
           CPROC                           *MODULE
                                                                    Bottom
Command
===> ________________________________________________________________________
F3=Exit   F4=Prompt   F5=Refresh   F9=Retrieve   F12=Cancel
```

Figure 100. 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 101 on page 250.
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
         DEBUGEX            MYLIB          *PGM
           DBGEX                           *MODULE     Selected
           CPROC                           *MODULE
                                                                    Bottom
Command
===> ________________________________________________________________________
F3=Exit   F4=Prompt   F5=Refresh   F9=Retrieve   F12=Cancel
```

Figure 101. Removing an ILE Program from a Debug Session

Viewing the Program Source

The Display Module Source display shows the source of an ILE program object one module object at a time. The source of an ILE module object 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)

The source of an OPM program can be shown if the following conditions are met:

1. The OPM program was compiled with OPTION(*LSTDBG) or OPTION(*SRCDBG). (Only RPG and COBOL programs can be compiled with *LSTDBG.)
2. The ILE debug environment is set to accept OPM programs; that is the value of OPMSRC is *YES. (The system default is OPMSRC(*NO).)

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 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 an ILE 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. If you use this option with an OPM program object, then the source or listing view is shown (if available).

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 99 on page 248 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 102 on page 251 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 103 on page 252. 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
         DEBUGEX           MYLIB          *PGM
           DBGEX                          *MODULE     Selected
 5         CPROC                          *MODULE
                                                                        Bottom
Command
===> ________________________________________________________________________
F3=Exit   F4=Prompt   F5=Refresh   F9=Retrieve   F12=Cancel
```

Figure 102. Changing to a Different Module

```
                             Display Module Source
Program:   DEBUGEX        Library:   MYLIB          Module:   CPROC
     1       #include <stdlib.h>
     2       #include <string.h>
     3       #include <stdio.h>
     4       extern char EXPORTFLD[6];
     5
     6       char *c_proc(unsigned int size, char *inzval)
     7       {
     8          char *ptr;
     9          ptr = malloc(size);
    10          memset(ptr, *inzval, size );
    11          printf("import string: %6s.\n",EXPORTFLD);
    12          return(ptr);
    13       }
                                                                       Bottom
Debug . . .   _________________________________________________________________
_______________________________________________________________________________
F3=End program   F6=Add/Clear breakpoint   F10=Step   F11=Display variable
F12=Resume       F17=Watch variable   F18=Work with watch   F24=More keys
```

Figure 103. Source View of ILE C procedure cproc

Changing the View of a Module

Several different views of an ILE RPG 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 104 on page 252. 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 104 on page 252.

```
                             Display Module Source
 ..............................................................................
 :                                Select View                                 :
 :                                                                            :
 :  Current View . . . :     ILE RPG Copy View                                :
 :                                                                            :
 :  Type option, press Enter.                                                 :
 :    1=Select                                                                :
 :                                                                            :
 :  Opt     View                                                              :
 :   1        ILE RPG Listing View                                            :
 :            ILE RPG Source View                                             :
 :            ILE RPG Copy View                                               :
 :                                                                            :
 :                                                                    Bottom  :
 :  F12=Cancel                                                                :
 :                                                                            :
 :............................................................................:
                                                                       More...
Debug . . .   _________________________________________________________________
_______________________________________________________________________________
F3=End program   F6=Add/Clear breakpoint   F10=Step   F11=Display variable
F12=Resume       F17=Watch variable   F18=Work with watch   F24=More keys
```

Figure 104. 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.

There are two types of breakpoints: job and thread. Each **thread** in a threaded application may have it's own thread breakpoint at the same position at the same time. Both job and thread breakpoints can be unconditional or conditional. In general, there is one set of debug commands and Function keys for job breakpoints and another for thread breakpoints. For the rest of this section on breakpoints, the word breakpoint refers to both job and thread, unless specifically mentioned otherwise.

Note: Breakpoints are automatically generated for input and output specifications if the default OPTION(*DEBUGIO) is specified. If you do not want to generate breakpoints, specify OPTION(*NODEBUGIO).

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
  - You can either have a job or thread breakpoint on a specified position at the same time, but not both.

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 .

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 Job Breakpoints Using Statement Numbers" on page 259.

Setting and Removing Unconditional Job Breakpoints

You can set or remove an unconditional Job breakpoint by using:

- F6 (Add/Clear breakpoint) or F13 (Work with module breakpoints) from the Display Module Source display
- The BREAK debug command to set a job breakpoint
- The CLEAR debug command to remove a jobbreakpoint
- The Work with Module Breakpoints display.

The simplest way to set and remove an unconditional job 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 job breakpoint using F13 (Work with module breakpoints), press F13 (Work with module breakpoints) from the Display Module Source display. A list of options appear which allow you to set or remove breakpoints. If you select 4 (Clear), a job breakpoint is removed from the line.

An alternate method of setting and removing unconditional job breakpoints is to use the BREAK and CLEAR debug commands. To set an unconditional job 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 job 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. When a job breakpoint is cleared, it is also cleared for all threads.

Example of Setting an Unconditional Job Breakpoint

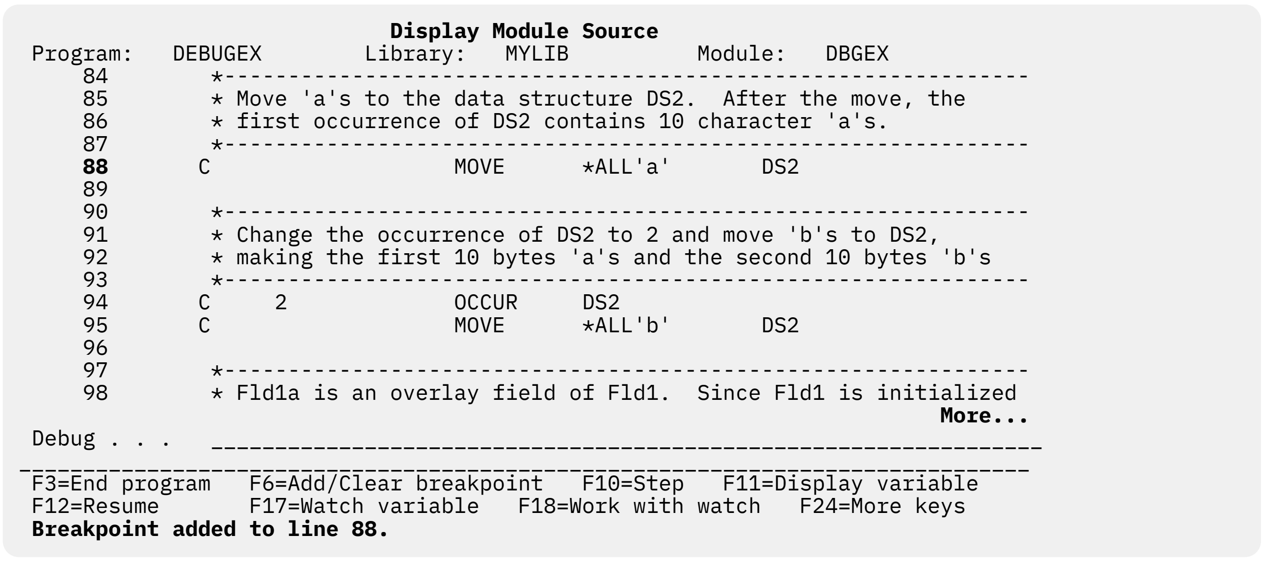
In this example you set an unconditional job breakpoint using F6 (Add/Clear breakpoint). The breakpoint is to be set on the first runnable Calculation specification so that the various fields and data structures can be displayed.

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 job breakpoint in the module shown, continue with step "3" on page 254. If you want to set a job 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 88.
4. Press F6 (Add/Clear breakpoint). If there is no breakpoint on the line 88, then an unconditional breakpoint is set on that line, as shown in Figure 105 on page 255. 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 88, since it is the first runnable statement.



```
                            Display Module Source
 Program:   DEBUGEX        Library:   MYLIB          Module:   DBGEX
     84        *---------------------------------------------------------------
     85        * Move 'a's to the data structure DS2.  After the move, the
     86        * first occurrence of DS2 contains 10 character 'a's.
     87        *---------------------------------------------------------------
     88       C                   MOVE      *ALL'a'       DS2
     89
     90        *---------------------------------------------------------------
     91        * Change the occurrence of DS2 to 2 and move 'b's to DS2,
     92        * making the first 10 bytes 'a's and the second 10 bytes 'b's
     93        *---------------------------------------------------------------
     94       C     2             OCCUR     DS2
     95       C                   MOVE      *ALL'b'       DS2
     96
     97        *---------------------------------------------------------------
     98        * 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       F17=Watch variable   F18=Work with watch   F24=More keys
 Breakpoint added to line 88.
```

Figure 105. Setting an Unconditional Job 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 Unconditional Thread Breakpoints

You can set or remove an unconditional thread breakpoint by using:

- The Work with Module Breakpoints display
- The TBREAK debug command to set a thread breakpoint in the current thread
- The CLEAR debug command to remove a thread breakpoint

To set an unconditional thread breakpoint using the Work with Module Breakpoints display:

- Type 1 (Add) in the *Opt* field.
- In the *Thread* field, type the thread identifier.
- Fill in the remaining fields as if it were an unconditional job breakpoint.
- Press Enter.

Note: The *Thread* field is displayed when the DEBUG option on the SPAWN command is greater than or equal to one.

The TBREAK debug command has the same syntax as the BREAK debug command. Where the BREAK debug command sets a job breakpoint at the same position in all threads, the TBREAK debug command sets a thread breakpoint in a single thread — the current thread.

The **current thread** is the thread that is currently being debugged. Debug commands are issued to this thread. When a debug stop occurs, such as a breakpoint, the current thread is set to the thread where the debug stop happened. The debug THREAD command and the 'Work with Debugged Threads' display can be used to change the current thread.

To remove an unconditional thread breakpoint use the CLEAR debug command. When a thread breakpoint is cleared, it is cleared for the current thread only.

Setting and Removing Conditional Job Breakpoints

You can set or remove a conditional job breakpoint by using:

- The Work with Module Breakpoints display

- The BREAK debug command to set a job 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 job 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 job breakpoints. An example of the display is shown in Figure 106 on page 257.

To make the job breakpoint conditional, specify a conditional expression in the *Condition* field. If the line on which you want to set a job breakpoint is not a runnable statement, the breakpoint will be set at the next runnable statement.

If a thread column is shown, before pressing Enter, type *JOB in the *Thread* field.

Once you have finished specifying all of the job 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 job breakpoint is reached, the conditional expression associated with the job 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 258 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 Job Breakpoint Using F13

In this example you set a conditional job breakpoint using F13 (Work with module breakpoints).

1. To set a conditional job 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 127 when *IN02='1', type 127 for the *Line* field, *IN02='1' for the *Condition* field.
4. If a thread column is shown, before pressing Enter, type *JOB in the thread field.

   Figure 106 on page 257 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
        127     *in02='1'
        88
        102
                                                                       Bottom
Command
===> ________________________________________________________________________
F3=Exit   F4=Prompt   F5=Refresh   F9=Retrieve   F12=Cancel
Breakpoint added to line 127.
```

Figure 106. Setting a Conditional Job Breakpoint

A conditional job breakpoint is set on line 127. 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.

5. 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 . Your 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. At this point you can step through the program or resume processing.

Example of Setting a Conditional Job 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 job breakpoint using the BREAK command:

1. From the Display Module Source display, enter:

   ```
   break 128 when BigDate='1994-09-30'
   ```

   A conditional job breakpoint is set on line 128.
2. After the breakpoint is set, press F3 (End Program) to leave the ILE source . 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
    122
    123        *---------------------------------------------------------------
    124        * After the following SETON operation, *IN02 = '1'.
    125        *---------------------------------------------------------------
    126       C                   SETON
    127       C                   IF        *IN02
    128       C                   MOVE      '1994-09-30'  BigDate
    129       C                   ENDIF
    130
    131        *---------------------------------------------------------------
    132        * Put a new value in the second cell of Arry.
    133        *---------------------------------------------------------------
    134       C                   MOVE      4             Arry
    135
    136        *---------------------------------------------------------------
                                                                        More...
 Debug . . .   break 128 when BigDate='1994-09-30'______________________________
________________________________________________________________________________
 F3=End program   F6=Add/Clear breakpoint   F10=Step   F11=Display variable
 F12=Resume       F17=Watch variable   F18=Work with watch   F24=More keys
```

Figure 107. Setting a Conditional Job 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 67 on page 259 for the possible combinations of non-numeric conditional breakpoint expressions.

The sort sequence table used by the source 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 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 handles comparisons; all characters, including the shift-out/shift-in characters, are assigned weights.

Note:

1. The alternate sequence specified by ALTSEQ (*SRC) on the Control specification is not available to the ILE source . Instead the source uses the *HEX sort sequence table.
2. The name of the sort sequence table is saved during compilation. At debug time, the source 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 uses the *HEX sort sequence table.

Table 67. 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]

Note:

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 Job Breakpoints Using Statement Numbers

You set and remove conditional or unconditional job 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 job 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. Since ILE RPG allows more than one procedure per module, the *procedure-name* can be either the name of the main procedure or one of the subprocedures in a module. The variable *statement-number* is the statement number from the compiler listing on which you want to set a breakpoint.

Note: The statement number in the source listing is labeled as the Line Number when OPTION(*NOSRCSTMT) is specified, and as the Statement Number when OPTION(*SRCSTMT) is specified. For example, Figure 108 on page 260 shows a sample section of a listing with OPTION(*NOSRCSTMT). Figure 109 on page 260 shows the same section with OPTION(*SRCSTMT).

```
Line   <--------------------- Source Specifications ----------------------------------------------><---- Comments ----> Src Seq
Number ....1....+....2....+<-------- 26 - 35 -------->....4....+....5....+....6....+....7....+....8....+....9....+...10 Id  Number
                          S o u r c e   L i s t i n g
     1 C                   MOVE                        '123'         BI_FLD1                                              000100
     2 C                   SETON                                                          LR----                          000200
       * * * * *   E N D   O F   S O U R C E   * * * * *
```

*Figure 108. Sample Section of the Listing with OPTION(*NOSRCSTMT)*

```
Seq    <--------------------- Source Specifications ----------------------------------------------><---- Comments ----> Statement
Number ....1....+....2....+<-------- 26 - 35 -------->....4....+....5....+....6....+....7....+....8....+....9....+...10 Number
                          S o u r c e   L i s t i n g
000100 C                   MOVE                        '123'         BI_FLD1                                              000100
000200 C                   SETON                                                          LR----                          000200
       * * * * *   E N D   O F   S O U R C E   * * * * *
```

*Figure 109. Sample Section of the Compiler Listing with OPTION(*SRCSTMT)*

In this example, a Statement View is used to set a breakpoint for the procedure TEST. To set a breakpoint for the module with the *NOSRCSTMT listing, type:

```
BREAK TEST/2
```

To set a breakpoint for the module with the *SRCSTMT listing, type:

```
BREAK TEST/200
```

In both cases, the breakpoint is set on the '`SETON            LR----`' line.

```
                             Display Module Source
 Program:   TEST           Library:   MYLIB          Module:   TEST
   (Source not available.)
                                                                       Bottom
 Debug . . .   break TEST/2________________________________________________________
________________________________________________________________________________
 F3=End program   F6=Add/Clear breakpoint   F10=Step   F11=Display variable
 F12=Resume       F17=Watch variable   F18=Work with watch   F24=More keys
 Breakpoint added to statement 2 of procedure TEST.
```

Figure 110. Setting a Breakpoint Using Statement View

For all other debug views, the statement numbers can be used in addition to the program *line-numbers* in the . For example, to set a breakpoint at the beginning of subprocedure FmtCust in the Listing View below, type:

```
BREAK 34
```

Or

```
BREAK FmtCust/2600
```

In both cases, the breakpoint is set on the '`P  FmtCust           B`' line.

```
                            Display Module Source
 Program:   MYPGM          Library:   MYLIB          Module:   MYPGM
     33     002500  * Begin-procedure
     34     002600 P FmtCust         B
     35     002700 D FmtCust         PI            25A
     36     002800  * Procedure-interface (same as the prototype)
     37     002900 D    FirstName                  10A
     38     003000 D    LastName                   15A
     39     003100 D    ValidRec                     N
     40     003200  * Calculations
     41     003300 C                   IF        ValidRec = '0'
     42     003400 C                   RETURN    %TRIMR(FirstName) + ' ' + Last
     43     003500 C                   ENDIF
     44     003600 C                   RETURN    'Last Customer'
     45     003700  * End-procedure
     46     003800 P                 E
     47           *MAIN PROCEDURE EXIT
                                                                       More...
 Debug . . .   BREAK fmtcust/2600____________________________________________
________________________________________________________________________________
 F3=End program   F6=Add/Clear breakpoint   F10=Step   F11=Display variable
 F12=Resume       F17=Watch variable   F18=Work with watch   F24=More keys
 Breakpoint added to line 34.
```

*Figure 111. Setting a Breakpoint using Statement Numbers and a Listing View with OPTION(*SRCSTMT)*

To set a conditional job 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 procedure-name/statement-number
```

on the debug command line.

Setting and Removing Conditional Thread Breakpoints

You can set or remove a conditional thread breakpoint by using:

- The Work with Module Breakpoints display
- The TBREAK debug command to set a conditional thread breakpoint in the current thread
- The CLEAR debug command to remove a conditional thread breakpoint.

Using the Work with Module Breakpoints Display

To set a conditional thread breakpoint using the Work with Module Breakpoints display:

1. Type 1 (Add) in the *Opt* field.
2. In the *Thread* field, type the thread identifier.
3. Fill in the remaining fields as if it were a conditional job breakpoint.
4. Press Enter.

To remove a conditional thread breakpoint using the Work with Module Breakpoints display:

1. Type 4 (Clear) in the *Opt* field next to the breakpoint you want to remove.
2. Press Enter.

Using the TBREAK or CLEAR Debug Commands

You use the same syntax for the TBREAK debug command as you would for the BREAK debug command. The difference between these commands is that the BREAK debug command sets a conditional job breakpoint at the same position in all threads, while the TBREAK debug command sets a conditional thread breakpoint in the current thread.

To remove a conditional thread breakpoint, use the CLEAR debug command. When a conditional thread breakpoint is removed, it is removed for the current thread only.

Removing All Job and Thread Breakpoints

You can remove all job and thread 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.

Setting and Removing Watch Conditions

You use a **watch condition** to monitor if the current value of an expression or a variable changes while your program runs. Setting watch conditions is similar to setting conditional breakpoints, with one important difference:

- Watch conditions stop the program as soon as the value of a watched expression or variable changes from its current value.
- Conditional job breakpoints stop the program only if a variable changes to the value specified in the condition.

The watches an expression or a variable through the contents of a **storage address**, computed at the time the watch condition is set. When the content at the storage address is changed from the value it had when the watch condition was set or when the last watch condition occurred, the program stops.

Note: After a watch condition has been registered, the new contents at the watched storage location are saved as the new current value of the corresponding expression or variable. The next watch condition will be registered if the new contents at the watched storage location change subsequently.

Characteristics of Watches

You should know the following characteristics about watches before working with them:

- Watches are monitored system-wide, with a maximum number of 256 watches that can be active simultaneously. This number includes watches set by the system.

  Depending on overall system use, you may be limited in the number of watch conditions you can set at a given time. If you try to set a watch condition while the maximum number of active watches across the system is exceeded, you receive an error message and the watch condition is not set.

  Note: If an expression or a variable crosses a page boundary, two watches are used internally to monitor the storage locations. Therefore, the maximum number of expressions or variables that can be watched simultaneously system-wide ranges from 128 to 256.
- Watch conditions can only be set when a program is stopped under debug, and the expression or variable to be watched is in scope. If this is not the case, an error message is issued when a watch is requested, indicating that the corresponding call stack entry does not exist.
- Once the watch condition is set, the address of a storage location watched does not change. Therefore, if a watch is set on a temporary location, it could result in spurious watch-condition notifications.

  An example of this is the automatic storage of an ILE RPG subprocedure, which can be re-used after the subprocedure ends.

  A watch condition may be registered although the watched variable is no longer in scope. You must not assume that a variable is in scope just because a watch condition has been reported.
- Two watch locations in the same job must not overlap in any way. Two watch locations in different jobs must not start at the same storage address; otherwise, overlap is allowed. If these restrictions are violated, an error message is issued.

Note: Changes made to a watched storage location are ignored if they are made by a job other than the one that set the watch condition.

- After the command is successfully run, your application is stopped if a program in your session changes the contents of the watched storage location, and the **Display Module Source** display is shown.

  If the program has debug data, and a source text view is available, it will be shown. The source line of the statement that was about to be run when the content change at the storage-location was detected is highlighted. A message indicates which watch condition was satisfied.

  If the program cannot be debugged, the text area of the display will be blank.
- Eligible programs are automatically added to the debug session if they cause the watch-stop condition.
- When multiple watch conditions are hit on the same program statement, only the first one will be reported.
- You can set watch conditions also when you are using service jobs for debugging, that is when you debug one job from another job.

Setting Watch Conditions

Before you can set a watch condition, your program must be stopped under debug, and the expression or variable you want to watch must be in scope:

- To watch a global variable, you must ensure that the program in which the variable is defined is active before setting the watch condition.
- To watch a local variable, you must step into the procedure in which the variable is defined before setting the watch condition.

You can set a watch condition by using:

- F17 (Watch Variable) to set a watch condition for a variable on which the cursor is positioned.
- The WATCH debug command with or without its parameters.

Using the WATCH Command

If you use the WATCH command, it must be entered as a single command; no other debug commands are allowed on the same command line.

- To access the **Work With Watch** display shown below, type:

  ```
  WATCH
  ```

  on the debug command line, without any parameters.

```
                                 Work with Watch
                                                     System:
 Type options, press Enter.
   4=Clear 5=Display
 Opt    Num      Variable                  Address            Length
  -     1        SALARY                    080090506F027004   4
                                                               Bottom
Command
===>________________________________________________________________________
F3=Exit  F4=Prompt  F5=Refresh   F9=Retrieve   F12=Cancel
```

Figure 112. Example of a Work with Watch Display

The **Work with Watch** display shows all watches currently active in the debug session. You can clear, and display watches from this display. When you select Option 5 `Display`, the **Display Watch** window shown below displays information about the currently active watch.

```
                          Work with Watch
  ..........................................................
  :                      Display Watch                     :
  :                                                        :
  :  Watch Number ....:   1                                :
  :  Address .........:   080090506F027004                 :
  :  Length ..........:   4                                :
  :  Number of Hits ..:   0                                :
  :                                                        :
  :  Scope when watch was set:                             :
  :    Program/Library/Type:    PAYROLL     ABC    *PGM    :
  :                                                        :
  :    Module...:    PAYROLL                               :
  :    Procedure:    PAYROLL                               :
  :    Variable.:    SALARY                                :
  :                                                        :
  :  F12=Cancel                                            :
  :                                                        :
  ..........................................................
                                                                Bottom
 Command
 ===>________________________________________________________________________
 F3=Exit  F4=Prompt  F5=Refresh   F9=Retrieve   F12=Cancel
```

Figure 113. Example of a Display Watch Window

- To specify a variable or expression to be watched, type:

  ```
  WATCH expression
  ```

  on the debug command line.

  This command requests a breakpoint to be set if the value of `expression` is changed from its current value.

  Note: `expression` is used to determine the address of the storage location to watch and must resolve to a location that can be assigned to, for example:

  ```
  %SUBSTR(X 1 5)
  ```

  The scope of the expression variables in a watch is defined by the most recently issued QUAL command.

- To set a watch condition and specify a watch length, type:

  ```
  WATCH expression : watch length
  ```

  on a debug command line.

  Each watch allows you to monitor and compare a maximum of 128 bytes of contiguous storage. If the maximum length of 128 bytes is exceeded, the watch condition will not be set, and the issues an error message.

  By default, the length of the expression type is also the length of the watch-comparison operation. The **watch-length parameter** overrides this default. It determines the number of bytes of an expression that should be compared to determine if a change in value has occurred.

  For example, if a 4-byte integer is specified as the variable, without the watch-length parameter, the comparison length is four bytes. However, if the watch-length parameter is specified, it overrides the length of the expression in determining the watch length.

Displaying Active Watches

To display a system-wide list of active watches and show which job set them, type:

```
DSPDBGWCH
```

on a debug command line. This command brings up the **Display Debug Watches** display shown below.

```
                              Display Debug Watches
                                                              System:
  ------------Job---------------        NUM     LENGTH     ADDRESS
  MYJOBNAME1  MYUSERPRF1  123456          1          5     080090506F027004
  JOB4567890  PRF4567890  222222          1          8     09849403845A2C32
  JOB4567890  PRF4567890  222222          2          2     098494038456AA00
  JOB         PROFILE     333333         14          4     040689578309AF09
  SOMEJOB     SOMEPROFIL  444444          3          4     005498348048242A
Bottom
 Press Enter to continue
 F3=Exit  F5=Refresh   F12=Cancel
```

Figure 114. Example of a Display Debug Watch Display

Note: This display does not show watch conditions set by the system.

Removing Watch Conditions

Watches can be removed in the following ways:

- The CLEAR command used with the WATCH keyword selectively ends one or all watches. For example, to clear the watch identified by `watch-number`, type:

  ```
  CLEAR WATCH watch-number
  ```

  The watch number can be obtained from the **Work With Watches** display.

  To clear all watches for your session, type:

  ```
  CLEAR WATCH ALL
  ```

  on a debug command line.

  Note: While the CLEAR PGM command removes all breakpoints in the program that contains the module being displayed, it has no effect on watches. You must explicitly use the WATCH keyword with the CLEAR command to remove watch conditions.
- The CL End Debug (ENDDBG) command removes watches set in the local job or in a service job.

  Note: ENDDBG will be called automatically in abnormal situations to ensure that all affected watches are removed.
- The initial program load (IPL) of your IBM i removes all watch conditions system-wide.

Example of Setting a Watch Condition

In this example, you watch a variable SALARY in program `MYLIB/PAYROLL`. To set the watch condition, type:

```
WATCH SALARY
```

on a debug line, accepting the default value for the watch-length.

If the value of the variable SALARY changes subsequently, the application stops and the **Display Module Source** display is shown, as illustrated in Figure 115 on page 266.

```
                                Display Module Source
Program:   PAYROL           Library:   MYLIB       Module:  PAYROLL
    52  C                   eval      cnt = 1
    53  C                   dow       (cnt < EMPMAX)
    54  C                   eval      Pay_exmpt(cnt) = eflag(cnt)
    55  C                   eval      cnt = cnt + 1
    56  C                   enddo
    57  C
    58  C                   eval      index = 1
    59  C                   dow       index <= cnt
    60  C                   if        Pay_exmpt(index) = 1
    61  C                   eval      SALARY = 40 * Pay_wage(index)
    62  C                   eval      numexmpt = numexmpt + 1
    63  C                   else
    64  C                   eval      SALARY = Pay_hours(index)*Pay_wage(index)
    65  C                   endif
    66  C                   eval      index = index + 1
    67  C                   enddo
                                                                More...
Debug . . .   _________________________________________________________________
_______________________________________________________________________________
F3=End program  F6=Add/Clear breakpoint  F10=Step  F11=Display variable
F12=Resume       F17=Watch variable   F18=Work with watch   F24=More keys
Watch number  1 at line 65, variable: SALARY
```

Figure 115. Example of Message Stating WATCH was Successfully Set

- The line number of the statement where the change to the watch variable was detected is highlighted. This is typically the first executable line *following* the statement that changed the variable.
- A message indicates that the watch condition was satisfied.

  Note: If a text view is not available, a blank **Display Module Source** display is shown, with the same message as above in the message area.

  The following programs cannot be added to the ILE debug environment:

  1. ILE programs without debug data
  2. OPM programs with non-source debug data only
  3. OPM programs without debug data

In the first two cases, the stopped statement number is passed. In the third case, the stopped MI instruction is passed. The information is displayed at the bottom of a blank **Display Module Source** display as shown below. Instead of the line number, the statement or the instruction number is given.

```
                                Display Module Source
(Source not available)
F3=End program  F12=Resume  F14=Work with module list  F18=Work with watch
F21=Command entry  F22=Step into  F23=Display output
Watch number  1 at instruction 18,  variable: SALARY
```

Figure 116. Example of a Display Module Source Panel

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 . 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.

Breakpoints can be set before the program is called and while you are stepping through the program. Breakpoints can also be automatically generated for input and output specifications if the default OPTION(*DEBUGIO) is specified. If this option is selected, a STEP on a READ statement will stop at the input specification. You can choose not to generate breakpoints for input and output specifications with OPTION(*NODEBUGIO).

You can step into an OPM program if it has debug data available and if the debug session accepts OPM programs for debugging.

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 call statement, or
- Step into the call statement.

A call statement for ILE RPG includes any of the following operations:

- CALL
- CALLB
- CALLP
- Any operation where there is an expression in the extended-factor 2 field, and the expression contains a call to a procedure.

If you choose to **step over** the call statement, then you will stay inside the current procedure. The call statement is processed as a single step and the cursor moves to the next step after the call. Step over is the default step mode.

If you choose to **step into** the call statement, then each statement inside the call statement is run as a single step. Depending on the number of steps specified, the step command may end inside the call statement, in which case the source for the call statement is shown in the Display Module Source display.

Note: You cannot step over or step into RPG subroutines. You can, however, step over and into subprocedures.

Stepping Over Call Statements

You can step over call statements 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 call statement in a debug session. If the call statement to be run is a CALL operation 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. Similarly, if the call statement is an EVAL operation where a procedure is called in the expression, then the complete EVAL operation is performed, including the call to the procedure, before the calling program or procedure is stopped again.

Alternately, you can use the STEP OVER debug command to step over a call statement 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 that you want to run in the next step before processing is halted again. If this variable is omitted, the default is 1.

Stepping Into Call Statements

You can step into a call statement 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 or procedure in a debug session. If the next statement to be run is a call statement to another program or procedure, then pressing F22 (Step into) will cause the first runnable statement in the called program or procedure to be run. The called program or procedure will then be shown in the Display Module Source display.

Note: The called program or procedure 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 call statement 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 that you want to run in the next step before processing is halted again. If this variable is omitted, the default is 1.

If one of the statements that are run contains a call statement the will step into the called program or procedure. Each statement in the called program or procedure will be counted in the step. If the step ends in the called program or procedure, then the called program or procedure 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 operation 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.

In the example of DEBUGEX, if you enter STEP INTO (or press F22) while on the EVAL operation that calls the procedure c_proc, then you would step into the C module.

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 , 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.

TIP

In order to display data immediately before or after a subprocedure is run, place breakpoints on the procedure specifications that begin and end the subprocedure.

Example of Stepping Into an OPM Program Using F22

In this example, you use the F22 (Step Into) to step into the OPM 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 102, which is the last runnable statement before the CALL operation, type `Break 102` and press Enter.
3. Press F3 (End program) to leave the Display Module Source display.
4. Call the program. The program stops at breakpoint 102, as shown in Figure 117 on page 269.

```
                            Display Module Source
 Program:   DEBUGEX        Library:   MYLIB          Module:   DBGEX
     98        * Fld1a is an overlay field of Fld1.  Since Fld1 is initialized
     99        * to 'ABCDE', the value of Fld1a(1) is 'A'.  After the
    100        * following MOVE operation, the value of Fld1a(1) is '1'.
    101        *---------------------------------------------------------------
    102       C                   MOVE      '1'           Fld1a(1)
    103
    104        *---------------------------------------------------------------
    105        * Call the program RPGPGM, which is a separate program object.
    106        *---------------------------------------------------------------
    107       C     Plist1        PLIST
    108       C                   PARM                    PARM1
    109       C                   CALL      'RPGPGM'      Plist1
    110
    111        *---------------------------------------------------------------
    112        * Call c_proc, which imports ExportFld from the main procedure.
                                                                        More...
 Debug . . .   _________________________________________________________________
_______________________________________________________________________________
 F3=End program   F6=Add/Clear breakpoint   F10=Step   F11=Display variable
 F12=Resume       F17=Watch variable   F18=Work with watch   F24=More keys
 Breakpoint at line 102.
```

Figure 117. 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 118 on page 270.

   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
     1        *===============================================================
     2        *  RPGPGM - Program called by DEBUGEX to illustrate the STEP
     3        *           functions of the ILE source .
     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       F17=Watch variable   F18=Work with watch   F24=More keys
Step completed at line 13.
```

Figure 118. Stepping into RPGPGM

If the ILE source is not set to accept OPM programs, or if there is no debug data available, then you will see a blank Display Module Source display with a message indicating that the source is not available. (An OPM program has debug data if it was compiled with OPTION(*SRCDBG) or OPTION(*LSTDBG).)

Example of Stepping Into a Subprocedure

In this example, you use the F22 (Step Into) to step into the subprocedure Switch, which is in the module DEBUGEX.

1. Ensure that the Display Module Source display shows the source for DBGEX.
2. To set an unconditional breakpoint at line 120, which is the last runnable statement before the CALLP operation, type `Break 120` and press Enter.
3. Press F3 (End program) to leave the Display Module Source display.
4. Call the program. The program stops at breakpoint 119.
5. Press F22 (Step into). The call statement is run and then the display moves to the subprocedure, as in Figure 119 on page 270. The first runnable statement of RPGPGM is processed (line 13) and then processing stops.

```
                            Display Module Source
Program:   DEBUGEX        Library:   MYLIB         Module:   DBGEX
   141
   142        *============================================================
   143        * Define the subprocedure Switch.
   144        *============================================================
   145       P Switch          B
   146       D Switch          PI
   147       D   Parm                        1A
   148        *-------------------------------------------------------
   149        * Define a local variable for debugging purposes.
   150        *-------------------------------------------------------
   151       D Local           S             5A      INZ('aaaaa')
   152
   153       C                   IF        Parm = '1'
   154       C                   EVAL      Parm = '0'
   155       C                   ELSE
Debug . . .   _________________________________________________________________
_______________________________________________________________________________
F3=End program   F6=Add/Clear breakpoint   F10=Step   F11=Display variable
F12=Resume       F17=Watch variable   F18=Work with watch   F24=More keys
Step completed at line 145.
```

Figure 119. Stepping into Subprocedure Switch

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

For simple qualified names, of the form DS.SUBF, you can use either of these commands to display or change the variable:

```
EVAL SUBF OF DS
EVAL DS.SUBF
```

For complex qualified names, use the dot-qualification form of the name:

```
EVAL FAMILY.CHILD(2).PETS.PET(3).NAME
```

The scope of the fields used in the EVAL command can be defined by using the QUAL command in languages such as ILE C. However, this command does not currently apply to ILE RPG,

Note: You cannot display return values because there is no external name available for use with the EVAL debug command.

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 120 on page 272 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
    61       D  LastName                     10A   INZ('Jones    ')
    62       D  FirstName                    10A   INZ('Fred     ')
    63
    64        *-------------------------------------------------------------
    65        * Define prototypes for called procedures c_proc and switch
    66        *-------------------------------------------------------------
    67       D c_proc          PR               *   EXTPROC('c_proc')
    68       D   size                        10U 0 VALUE
    69       D   inzval                       1A   CONST
    70
    71       D Switch          PR
    72       D   Parm                         1A
    73
    74        *-------------------------------------------------------------
    75        * Define parameters for non-prototyped call
                                                                    More...
Debug . . .   eval LastName____________________________________________________
_______________________________________________________________________________
F3=End program   F6=Add/Clear breakpoint   F10=Step   F11=Display variable
F12=Resume       F17=Watch variable   F18=Work with watch   F24=More keys
LASTNAME = 'Jones     '
```

Figure 120. Displaying a Field using the EVAL debug command

Figure 121 on page 273 shows the use of the EVAL command with different types of RPG fields. The fields are based on the source in Figure 129 on page 287. Additional examples are also provided in the source 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 Int3                               3I 0 INZ(-128)
  INT3 = -128
> EVAL Int5                               5I 0 INZ(-2046)
  INT5 = -2046
> EVAL Int10                             10I 0 INZ(-31904)
  INT10 = -31904
> EVAL Int20                             20I 0 INZ(-463972)
  INT20 = -463972
> EVAL Unsigned3                          3U 0 INZ(128)
  UNSIGNED3 = 128
> EVAL Unsigned5                          5U 0 INZ(2046)
  UNSIGNED5 = 2046
> EVAL Unsigned10                        10U 0 INZ(31904)
  UNSIGNED10 = 31904
> EVAL Unsigned20                        20U 0 INZ(463972)
  UNSIGNED20 = 463972
> 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 121. Sample EVAL commands based on Module DBGEX

Unexpected Results when Evaluating Variables

If you specify OPTION(*NOUNREF) on the command or Control specification keyword, unreferenced variables in your program source are not generated into the RPG module. If you try to evaluate the unreferenced variable in the debugger, the debugger will indicate that the variable does not exist.

If you are surprised at the value of variables while debugging, check if any of the following is true:

- Your module is optimized. If the module is optimized, the may not show the most current value of a variable. Also if you change a variable using the debugger, the effects of your change may not be reflected in the way the program runs.
- Some input fields are not being read from the file. Normally, input fields that are not used in the program are not affected by an input operation. If you specify the DEBUG keyword on your control specification with no parameters, or with a parameter of either *INPUT or *YES, all input fields will be read in.

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.

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 122 on page 274 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 122. 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. To determine the value of the table index, enter the following command:

```
EVAL _QRNU_TABI_name
```

where *name* represents the table name in question.

Figure 123 on page 274 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 _QRNU_TABI_TableA ** Display value of current index **
  _QRNU_TABI_TABLEA = 1
> 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 123. 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.

If the data structure is qualified, specify the subfields using either of the following notations:

```
EVAL subfield-name OF datastructure-name
EVAL datastructure-name.subfield-name:
```

For example, to display subfield NAME of qualified data structure INFO, type one of the following:

```
EVAL NAME OF INFO
```

```
EVAL NAME OF INFO EVAL INFO.NAME
```

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.

To determine the value of the current index, enter the following command:

```
EVAL _QRNU_DSI_name
```

where *name* represents the data structure name in question.

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 124 on page 276 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 _QRNU_DSI_DS1     ** Determine current index value **
  _QRNU_DSI_DS1 = 1
> EVAL DS1=%INDEX(2)     ** Change the occurrence of DS1 **
  DS1=%INDEX(2) = 2
> 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'
> EVAL QUALDS.ID_NUM              ** Display a subfield of a qualified DS
     QUALDS.ID_NUM = 1100022
> EVAL LIKE_QUALDS.ID_NUM         ** Display the same subfield in a different DS
     LIKE_QUALDS.ID_NUM = 0
> EVAL LIKE_QUALDS.COUNTRY(1)     ** An array element from a qualified DS
     LIKE_QUALDS.COUNTRY(1) = 'CANADA'
> EVAL cust(1).parts.item(2).Id_Num       ** Display a subfield of a complex structure
     CUST(1).PARTS.ITEM(2).ID_NUM = 15
```

Figure 124. 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 125 on page 277 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 125. 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     C1C2C3.. ........ ........ ........   - 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'
```

Displaying UCS-2 Data

The value displayed for UCS-2 fields has been translated into readable characters. For example, if a UCS-2 field has been set to %UCS2('abcde'), then the value displayed for that field would be 'abcde'. You can display UCS-2 data in any field by using the :u suffix for EVAL.

Displaying Variable-Length Fields

When you use EVAL fldname for a variable length field, only the data portion of the field is shown. When you use any suffix such as :c or :x for the field, the entire field including the length is shown. To determine the current length of a variable length field, use EVAL fldname:x. The length is the first four hexadecimal digits, in binary format. You must convert this value to decimal form to get the length; for example, if the result is 003DF1F2..., the length is 003D which is (3 * 16) + 13 = 61.

Displaying Data Addressed by Pointers

If you want to see what a pointer is pointing to, you can use the EVAL command with the :c or :x suffix. For example, if pointer field PTR1 is pointing to 10 bytes of character data,

```
EVAL PTR1:c 10
```

will show the contents of those 10 bytes.

You can also show the contents in hexadecimal using:

```
EVAL PTR1:x 10
```

This would be especially useful when the data that the pointer addresses is not stored in printable form, such as packed or binary data.

If you have a variable FLD1 based on basing pointer PTR1 that is itself based on a pointer PTR2, you will not be able to evaluate FLD1 using a simple EVAL command in the debugger.

Instead, you must explicitly give the debugger the chain of basing pointers:

```
===> EVAL PTR2->PTR1->FLD1
```

For example, if you have the following definitions:

```
D pPointers       S               *
D pointers        DS                    based(pPointers)
D  p1                             *
D  p2                             *
D data1           S             10A   based(p1)
D data2           S             10A   based(p2)
```

you can use these commands in the debugger to display or change the values of DATA1 and DATA2:

```
===> eval pPointers->p1->data1
===> eval pPointers->p2->data2 = 'new value'
```

To determine the expression to specify in the debugger, you start from the end of the expression with the value that you want to evaluate:

```
    data1
```

Then you move to the left and add the name that appears in the BASED keyword for the definition of data1, which is p1:

```
    p1->data1
```

Then you move to the left again and add the name that appears in the BASED keyword for the definition of p1, which is pPointers:

```
    pPointers->p1->data1
```

The expression is complete when the pointer that you have specified was not defined with the BASED keyword. In this case, pPointers is not defined as based, so the debug expression is now complete.

```
===> eval pPointers->p1->data1
```

Evaluating Based Variables

When a variable is based on a pointer, the variable might not be available for evaluation by the debugger using a normal EVAL command. This can happen when the basing pointer is itself based, or when the basing pointer is an entry parameter passed by reference (including read-only reference using the CONST keyword).

For example, in the following program, "basedFld" is based on pointer "parmPtr" which is an input parameter.

```
 /copy myPgmProto
D myPgm           pi
D   parmPtr                       *
D basedFld        s               5a   based(parmPtr)
```

To evaluate "basedFld" in the debugger, use one of these methods:

1. Evaluate the basing pointer using the :c or :x notation described in "Displaying Data Addressed by Pointers" on page 278. For example

   ```
             ===> eval parmPtr:c
             ===> eval parmPtr:x
   ```

   Note: this method does not work well with data that has a hexadecimal representation that does not resemble the natural representation, such as packed, integer or UCS-2 data.

2. Use the debugger's "arrow" notation to explicitly specify the basing pointer. This method can also be used to change the variable.

   ```
             ===> eval parmPtr->basedFld
             ===> eval parmPtr->basedFld = 'abcde'
   ```

If a variable has more than two levels of basing pointer, the second method must be used. For example, in the following program, variable "basedVal" has three levels of basing pointer; it is based on pointer "p1" which is based on pointer "p2" which is further based on pointer "p3". Variable "basedVal" cannot be evaluated in the debugger using simply "EVAL basedVal".

```
D storage         s               5a   inz('abcde')
D val             s               5a
D basedVal        s               5a   based(p1)
D p1              s                *   based(p2)
D p2              s                *   based(p3)
D p3              s                *
D ptr1            s                *   inz(%addr(storage))
D ptr2            s                *   inz(%addr(ptr1))
D ptr3            s                *   inz(%addr(ptr2))
C                   eval      p3 = ptr3
C                   eval      p2 = ptr2
C                   eval      p1 = ptr1
C                   eval      val = basedVal
```

To display a variable such as "basedVal", use the debugger's *p1->p2->name* notation to explicitly specify the basing pointer. To use this notation, specify the variable you want to display, then working to the left, specify the basing pointer name followed by an arrow (->). If the basing pointer is itself based, specify the second basing pointer followed by an arrow, to the left of the previous basing pointer.

For example, to evaluate basedVal:

```
      ===> EVAL p3->p2->p1->basedVal
                            aaaaaaaa
                        bbbb
                    cccc
                dddd
      a. variable name
      b. basing pointer of variable ->
```

```
     c. basing pointer of basing pointer ->
     d. and so on
```

Displaying Null-Capable Fields

You can use the EVAL debug command to display the null indicator of a null-capable field. The null indicator is an internal variable (similar to the index variable for multiple-occurrence DS) which is named _QRNU_NULL_fieldname. The fieldname can be the name of an array if the array is null-capable.

When the debugger displays a null-capable field, the content of the field is displayed regardless of whether the field is considered null. For example, suppose FLD1 is null-capable, and is currently null. Then the result of EVAL _QRNU_NULL_FLD1 is '1' and EVAL FLD1 shows the current content of FLD1, even though its null indicator is on.

```
EVAL _QRNU_NULL_FLD1     Result:  _QRNU_NULL_FLD1 = '1'
```

```
EVAL FLD1                Result:  FLD1 = 'abcde'
```

If a data structure has null-capable subfields, the null indicators for all the null-capable subfields of the data structure are themselves stored as subfields of the data structure _QRNU_NULL_dsname.

If the data structure is not qualified, the null indicator data structure is not qualified. The names of the null capable subfields are in the form _QRNU_NULL_subfieldname.

For example, if qualified data structure DS1 has null-capable subfields FLD1 and FLD2 and non-null-capable subfield FLD3, then the data structure _QRNU_NULL_DS1 would have indicator subfields _QRNU_NULL_NULLFLD1 and NULL2. To display all the null-capable subfields of the data structure, use the debug command

```
EVAL _QRNU_NULL_DS1      Result:  _QRNU_NULL_FLD1 OF _QRNU_NULL_DS1 = '1'
                                  _QRNU_NULL_FLD1 OF _QRNU_NULL_DS1 = '0'
```

If the data structure is qualified, the null indicator data structure is qualified. The names of the null capable subfields are the same as the names of the data structure subfields.

For example, if qualified data structure DS2 has null-capable subfields F1 and F2 and non-null-capable subfield F3, then the data structure _QRNU_NULL_DS2 would have indicator subfields F1 and F2. To display all the null-capable subfields of the data structure, use this debug command:

```
EVAL _QRNU_NULL_DS2      Result:  _QRNU_NULL_DS2.F1 = '0'
                                  _QRNU_NULL_DS2.F2 = '1'
```

To display the null indicator of a variable, use the same EVAL expression in the debugger as you would use to access the variable itself, replacing the outermost name with _QRNU_NULL_name.

```
EVAL FLD1                         Result: 'abc'
EVAL _QRNU_NULL_FLD1              Result: '0'

EVAL SUBF2                        Result: 0
EVAL _QRNU_NULL_SUBF2             Result: '1'

EVAL ARR(3)                       Result: 13
EVAL _QRNU_NULL_ARR(3)            Result: '1'

EVAL DS3.INFO(2).SUB4             Result: 'xyz'
EVAL _QRNU_NULL_DS3.INFO(2).SUB4  Result: '0'
```

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.

%VARS
Identifies the specified parameter as a variable.

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 126 on page 281 shows some examples of the use of %SUBSTR based on the source in Figure 129 on page 287.

```
 > 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 126. 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 123 on page 274 and Figure 124 on page 276.

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.

Use the %VARS debug built-in function when the variable name conflicts with any of the debug command names. For example, EVAL %VAR(EVAL) can be used to evaluate a variable named EVAL, whereas EVAL EVAL would be a syntax error.

Debugging an XML-SAX Handling Procedure

The second parameter passed to an XML-SAX event handling procedure is a numeric value indicating which SAX event was discovered by the parser.

In your RPG code, you can test the event value using special values like `*XML_START_ELEMENT` and `*XML_DOCTYPE_DECL`.

However, these special values are not available in the debugger. Instead, you can use a special array that is made available if you code the DEBUG(*XMLSAX) keyword in your Control specification. The name of the array is _QRNU_XMLSAX; the values of the array elements are the same as the names of the special words, minus the leading "*XML_".

For example, if the name of the second parameter of your XML-SAX event handling procedure is "xmlEvent", then use the following debugger expression to determine the name of the event:

```
EVAL _QRNU_XMLSAX(xmlEvent)

Result: _QRNU_XMLSAX(XMLEVENT) = 'START_DOCUMENT      '
```

The third parameter passed to the event handler is a pointer to the data. See "Displaying Data Addressed by Pointers" on page 278, using the value of the fourth parameter to determine the length of the data, in bytes.

For an Exception event, the fifth parameter holds the error code related to the parsing exception. See "Processing XML Documents" on page 204 for the meanings of the error codes.

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 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'

UCS-2 fields must be changed using hexadecimal constants. For example, since %UCS2('AB') = U'00410042', then to set a UCS-2 field to the UCS-2 form of 'AB' in the debugger, you would use EVAL ucs2 = X'00410042'.

Variable-length fields can be assigned using, for example, EVAL varfldname = 'abc'. This sets the data part of the field to 'abc' and the length part to 3.

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.

To change the null indicator of a variable, use the same EVAL expression in the debugger as you would use to access the variable itself, replacing the outermost name with _QRNU_NULL_name.

```
EVAL FLD1 = 3
EVAL _QRNU_NULL_FLD1 = '0'

EVAL SUBF2 = 5
EVAL _QRNU_NULL_SUBF2 = '0'

EVAL ARR(3) = 0
EVAL _QRNU_NULL_ARR(3) = '1'

EVAL DS3.INFO(2).SUB4 = 'some value'
EVAL _QRNU_NULL_DS3.INFO(2).SUB4 = '0'
```

For more information on debugging null-capable fields, see "Displaying Null-Capable Fields" on page 280.

Figure 127 on page 284 shows some examples of changing field values based on the source in Figure 129 on page 287. 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 127. 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 128 on page 285 shows some examples of displaying field attributes based on the source in Figure 129 on page 287. 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 Int3
  TYPE = INTEGER, LENGTH = 1 BYTES
> ATTR Int5
  TYPE = INTEGER, LENGTH = 2 BYTES
> ATTR Unsigned10
  TYPE = CARDINAL, LENGTH = 4 BYTES
> ATTR Unsigned20
  TYPE = CARDINAL, LENGTH = 8 BYTES
> ATTR Float4
  TYPE = REAL, LENGTH = 4 BYTES
> ATTR Float8
  TYPE = REAL, LENGTH = 8 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 128. 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

You should be aware of the following conditions that exist when you are working with source debug National Language Support for ILE RPG

- 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 129 on page 287 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 130 on page 289 and Figure 131 on page 290 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 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 129 on page 287, 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 131 on page 290, 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 130 on page 289, 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 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 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"')
     D UCS2String      S              5C   INZ(%UCS2('ucs-2'))
     D CharVarying     S              5A   INZ('abc') VARYING
     D Int3            S              3I 0 INZ(-128)
     D Int5            S              5I 0 INZ(-2046)
     D Int10           S             10I 0 INZ(-31904)
     D Int20           S             20I 0 INZ(-463972)
     D Unsigned3       S              3U 0 INZ(128)
     D Unsigned5       S              5U 0 INZ(2046)
     D Unsigned10      S             10U 0 INZ(31904)
     D Unsigned20      S             20U 0 INZ(463972)
     D Float4          S              4f   INZ(7.2098)
     D Float8          S              8f   INZ(-129.0978652)
     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          C                   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)
```

Figure 129. Source for Module DBGEX

```
 *
D DS3             DS
D  Title                          5A   INZ('Mr.  ')
D  LastName                      10A   INZ('Jones    ')
D  FirstName                     10A   INZ('Fred     ')

D QUALDS          DS                   QUALIFIED
D  Id_Num                         8S 0
D  Country                       20A   DIM(10)
D LIKE_QUALDS     DS                   LIKEDS(QUALDS)
D itemInfo        DS                   QUALIFIED
D   ID_Num                       10I 0
D   name                         25A
D items           DS                   QUALIFIED
D   numItems                     10I 0
D   item                               LIKEDS(itemInfo) DIM(10)
D cust            DS                   QUALIFIED DIM(10)
D   name                         50A
D   parts                              LIKEDS(items)
 *-----------------------------------------------------------------*
 * Define prototypes for called procedures c_proc and switch
 *-----------------------------------------------------------------*
D c_proc          PR              *   EXTPROC('c_proc')
D   size                         10U 0 VALUE
D   inzval                        1A   CONST
D Switch          PR
D   Parm                          1A
 *-----------------------------------------------------------------*
 * Define parameters for non-prototyped call
 *     PARM1 is used when calling RPGPROG program.
 *-----------------------------------------------------------------*
D PARM1           S               4P 3 INZ(6.666)
D EXPORTFLD       S               6A   INZ('export') EXPORT
 *=================================================================*
 * Now the operation 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 the main procedure.
 *-----------------------------------------------------------------*
C                   EVAL      SpcPtr = c_proc(SpcSiz : 'P')
```

```
 *-----------------------------------------------------------------*
 * Call a local subprocedure  Switch, which reverses the value of
 * an indicator.
 *-----------------------------------------------------------------*
C                   EVAL      *IN10 = '0'
C                   CALLP     Switch(*in10)
```

```
      *---------------------------------------------------------------*
      * After the following SETON operation, *IN02 = 1.
      *---------------------------------------------------------------*
     C                   SETON                                        020406
     C                   IF        *IN02 = '1'
     C                   MOVE      '1994-09-30'  BigDate
     C                   ENDIF
      *---------------------------------------------------------------*
      * Put a new value in the second cell of Arry.
      *---------------------------------------------------------------*
     C                   MOVE      4             Arry
      *---------------------------------------------------------------*
      *  Now start a formatted dump and return, by setting on LR.
      *---------------------------------------------------------------*
     C                   DUMP
     C                   SETON                                        LR
      *===============================================================*
      * Define the subprocedure Switch.
      *===============================================================*
     P Switch          B
     D Switch          PI
     D   Parm                         1A
      *---------------------------------------------------------------*
      * Define a local variable for debugging purposes.
      *---------------------------------------------------------------*
     D Local           S              5A      INZ('aaaaa')
     C                   IF        Parm = '1'
     C                   EVAL      Parm = '0'
     C                   ELSE
     C                   EVAL      Parm = '1'
     C                   ENDIF
     P Switch          E
      *===============================================================*
      * Compile-time data section for Table.                          *
      *===============================================================*
**
aaa
bbb
ccc
```

```
      *===============================================================*
      *  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 130. Source for OPM Program RPGPGM

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
extern char EXPORTFLD[6];
char *c_proc(unsigned int size, char *inzval)
{
   char *ptr;
   ptr = malloc(size);
   memset(ptr, *inzval, size );
   printf("import string: %6s.\n",EXPORTFLD);
   return(ptr);
}
```

Figure 131. Source for C Procedure cproc

Handling Exceptions

This chapter explains how ILE RPG exception handling works, and how to use:

- Exception handlers
- ILE RPG-specific handlers
- ILE condition handlers
- Cancel handlers

ILE RPG supports the following types of exception handlers:

- RPG-specific handlers, for example, the use of an error indicator, an 'E' operation code extender, a MONITOR group, 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 Exception Handling” on page 293
- “Specifying Error Indicators or the 'E' Operation Code Extender” on page 298
- “Using a File Error (INFSR) Subroutine” on page 301
- “Using a MONITOR Group” on page 299
- “Using a Program Error Subroutine” on page 303

Additionally, to learn how ILE exception handling works, read:

- “Exception Handling Overview” on page 291 (for general concepts)
- “Using RPG-Specific Handlers” on page 298
- The sections on error handling in *ILE Concepts*.

For information on exception handling and the RPG cycle, see *IBM Rational Development Studio for i: ILE RPG 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 (that is, 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 161 for further discussion of a call stack.)

Figure 132 on page 292 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 132 on page 292 Procedure P1 is the control boundary.

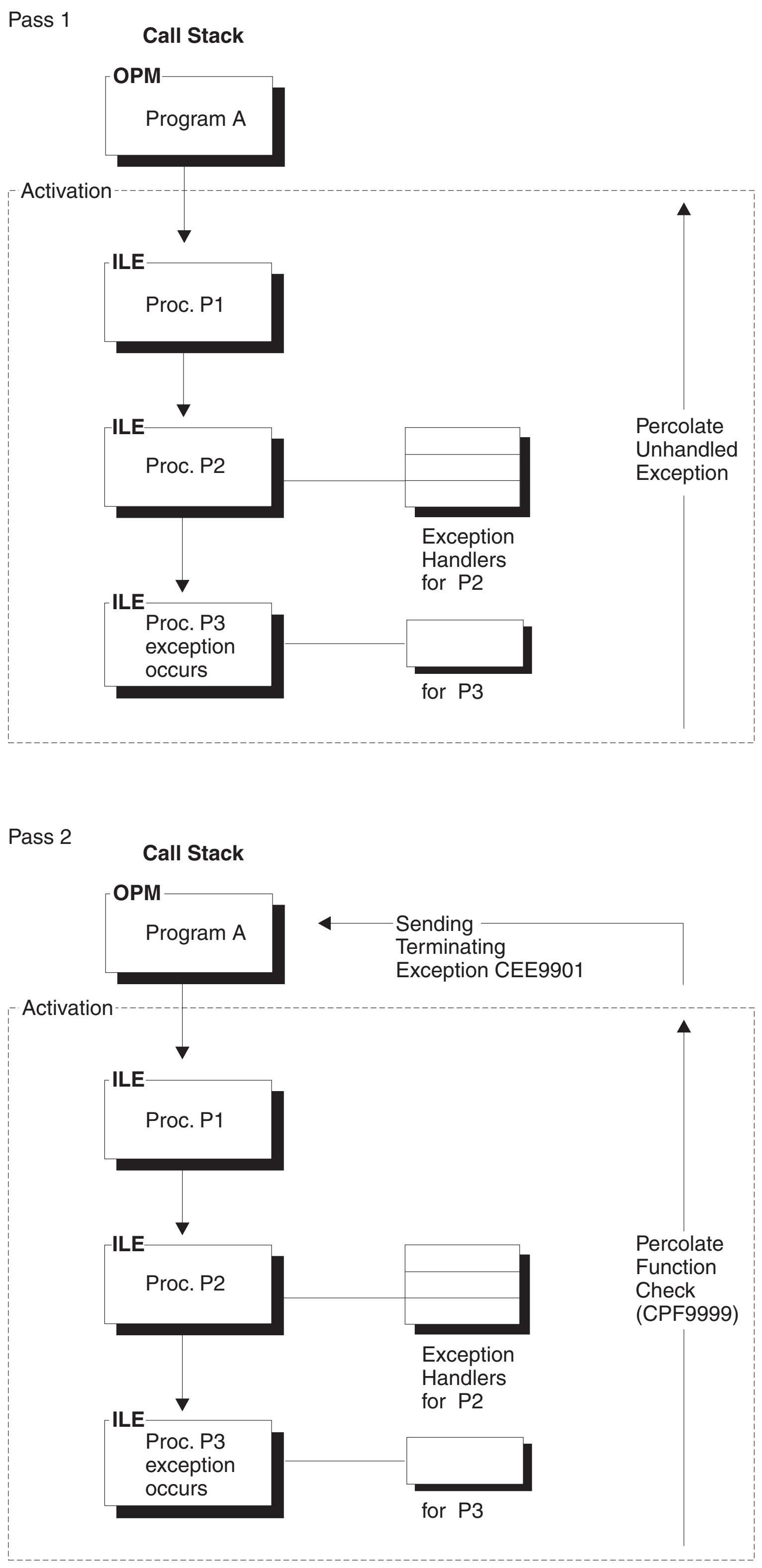

Pass 1
Call Stack
OPM
Program A
Activation
ILE
Proc. P1
ILE
Proc. P2
Exception Handlers for P2
ILE
Proc. P3 exception occurs
for P3
Percolate Unhandled Exception
Pass 2
Call Stack
OPM
Program A
Sending Terminating Exception CEE9901
Activation
ILE
Proc. P1
ILE
Proc. P2
Exception Handlers for P2
ILE
Proc. P3 exception occurs
for P3
Percolate Function Check (CPF9999)

Figure 132. 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 Exception Handling

ILE RPG provides four types of exception handling mechanisms:

- An error indicator or an 'E' operation code extender handler
- A MONITOR group
- 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 five 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. Specify the operation code extender 'E' for the appropriate operation code.
3. Include the code that produces the exception within a MONITOR group.
4. Code a file error subroutine, which is defined by the INFSR keyword on a file description specification, for file exceptions. The file error subroutine must be coded in the same scope as the file; a global file in a cycle module must have its subroutine in the cycle-main procedure, and a local file must have its subroutine in the same subprocedure as the file. You cannot code an INFSR for a global file that is used in a subprocedure.
5. Code a program error subroutine, which is named *PSSR, for program exceptions. Note that a *PSSR is local to the procedure in which it is coded. This means that a *PSSR in a main procedure will handle only those program errors associated with the main procedure. Similarly, a *PSSR in a subprocedure will only handle the errors in that subprocedure.

Exception Handling within a Cycle-Main Procedure

When an exception occurs within a cycle-main procedure ILE RPG 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 operation.
2. If an 'E' operation code extender is present on the calculation specification and the exception is one that is expected for that operation:
   a. The return values for the built-in funtions %STATUS and %ERROR are set.

      Note: %STATUS is set when any exception occurs even if the 'E' extender is not specified.
   b. The exception is handled
   c. Control resumes with the next ILE RPG operation.
3. If no error indicator or 'E' extender is present and the code that generates the exception is in the MONITOR block of a MONITOR group, control will pass to the on-error section of the MONITOR group.

4. If no error indicator or 'E' extender is present, no active MONITOR group could handle the exception, *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.
5. If no error indicator, 'E' extender, or error subroutine is coded and no active MONITOR group could handle the exception, 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 295 for a full description of the RPG default handler.

Exception Handling within Subprocedures

Exception handling within a subprocedure, including one designated as a linear-main procedure, differs from exception handling within a cycle-main procedure in the following ways:

- If you are using a global file, then because you cannot code an INFSR subroutine for that file, you should handle file errors using error indicators, the 'E' operation code extender, or a MONITOR group.
- There is no default handler; in other words, users will never see an inquiry message.

Exception handling within a subprocedure differs from a cycle-main procedure primarily because there is no RPG cycle code generated for subprocedures. As a result there is no default exception handler for subprocedures and so situations where the default handler would be called for a cycle-main procedure correspond to abnormal end of the subprocedure. This means that:

- Factor 2 of an ENDSR operation for a *PSSR subroutine within a subprocedure must be blank. A blank factor 2 in a cycle-main procedure would result in control being passed to the default handler. In a subprocedure, if the ENDSR is reached, then the subprocedure will end abnormally and RNX9001 will be signalled to the caller of the subprocedure.
- If there is no *PSSR and a function check occurs, the procedure is removed from the call stack and the exception is percolated to the caller.
- Since an inquiry message is never issued for an error in a subprocedure, you do not have access to the 'Retry' function available for some I/O errors. If you expect record-lock errors in a subprocedure, you should code an error indicator or an 'E' extender and check if the status is related to a record being locked.

Note that the PSDS and INFDS for global files have module scope. Both main procedures and subprocedures can access them.

TIP

A *PSSR is local to the procedure in which it is coded; therefore, to have a common error routine, you can code a procedure to handle the error and call the procedure from each local *PSSR.

Differences between OPM and ILE RPG Exception Handling

For the most part, exception handling behaves the same in OPM RPG and ILE RPG. 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 will start with the prefix 'RNQ', not 'RPG', as in OPM RPG.

Certain behavioral differences exist for some specific errors. See “Appendix A. Behavioral Differences Between OPM RPG/400 and ILE RPG for AS/400” on page 421 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, 'E' extender, or error 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 procedure, exception handlers have the following priority:

1. Either an error indicator or an 'E' extender handler
2. MONITOR group
3. ILE condition handler
4. I/O error subroutine handler (for file errors) and Program error subroutine handler (for all other errors)
5. RPG default handler for unhandled exceptions (cycle-main procedure only)

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 that has not been handled by an exception handler associated with the call stack entry that first received the exception. When an exception is unhandled, one of the following actions occurs:

If the message type is a function check (CPF9999) *associated with a cycle-main procedure* 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 133 on page 296 depicts this scenario and the events which occur.

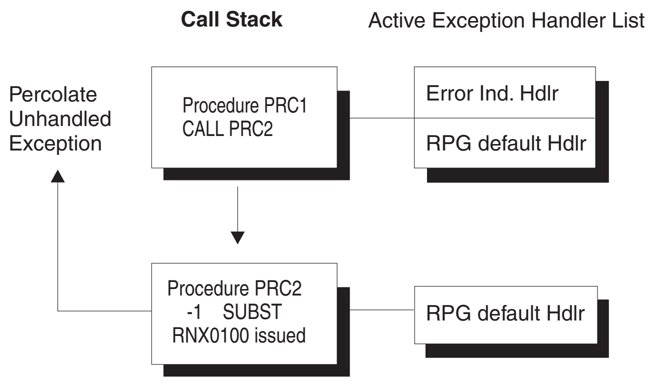


Figure 133. Scenario for Unhandled Escape Message

The following then occurs:

1. Since there is no error indicator, active MONITOR group, 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 in a cycle-main procedure 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, no active MONITOR group, and 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 134 on page 297 depicts this scenario and the events which occur.

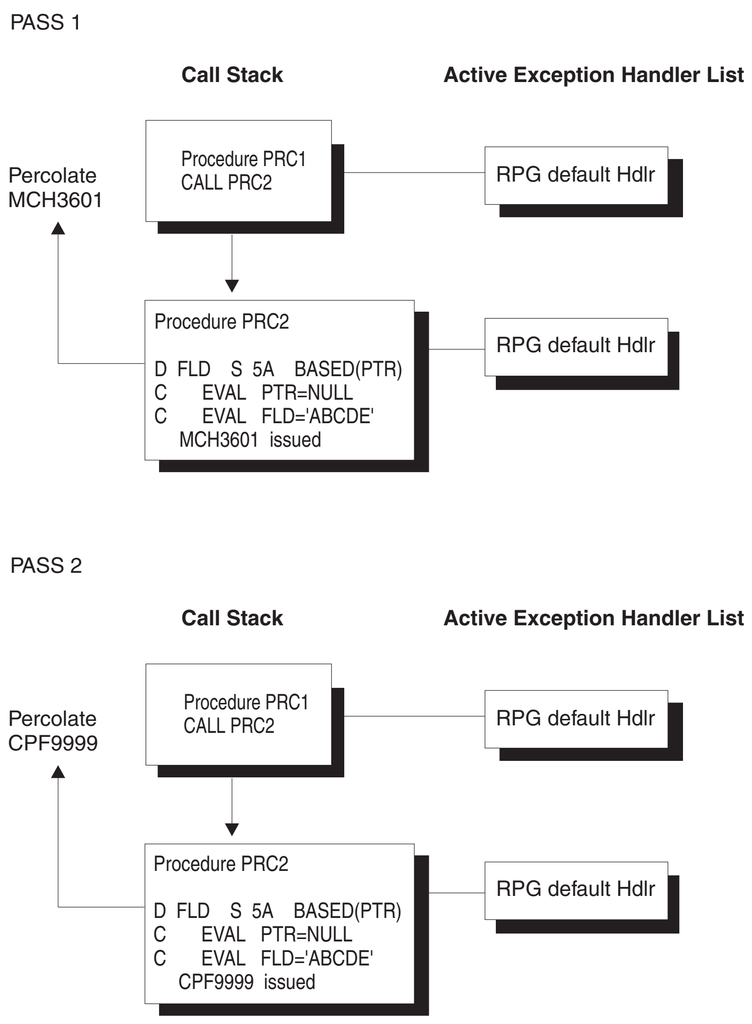


Figure 134. Scenario for Unhandled Function Check

The following then occurs:

1. Since there are no error handlers 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:

- Monitor group
- *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 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 *IBM Rational Development Studio for i: ILE RPG Reference*. For more information on optimization, see "Changing the Optimization Level" on page 128.

Using RPG-Specific Handlers

ILE RPG provides four ways for you to enable HLL-specific handlers and to recover from the exception:

1. error indicators or 'E' operation code extender
2. MONITOR group
3. INFSR error subroutine
4. *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.

If you are using the 'E' extender instead of error indicators, the relevant program and file error information can be obtained by using the %STATUS and %ERROR built-in-functions.

This section provides some examples of how to use each of these RPG constructs. The *IBM Rational Development Studio for i: ILE RPG 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 or the 'E' Operation Code Extender

Operation codes that allow an error indicator also allow the 'E' operation code extender. The CALLP operation also allows the 'E' extender although it does not allow an error indicator. This provides two ILE RPG error handling methods that are essentially the same. Either an error indicator or the 'E' extender can be used to handle the exception for the same operation code, not both.

Note: If an error indicator or an 'E' extender 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 or 'E' extender would be ignored. The error would be treated like any other program error.

To enable the RPG error indicator handler, you specify an error indicator in positions any operation that supports it. 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.

To enable the 'E' operation code extender handler, you specify an 'E' (or 'e') with any of the operation codes that support it, for example, CHAIN(E). Coding the 'E' extender affects the value returned by the built-in functions %ERROR and %STATUS for exceptions. Before the operation begins, the value returned by these built-in functions is set to zero. If an exception occurs on the operation, the return values for these built-in functions are updated accordingly, the appropriate data structure (PSDS or INFDS) is updated, and control returns to the next sequential instruction. You can then use these built-in functions to test the returned values and determine what action to take.

When you specify an error indicator or an 'E' extender 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.

Using a MONITOR Group

A MONITOR group performs conditional error handling based on the status code. If an error occurs, control passes to the appropriate ON-ERROR group within the MONITOR group.

If all the statements in the MONITOR block are processed without errors, control passes to the statement following the ENDMON statement.

The MONITOR group can be specified anywhere in calculations. It can be nested within IF, DO, SELECT, or other MONITOR groups. The IF, DO, and SELECT groups can be nested within MONITOR groups.

If a MONITOR group is nested within another MONITOR group, the innermost group is considered first when an error occurs. If that MONITOR group does not handle the error condition, the next group is considered.

Level indicators can be used on the MONITOR operation, to indicate that the MONITOR group is part of total calculations. For documentation purposes, you can also specify a level indicator on an ON-ERROR or ENDMON operation but this level indicator will be ignored.

Conditioning indicators can be used on the MONITOR statement. If they are not satisfied, control passes immediately to the statement following the ENDMON statement of the MONITOR group. Conditioning indicators cannot be used on ON-ERROR operations individually.

If a MONITOR block contains a call to a subprocedure, and the subprocedure has an error, the subprocedure's error handling will take precedence. For example, if the subprocedure has a *PSSR subroutine, it will get called. The MONITOR group containing the call will only be considered if the subprocedure fails to handle the error and the call fails with the error-in-call status of 00202.

The MONITOR group does handle errors that occur in a subroutine. If the subroutine contains its own MONITOR groups, they are considered first.

Branching operations are not allowed within a MONITOR block, but are allowed within an ON-ERROR block.

A LEAVE or ITER operation within a MONITOR block applies to any active DO group that contains the MONITOR block. A LEAVESR or RETURN operation within a MONITOR block applies to any subroutine, subprocedure, or procedure that contains the MONITOR block.

On each ON-ERROR statment, you specify which error conditions the ON-ERROR group handles. You can specify any combination of the following, separated by colons:

nnnnn
A status code

***PROGRAM**
Handles all program-error status codes, from 00100 to 00999

***FILE**
Handles all file-error status codes, from 01000 to 09999

***ALL**
Handles both program-error and file-error codes, from 00100 to 09999. This is the default.

Status codes outside the range of 00100 to 09999, for example codes from 0 to 99, are not monitored for. You cannot specify these values for an ON-ERROR group. You also cannot specify any status codes that are not valid for the particular version of the compiler being used.

If the same status code is covered by more than one ON-ERROR group, only the first one is used. For this reason, you should specify special values such as *ALL after the specific status codes.

Any errors that occur within an ON-ERROR group are not handled by the MONITOR group. To handle errors, you can specify a MONITOR group within an ON-ERROR group.

```
 * The MONITOR block consists of the READ statement and the IF
 * group.
 * - The first ON-ERROR block handles status 1211 which
 *   is issued for the READ operation if the file is not open.
 * - The second ON-ERROR block handles all other file errors.
 * - The third ON-ERROR block handles the string-operation status
 *   code 00100 and array index status code 00121.
 * - The fourth ON-ERROR block (which could have had a factor 2
 *   of *ALL) handles errors not handled by the specific ON-ERROR
 *   operations.
 *
 * If no error occurs in the MONITOR block, control passes from the
 * ENDIF to the ENDMON.
C                   MONITOR
C                   READ      FILE1
C                   IF        NOT %EOF
C                   EVAL      Line = %SUBST(Line(i) :
C                                          %SCAN('***': Line(i)) + 1)
C                   ENDIF
C                   ON-ERROR  1211
C                    ... handle file-not-open
C                   ON-ERROR  *FILE
C                    ... handle other file errors
C                   ON-ERROR  00100 : 00121
C                    ... handle string error and array-index error
C                   ON-ERROR
C                    ... handle all other errors
C                   ENDMON
```

Figure 135. MONITOR Operation

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 in a cycle-main procedure by specifying a value in Factor 2 of the ENDSR operation of the subroutine. In a subprocedure, factor 2 of the ENDSR must be blank. Use either a GOTO or a RETURN operation prior to the ENDSR operation to prevent the subprocedure from ending abnormally.
- 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 306.

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.

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 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.
- An INFSR cannot handle errors in a global file used by a subprocedure.

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 307. A Factor 2 is not allowed for a file error subroutine in a subprocedure.
5. Code the rest of the file error subroutine. While any of the ILE RPG 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 136 on page 302 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 136. 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'
```

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.

You can code a *PSSR for any (or all) procedures in the module. Each *PSSR is local to the procedure in which it is coded.

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 subprocedures, factor 2 must be blank. For a discussion of the valid entries for Factor 2, see "Specifying a Return Point in the ENDSR Operation" on page 307.
4. Code the rest of the program error subroutine. Any of the ILE RPG compiler operations can be used in the program error subroutine. The ENDSR operation must be the last specification for the program error subroutine.

Figure 137 on page 304 shows an example of a program error subroutine in a cycle-main procedure.

```
 *-----------------------------------------------------------------*
 *  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 main 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 137. Example of *PSSR Subroutine in Cycle-Main Procedure*

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 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.

Figure 138 on page 305 and Figure 139 on page 305 show how you would code similar program error subroutines in a subprocedure. In one example, you code a GOTO and in the other you code a RETURN operation.

```
 *-----------------------------------------------------------------*
 *  Start of subprocedure definition
 *-----------------------------------------------------------------*
P SubProc         B
D SubProc         PI             5P 0
 ...
 *-----------------------------------------------------------------*
 *  Body of code goes here including recovery code.
 *-----------------------------------------------------------------*
C      TryAgain     TAG
C      X            DIV       Divisor       Result
C                   Return    Result
 *-----------------------------------------------------------------*
 *  An error occurs when division by zero takes place.
 *  Control is passed to the *PSSR subroutine.
 *-----------------------------------------------------------------*
C     *PSSR         BEGSR
 *-----------------------------------------------------------------*
 * If this is a divide-by-zero error, add 1 to the divisor
 * and try again
 *-----------------------------------------------------------------*
C                   IF        Err = 102
C                   ADD       1             Divisor
C                   GOTO      TryAgain
C                   ENDIF
 *-----------------------------------------------------------------*
 * If control reaches ENDSR, the procedure will fail
 *-----------------------------------------------------------------*
C                   ENDSR
P                 E
```

*Figure 138. Example of Subprocedure *PSSR Subroutine with GOTO*

```
 *-----------------------------------------------------------------*
 *  Start of subprocedure definition
 *-----------------------------------------------------------------*
P SubProc         B
D SubProc         PI             5P 0
 ...
 *-----------------------------------------------------------------*
 *  Body of code goes here including division operation.
 *-----------------------------------------------------------------*
C      X            DIV       Divisor       Result
C                   Return    Result
 *-----------------------------------------------------------------*
 *  An error occurs when division by zero takes place.
 *  Control is passed to the *PSSR subroutine.
 *-----------------------------------------------------------------*
C     *PSSR         BEGSR
 *-----------------------------------------------------------------*
 * If this is a divide-by-zero error, return 0 from the subprocedure
 *-----------------------------------------------------------------*
C                   IF        Err = 102
C                   RETURN    0
C                   ENDIF
 *-----------------------------------------------------------------*
 * If control reaches ENDSR, the procedure will fail
 *-----------------------------------------------------------------*
C                   ENDSR
P                 E
```

*Figure 139. Example of Subprocedure *PSSR Subroutine with RETURN*

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 140 on page 306 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.     *
 *=================================================================*
 *-----------------------------------------------------------------*
 * 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                   IF        InPssr = 1
C                   MOVE      '*CANCL'      ReturnPt          6
C                   Z-ADD     0             InPssr            1 0
C                   ELSE
C                   Z-ADD     1             InPssr
 *                                                                 *
 *         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
```

Figure 140. Avoiding a Loop in an Error Subroutine

To create the program and start debugging it, using the source in Figure 140 on page 306, type:

```
CRTBNDRPG PGM(MYLIB/NOLOOP) DBGVIEW(*SOURCE)
STRDBG PGM(MYLIB/NOLOOP)
```

Set a break point on the BEGSR line of the *PSSR subroutine so you can step through the *PSSR subroutine.

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 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 in a cycle-main procedure, 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 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 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 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.

Note: You cannot specify a factor 2 entry for an ENDSR in a subprocedure. If you want to resume processing in the subprocedure, you have to use a GOTO operation to a TAG in the body of the subprocedure. Alternatively, you can code a RETURN operation in the *PSSR. The subprocedure will then return to the caller.

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.

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.

  The ILE bindable API CEEHDLR is scoped to the invocation 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. However, it will not affect subprocedures, nor will a subprocedure calling CEEHDLR affect the cycle-main procedure.

  If a subprocedure is called recursively, only the invocation that calls CEEHDLR is affected by it. If you want the condition handler active for every invocation, then CEEHDLR must be called by each invocation.

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 141 on page 309 shows the source for the procedure RPGHDLR. The procedure defines three procedure parameters: an ILE condition token structure, a pointer to a communication area between SHOWERR and RPGHDLR, 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, and if SHOWERR has indicated that out-of-bounds errors maybe ignored, it writes 'Handling...' to QSYSPRT and then sets the action to 'Resume'.
   - Otheriwse, 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

      D RPGHDLR         PR
      D  Parm1                              LIKE(CondTok)
      D  Parm2                          *
      D  Parm3                        10I 0
      D  Parm4                              LIKE(CondTok)

       *-----------------------------------------------------------------*
       * Procedure parameters                                            *
       * 1. Input: Condition token structure                             *
       * 2. Input: Pointer to communication area containing              *
       *        a. A pointer to the PSDS of the procedure being handled  *
       *        b. An indicator telling whether a string error is valid  *
       * 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 RPGHDLR         PI
      D  InCondTok                          LIKE(CondTok)
      D  pCommArea                      *
      D  Action                       10I 0
      D  OutCondTok                         LIKE(CondTok)
```

Figure 141. Source for Condition Handler for Out-of-Bounds Substring Error

```
     D CondTok         DS                   BASED(pCondTok)
     D  MsgSev                        5I 0
     D  MsgNo                         2A
     D                                1A
     D  MsgPrefix                     3A
     D  MsgKey                        4A

     D CommArea        DS                   BASED(pCommArea)
     D   pPSDS                         *
     D   AllowError                   1N

     D PassedPSDS      DS                   BASED(pPSDS)
     D   ProcName              1     10

      *
      * Action codes are:
      *
     D Resume          C                    10
     D Percolate       C                    20

      *-------------------------------------------------------------*
      *      Point to the input condition token                     *
      *-------------------------------------------------------------*
     C                   EVAL      pCondTok = %ADDR(InCondTok)

      *-------------------------------------------------------------*
      *      If substring error, then handle else percolate.        *
      *      Note that the message number value (MsgNo) is in hex.  *
      *-------------------------------------------------------------*
     C                   EXCEPT
     C                   IF        MsgPrefix = 'RNX' AND
     C                             MsgNo     = X'0100' AND
     C                             AllowError = '1'
     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                                          'HDLR: In Handler for '
     O                       ProcName
     OQSYSPRT   E            Handling
     O                                          'HDLR: Handling...'
     OQSYSPRT   E            Perclating
     O                                          'HDLR: Percolating...'
```

Figure 142 on page 311 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 communication area which contains a pointer to the module's PSDS and an indicator telling whether the out-of-bounds string error can be ignored. 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. Indicate to RPGHDLR that the out-of-bounds error is allowed, and then generate an out-of-bounds substring error, then set off the indicator so that RPGHDLR will not allow any unexpected out-of-bounds string errors.

   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, this     *
      * area will contain a pointer to the PSDS of this module, and     *
      * an indicator telling whether an error is allowed.               *
      *                                                                 *
      * We should make sure this program (SHOWERR) does not ignore any  *
      * handled errors, so we will check the 'Error' indicator after    *
      * any operation that might cause an error that RPGHDLR will       *
      * "allow".  We will also check at the end of the program to make  *
      * sure we didn't miss any errors.                                 *
      *-----------------------------------------------------------------*
     D pConHdlr        S               *   PROCPTR
     D                                     INZ(%paddr('RPGHDLR'))

      *-----------------------------------------------------------------*
      * Communication area                                              *
      *-----------------------------------------------------------------*
     D CommArea        DS                  NOOPT
     D   pPsds                         *   INZ(%ADDR(DSPsds))
     D   AllowError                   1N   INZ('0')

      *-----------------------------------------------------------------*
      * PSDS                                                            *
      *-----------------------------------------------------------------*
     D DSPsds         SDS                  NOOPT
     D   ProcName        *PROC

      *-----------------------------------------------------------------*
      * Variables that will be used to cause errors                     *
      *-----------------------------------------------------------------*
     D Arr1            S             10A   DIM(5)
     D Num             S              5P 0

      *-----------------------------------------------------------------*
      * CEEHDLR Interface                                               *
      *-----------------------------------------------------------------*
     D CEEHDLR         PR
     D   pConHdlr                      *   PROCPTR
     D   CommArea                      *   CONST
     D   Feedback                    12A   OPTIONS(*OMIT)

      *-----------------------------------------------------------------*
      * CEEHDLU Interface                                               *
      *-----------------------------------------------------------------*
     D CEEHDLU         PR
     D   pConHdlr                      *   PROCPTR
     D   Feedback                    12A   OPTIONS(*OMIT)
```

Figure 142. Source for Registering a Condition Handler

```
      *-----------------------------------------------------------------*
      * Register the handler and generate errors                        *
      *-----------------------------------------------------------------*
     C                   EXSR      RegHndlr

      *-----------------------------------------------------------------*
      *      Generate a substring error                                 *
      *      This is an "allowed" error for this example (RPGHDLR       *
      *      handles the exception, allowing control to return to the   *
      *      next instruction after the error).                         *
      *      RPGHDLR will not allow the error unless the "AllowError"   *
      *      indicator is set on.  This ensures that if, for example,   *
      *      a SCAN operation is added to SHOWERR later, RPGHDLR will   *
      *      not by default allow it to have an error.                  *
      *-----------------------------------------------------------------*
     C                   Z-ADD     -1            Num
     C                   EVAL      AllowError = '1'
     C     Num           SUBST     'Hello'       Examp            10
     C                   EVAL      AllowError = '0'

      *-----------------------------------------------------------------*
      *      The exception was handled by the handler and control       *
      *      resumes here.                                              *
      *-----------------------------------------------------------------*
     C                   EXCEPT    ImBack

      *-----------------------------------------------------------------*
      *      Generate an array out of bounds error                      *
      *      This is not an "expected" error for this example.          *
      *-----------------------------------------------------------------*
     C                   Z-ADD     -1            Num
     C                   MOVE      Arr1(Num)     Arr1(Num)

      *-----------------------------------------------------------------*
      *      The exception was not handled by the handler, so,          *
      *      control does not return here.  The exception is            *
      *      percolated and control resumes in the *PSSR.               *
      *-----------------------------------------------------------------*

      *-----------------------------------------------------------------*
      *      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                   SETON                                        LR

      *=================================================================*
      * RegHdlr - Call the API to register the Handler                  *
      *=================================================================*
     C     RegHndlr      BEGSR
     C                   CALLP     CEEHDLR(pConHdlr : %ADDR(CommArea) : *OMIT)
     C                   ENDSR

      *=================================================================*
      * DeRegHndlr - Call the API to unregister the Handler             *
      *=================================================================*
     C     DeRegHndlr    BEGSR
     C                   CALLP     CEEHDLU(pConHdlr : *OMIT)
     C                   ENDSR
```

```
      *=================================================================*
      * *PSSR: Error Subroutine for the procedure                       *
      *=================================================================*
     C     *PSSR         BEGSR
     C                   EXCEPT    InPssr
     C                   EXCEPT    Cancelling
      C                   ENDSR     '*CANCL'

      *=================================================================*
      * Procedure Output                                                *
      *=================================================================*
     OQSYSPRT   E            ImBack
     O                                          'I''m Back'
     OQSYSPRT   E            InPssr
     O                                          'In PSSR'
     OQSYSPRT   E            Cancelling
     O                                          'Cancelling...'
```

If you want to try these procedures, follow these steps:

1. To create the procedure RPGHDLR, using the source shown in Figure 141 on page 309, type:

   ```
   CRTRPGMOD MODULE(MYLIB/RPGHDLR)
   ```

2. To create the procedure SHOWERR, using the source shown in Figure 142 on page 311, 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:

   ```
    HDLR: In Handler for SHOWERR
    HDLR: Handling...
    I'm Back
    HDLR: In Handler for SHOWERR
    HDLR: 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).

The Register Call Stack Entry Termination User Exit Procedure (CEERTX) and the Call Stack Entry Termination User Exit Procedure (CEEUTX) ILE bindable APIs provide a way of dynamically registering a user-defined routine to be run when the call stack entry for which it is registered is cancelled. Once registered, the cancel handler remains in effect until the call stack entry is removed, or until CEEUTX is called to disable it. For more information on these ILE bindable APIs, see the *CL and APIs* section of the *Programming* category in the **IBM i Information Center** at this Web site - http://www.ibm.com/systems/i/infocenter/.

Figure 143 on page 314 shows an example of enabling and coding a cancel handler for a subprocedure. (Cancel handlers can also be enabled for cycle-main procedures in the same way.)

```
      *-----------------------------------------------------------------
      * Define the prototype for the cancel handler.  This procedure is
      * a local procedure.
      *-----------------------------------------------------------------
     D CanHdlr         PR
     D   pMsg                          *
      *-----------------------------------------------------------------
      * Define the prototype for a subprocedure to enable the cancel
      * handler.
      *-----------------------------------------------------------------
     D Enabler         PR
      *-----------------------------------------------------------------
      * Define the prototype for a subprocedure to call Enabler
      *-----------------------------------------------------------------
     D SubProc         PR
      *-----------------------------------------------------------------
      * Main procedure.  Call SubProc three times.
      *-----------------------------------------------------------------
     C                   CALLP     SubProc
     C                   CALLP     SubProc
     C                   CALLP     SubProc
     C                   SETON                                        LR
      *-----------------------------------------------------------------
      * Procedure SubProc.  Call Enabler.  Since this call will fail,
      * define a local *PSSR subroutine to handle the error.
      *-----------------------------------------------------------------
     P SubProc         B
     C                   CALLP     Enabler
      *-----------------------------------------------------------------
      * The PSSR has a RETURN operation, so the call from the main
      * procedure to SubProc will not fail.
      *-----------------------------------------------------------------
     C     *PSSR         BEGSR
     C     'Subproc PSSR'DSPLY
     C                   RETURN
     C                   ENDSR
     P SubProc         E
```

Figure 143. Enabling and Coding a Cancel Handler for a Subprocedure

```
      *-----------------------------------------------------------------
      * Procedure Enabler.  This procedure enables a cancel handler,
      * then gets an error which causes Enabler to be canceled.
      *-----------------------------------------------------------------
     P Enabler         B
      * Local variables
     D Handler         S               *   PROCPTR  INZ(%PADDR('CANHDLR'))
     D Msg             S             20A
     D pMsg            S               *   INZ(%ADDR(Msg))
     D Zero            S              5P 0 INZ(0)
     D Count           S              5I 0 INZ(0) STATIC
     D Array           S              1A   DIM(2)
      *-----------------------------------------------------------------
      * Enable the cancel handler.  When this procedure gets canceled,
      * procedure 'CANHDLR' will be called.
      *-----------------------------------------------------------------
     C                   CALLB     'CEERTX'
     C                   PARM                    Handler
     C                   PARM                    pMsg
     C                   PARM                    *OMIT
      *-----------------------------------------------------------------
      * This procedure will be called three times.  The first two times
      * will get an error while the cancel handler is enabled.
      *-----------------------------------------------------------------
     C                   EVAL      Count = Count + 1
     C                   SELECT
     C                   WHEN      Count = 1
     C                   EVAL      Msg = 'Divide by zero'
     C                   EVAL      Zero = Zero / Zero
     C                   WHEN      Count = 2
     C                   EVAL      Msg = 'String error'
     C     'A'           SCAN      'ABC':Zero    Zero
      *-----------------------------------------------------------------
      * On the third call, disable the cancel handler.  The array index
      * error will cause the procedure to fail, but the handler will
      * not be invoked.
      *-----------------------------------------------------------------
     C                   WHEN      Count = 3
     C                   CALLB     'CEEUTX'
     C                   PARM                    Handler
     C                   PARM                    *OMIT
     C                   EVAL      Msg = 'Array index error'
     C                   EVAL      Array(Zero) = 'x'
     C                   ENDSL
     P Enabler         E
```

```
      *-----------------------------------------------------------------
      * Define the cancel handler.  The parameter is a pointer to the
      * 'communication area', a message to be displayed.
      *-----------------------------------------------------------------
     P CanHdlr         B
     D CanHdlr         PI
     D   pMsg                          *
      *-----------------------------------------------------------------
      * Define a field based on the input pointer pMsg.
      *-----------------------------------------------------------------
     D Msg             S             20A         BASED(pMsg)
      *-----------------------------------------------------------------
      * Display the message set by the procedure that enabled the
      * handler.
      *-----------------------------------------------------------------
     C     'Cancel Hdlr 'DSPLY                   Msg
     P CanHdlr         E
```

The following is the output from program CANHDLR. Note that the *PSSR of the procedure SubProc is called three times but the cancel handler is only called twice because it was disabled before the third error.

```
DSPLY  Cancel Hdlr     Divide by zero
DSPLY  Subproc PSSR
DSPLY  Cancel Hdlr     String error
DSPLY  Subproc PSSR
DSPLY  Subproc PSSR
```

Figure 144. Output from CANHDLR program

Problems when ILE CL Monitors for Notify and Status Messages

If your ILE RPG procedure is called by an ILE CL procedure in the same activation group, and the caller is monitoring for status or notify messages, then your ILE CL caller may get control prematurely because of a notify or status message that the ILE RPG procedure was trying to ignore.

For example, if the ILE RPG procedure writes a record to a printer file and the actual printer file has a shorter record length that was declared in the RPG procedure, notify message CPF4906 is sent to the RPG procedure. The RPG exception handling percolates this message which causes the default reply of 'I' to ignore the message. This should allow the output operation to continue normally, and the RPG procedure should proceed to the next instruction.

However, when the ILE CL MONMSG gets control, control passes immediately to the action for the MONMSG or the next statement in the ILE CL procedure.

Note: For this problem to occur, the procedure monitoring for the message does not have to be the immediate caller of the RPG procedure.

This problem is most likely to occur with a MONMSG in an ILE CL caller, but it can also occur with other ILE languages that can monitor for notify and status messages, including ILE RPG using ILE condition handlers enabled using CEEHDLR.

If you encounter this problem, you have two possible ways to avoid it:

1. Ensure that the caller is in a different activation group from the ILE RPG procedure.
2. Enable an ILE condition handler in the RPG procedure. In the handler, if the message is one that you want to ignore, indicate that the message should be handled. Otherwise, indicate that it should be percolated.

   You could also make this handler more generic, and have it ignore all messages with a severity of 0 (information) and 1 (warning).

   Figure 145 on page 317 shows an example of a ILE condition handler that ignores CPF4906.

```
      *-----------------------------------------------------------------
      * Handler definitions
      *-----------------------------------------------------------------
     D Action          S             10I 0
     D Token           DS
     D   MsgSev                       5I 0
     D   MsgNo                        2A
     D                                1A
     D   Prefix                       3A
     D                                4A
      *-----------------------------------------------------------------
      * Actions
      *-----------------------------------------------------------------
     D Handle          C                   10
     D Percolate       C                   20
      *-----------------------------------------------------------------
      * Severities
      *-----------------------------------------------------------------
     D Info            C                   0
     D Warning         C                   1
     D Error           C                   2
     D Severe          C                   3
     D Critical        C                   4
     C     *ENTRY        PLIST
     C                   PARM                    Token
     C                   PARM                    dummy             1
     C                   PARM                    Action
      *-----------------------------------------------------------------
      * If this is CPF4906, handle the notify msg, otherwise percolate
      *-----------------------------------------------------------------
     C                   IF        Prefix = 'CPF' AND
     C                             MsgNo = X'4906'
     C                   EVAL      Action = Handle
     C                   ELSE
     C                   EVAL      Action = Percolate
     C                   ENDIF
     C                   RETURN
```

Figure 145. ILE Condition Handler that Ignores CPF4906

Figure 146 on page 317 shows how you would code the calculations if you wanted to ignore all status and notify messages. Escape messages and function checks have a severity of 2 (Error) or higher.

```
      *-----------------------------------------------------------------
      * Handle information or warning messages, otherwise percolate
      *-----------------------------------------------------------------
     C                   IF        MsgSev <= Warning
     C                   EVAL      Action = Handle
     C                   ELSE
     C                   EVAL      Action = Percolate
     C                   ENDIF
     C                   RETURN
```

Figure 146. How to Ignore Status and Notify Messages

Obtaining a Dump

This chapter describes how to obtain an ILE RPG formatted dump and provides a sample formatted dump.

Obtaining an ILE RPG Formatted Dump

To obtain an ILE RPG formatted dump (printout of storage) for a procedure while it is running, you can:

- Code one or more DUMP operation codes in the 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* manual.

The formatted dump includes field contents, data structure contents, array and table contents, the file information data structures, and the program status data structure. 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 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).

The dump information includes the global data associated with the module. Depending on whether the cycle-main procedure is active, the global data may not represent the values assigned during processing of the *INZSR. If a program consists of more than one procedure, the information in the formatted dump also reflects information about *every* procedure that is active at the time of the dump request. If a procedure is not active, the values of variables in automatic storage will not be valid. If a procedure has not been called yet, the static storage will not be initialized yet. If a procedure has been called recursively, only the information for the most recent invocation will be shown.

There are two occasions when dump data may not be available:

- If the program object was created with debug view *NONE. The dump will contain only the PSDS, file information, and the *IN indicators.
- If a single variable or structure requires more than 16 MB of dump data. This typically occurs with variables or structures that are larger than 5 MB.

If you do not want a user to be able to see the values of your program's variables in a formatted dump, do one of the following:

- Ensure that debug data is not present in the program by removing observability.
- Give the user sufficient authority to run the program, but not to perform the formatted dump. This can be done by giving *OBJOPR plus *EXECUTE authority.

Using the DUMP Operation Code

You can code one or more DUMP operation codes in the calculations of your source to obtain a ILE RPG formatted dump. A new QPPGMDMP spool file is created whenever the DUMP operation occurs.

Note the following about the DUMP operation:

- To determine whether a DUMP operation will cause a formatted dump to be produced, you must check the operation extender on the DUMP operation, and the DEBUG keyword on the control specification. The formatted dump will be produced if the (A) extender on the DUMP operation is specified, or if the DEBUG keyword was specified with no parameter or with a parameter of *DUMP or *YES. Otherwise, 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.

Example of a Formatted Dump

The following figures show an example of a formatted dump of a module similar to DBGEX (see "Sample Source for Debug Examples" on page 286). 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
  Library . . . . . . . . . . . . . . . :   MYLIB          A
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. . . . . . :
Last File Used . . . . . . . . . . . . :
Last File Status . . . . . . . . . . . :
Last File Operation . . . . . . . . . . :                 G
Last File Routine . . . . . . . . . . . :
Last File Statement . . . . . . . . . . :
Last File Record Name . . . . . . . . . :   . . . . . .
Job Name . . . . . . . . . . . . . . . :   MYUSERID
User Name . . . . . . . . . . . . . . . :   MYUSERID
Job Number . . . . . . . . . . . . . . :   002273
Date Entered System . . . . . . . . . . :   09/30/1995
Date Started. . . . . . . . . . . . . . :   *N/A*
Time Started. . . . . . . . . . . . . . :   *N/A*         H
Compile Date . . . . . . . . . . . . . :   123095
Compile Time . . . . . . . . . . . . . :   153438
Compiler Level . . . . . . . . . . . . :   0001
Source File . . . . . . . . . . . . . . :   QRPGLESRC
  Library . . . . . . . . . . . . . . . :   MYLIB
Member . . . . . . . . . . . . . . . . :   DBGEX2
```

Figure 147. 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 source statement in error.

E
ILE RPG 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*
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  K
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
```

Figure 148. Feedback Areas section of Formatted Dump

```
 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      *                                *
```

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.

J

This is the file open feedback information for the file. For a description of the fields, refer to the *Db2 for i* section of the *Database and File Systems* category in the **IBM i Information Center** at this Web site - http://www.ibm.com/systems/i/infocenter/.

K

This is the common I/O feedback information for the file. For a description of the fields, see the above Web site.

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 149. 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 inquiry message with an F option.

Data Information

```
 ILE RPG FORMATTED DUMP
 Module Name. . . . . . . . . . . . . . :   DBGEX2
 Optimization Level . . . . . . . . . . :   *NONE          L   M
 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'
 N
 NAME                  ATTRIBUTES           VALUE
 _QRNU_DSI_DS1         INT(10)              1                '00000001'X       O
 _QRNU_DSI_DS2         INT(10)              2                '00000002'X
 _QRNU_NULL_ARR        CHAR(1)              DIM(8)      P
                         (1-2)              '1'              'F1'X
                         (3)                '0'              'F0'X
                         (4)                '1'              'F1'X
                         (5-6)              '0'              'F0'X
                         (7)                '1'              'F1'X
                         (8)                '0'              'F0'X
 _QRNU_NULL_FLDNULL    CHAR(1)              '1'              'F1'X
 _QRNU_TABI_TABLEA     INT(10)              1                '00000001'X       Q
 ARR                   CHAR(2)              DIM(8)
                         (1-3)              'AB'             'C1C2'X
                         (4-7)              '  '             '4040'X
                         (8)                '1'              'F1'X
 ARRY                  ZONED(3,2)           DIM(2)
                         (1-2)              1.24             'F1F2F4'X
 BASEONNULL            CHAR(10)             NOT ADDRESSABLE
 BASEPTR               POINTER              SPP:E30095A62F001208
 BASESTRING            CHAR(6)              'ABCDEF'         'C1C2C3C4C5C6'X
 BIGDATE               DATE(10)             '1994-09-30'     'F1F9F9F460F0F960F3F0'X
 BIGTIME               TIME(8)              '12.00.00'       'F1F24BF0F04BF0F0'X
 BIGTSTAMP             TIMESTAMP(26)        '9999-12-31-12.00.00.000000'
                       VALUE IN HEX
'F9F9F9F960F1F260F3F160F1F24BF0F04BF0F04BF0F0F0F0F0F0'X
 BIN4D3                BIN(4,3)             -4.321           'EF1F'X
 BIN9D7                BIN(9,7)             98.7654321       '3ADE68B1'X
 DBCSSTRING            GRAPHIC(3)           ' BBCCDD '       'C2C2C3C3C4C4'X
```

Figure 150. Data section of Formatted Dump

```
DS1                   DS                   OCCURS(3)  R
  OCCURRENCE(1)
    FLD1              CHAR(5)              '1BCDE'          'F1C2C3C4C5'X
    FLD1A             CHAR(1)              DIM(5)
                        (1)                '1'              'F1'X
                        (2)                'B'              'C2'X
                        (3)                'C'              'C3'X
                        (4)                'D'              'C4'X
                        (5)                'E'              'C5'X
    FLD2              BIN(5,2)             123.45           '00003039'X
  OCCURRENCE(2)
    FLD1              CHAR(5)              'ABCDE'          'C1C2C3C4C5'X
    FLD1A             CHAR(1)              DIM(5)
                        (1)                'A'              'C1'X
                        (2)                'B'              'C2'X
                        (3)                'C'              'C3'X
                        (4)                'D'              'C4'X
                        (5)                'E'              'C5'X
    FLD2              BIN(5,2)             123.45           '00003039'X
  OCCURRENCE(3)
    FLD1              CHAR(5)              'ABCDE'          'C1C2C3C4C5'X
    FLD1A             CHAR(1)              DIM(5)
                        (1)                'A'              'C1'X
                        (2)                'B'              'C2'X
                        (3)                'C'              'C3'X
                        (4)                'D'              'C4'X
                        (5)                'E'              'C5'X
    FLD2              BIN(5,2)             123.45           '00003039'X
DS2                   CHAR(10)             DIM(2)     S
                        (1)                'aaaaaaaaaa'     '81818181818181818181'X
                        (2)                'bbbbbbbbbb'     '82828282828282828282'X
DS3                   DS                          T
  FIRSTNAME           CHAR(10)             'Fred      '     'C6998584404040404040'X
  LASTNAME            CHAR(10)             'Jones     '     'D1969585A24040404040'X
  TITLE               CHAR(5)              'Mr.  '          'D4994B4040'X
EXPORTFLD             CHAR(6)              'export'         '85A7979699A3'X
FLDNULL               ZONED(3,1)           24.3             'F2F4F3'X
FLOAT1                FLT(4)               1.234500000000E+007  U
                      VALUE IN HEX         '4B3C5EA8'X
FLOAT2                FLT(8)               3.962745000000E+047
                      VALUE IN HEX         '49D15A640A93FCFF'X
INT10                 INT(10)              -31904           'FFFF8360'X
INT5                  INT(5)               -2046            'F802'X
NEG_INF               FLT(8)               -HUGE_VAL  V
                      VALUE IN HEX         'FFF0000000000000'X
NOT_NUM               FLT(4)               *NaN       W
                      VALUE IN HEX         '7FFFFFFF'X
NULLPTR               POINTER              SYP:*NULL
PACKED1D0             PACKED(5,2)          -093.40          '09340D'X
PARM1                 PACKED(4,3)          6.666            '06666F'X
POS_INF               FLT(8)               HUGE_VAL   X
                      VALUE IN HEX         '7FF0000000000000'X
PROCPTR               POINTER              PRP:A00CA02EC200   Y
SPCPTR                POINTER              SPP:A026FA0100C0
SPCSIZ                BIN(9,0)             000000008.       '00000008'X
STRING                CHAR(6)              'ABCDEF'         'C1C2C3C4C5C6'X
TABLEA                CHAR(3)              DIM(3)
                        (1)                'aaa'            '818181'X
                        (2)                'bbb'            '828282'X
                        (3)                'ccc'            '838383'X
UNSIGNED10            UNS(10)              31904            '00007CA0'X
UNSIGNED5             UNS(5)               2046             '07FE'X
ZONEDD3D2             ZONED(3,2)           -3.21            'F3F2D1'X
```

```
Local variables for subprocedure SWITCH:  Z
NAME                  ATTRIBUTES           VALUE
_QRNL_PSTR_PARM       POINTER              SYP:*NULL
LOCAL                 CHAR(5)              '     '          '0000000000'X
PARM                  CHAR(1)              NOT ADDRESSABLE
            * * * * *   E N D   O F   R P G   D U M P   * * * * *
```

L

Optimization level

M
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.

N
Beginning of user variables, listed in alphabetical order, and grouped by procedure. Data that is local to a subprocedure is stored in automatic storage and is not available unless the subprocedure is active. Note that the hexadecimal values of all variables are displayed. :nt Names longer than 131 characters, will appear in the dump listing split across multiple lines. The entire name will be printed with the characters '...' at the end of the lines. If the final portion of the name is longer than 21 characters, the attributes and values will be listed starting on the following line.

O
Internally defined fields which contain indexes multiple-occurrence data structures.

P
Internally defined fields which contain the null indicators for null-capable fields.

Q
Internally defined fields which contain indexes for tables.

R
Multiple-occurrence data structure.

S
Data structures with no subfields are displayed as character strings.

T
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.

U
4-byte and 8-byte float fields.

V
Indicates negative infinity.

W
Stands for 'not a number' indicating that the value is not a valid floating-point number.

X
Indicates positive infinity.

Y
The attribute does not differentiate between basing and procedure pointer.

Z
The local data inside subprocedures is listed separately from the main source section.

Chapter 6. Working with Files and Devices

This section describes how to use files and devices in ILE RPG programs. Specifically, it shows how to:

- Associate a file with 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 Integrated Language Environment program that contains one or more procedures written in RPG IV.

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 IBM i 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.

RPG also allows access to files and devices not directly supported by the system, through the use of SPECIAL files. With a SPECIAL file, you must provide a program that handles the association of the name to the file, and the data management for the file. With other types of files, this is handled by RPG and the operating system.

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 151 on page 326 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 151. 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 IBM i file description that contains the specifications for the actual device.

The RPG device types correspond to the above file types as follows:

Table 68. Correlation of RPG Device Types with IBM i File Types

RPG Device Type	IBM i File Type
DISK	database, save, DDM files
PRINTER	printer files
WORKSTN	display, ICF files
SEQ	tape, diskette, save, printer, database
SPECIAL	N/A

Figure 152 on page 326 illustrates the association of the RPG file name FILEX, as coded in Figure 151 on page 326, with a system file description for a display file.

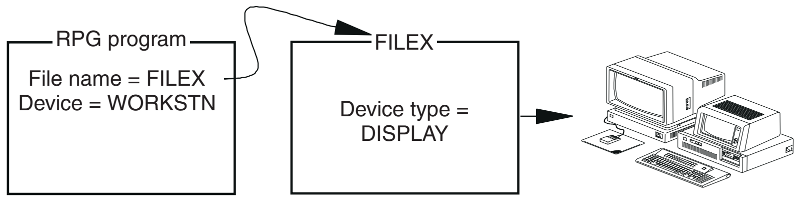


Figure 152. Associating a file name with a display file description

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 153 on page 327 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.

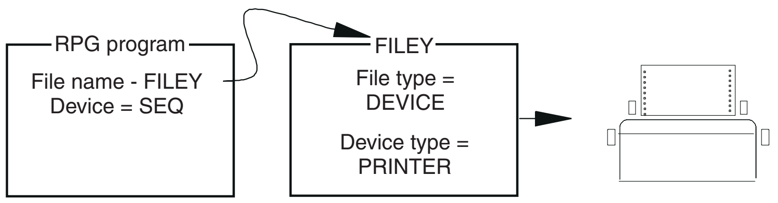


Figure 153. Associating a file name with a display file description

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 336.

Naming Files

On the IBM i, 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 program, the name used for a file within the source is specified in positions 7 through 16 in file description specifications. File names can be up to ten characters long and must be unique within their scope, global or local. The EXTFILE keyword can be used to locate the file at runtime; if the EXTFILE keyword is not specified, the same name is used at runtime to locate the file in the library list. For an externally-described file, the EXTDESC keyword is used to locate the file at compile time; if you want the same file to be used at runtime, you can specify EXTFILE(*EXTDESC).

If you do not specify the EXTFILE keyword to locating the file at runtime, or you do not specify the EXTDESC keyword to locate the file at compile time, you can use a file override command to specify a particular name, library or member. See "Overriding and Redirecting File Input and Output" on page 336 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**, you can use a data structure to hold the data for your file operations, or for global files, you can code the descriptions of the fields 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 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 333 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, or as subfields of a data structure.

At compile time, the compiler uses the data structure you have defined, or the field descriptions in any input or/and output specifications that you coded for the file. It does not retrieve the external descriptions.

Example of Some Typical Relationships between Programs and Files

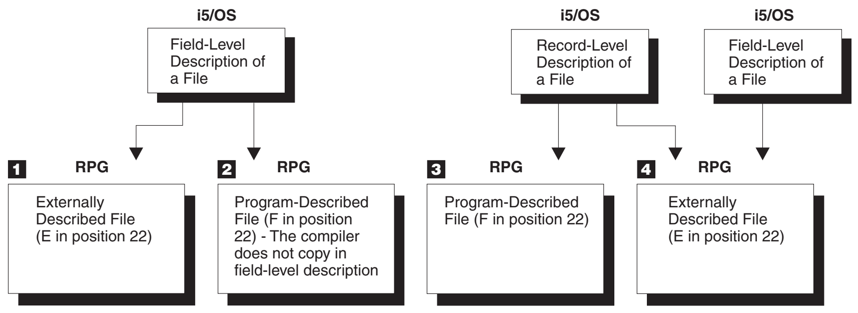

i5/OS
Field-Level Description of a File
i5/OS
Record-Level Description of a File
i5/OS
Field-Level Description of a File
1
RPG
Externally Described File (E in position 22)
2
RPG
Program-Described File (F in position 22) - The compiler does not copy in field-level description
3
RPG
Program-Described File (F in position 22)
4
RPG
Externally Described File (E in position 22)

Figure 154. Typical Relationships between an RPG Program and Files on the IBM i

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 336 for more information.

Defining Externally Described Files

You can use DDS to describe files to the IBM i 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).)

If your file is defined with the QUALIFIED keyword, the format names are specified in the program in the form filename.formatname except when you are specifying the format names in the keywords used to define the file. For example, assume a file is named MYFILE in your program, and it has formats FMT1, FMT2 and FMT3. To rename FMT3 to NEWFMT3, you specify RENAME(FMT3:NEWFMT3); to ignore FMT2, you specify IGNORE(FMT2). Within your calculations, or when specifying the LIKEREC keyword, you use the qualified form of the names, MYFILE.FMT1 and MYFILE.NEWFMT3.

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 global externally described files are optional, and that they are not allowed for externally described files in subprocedures or for qualified files.

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. Each file and unqualified record-format name in the program must be a unique symbolic name. If your file is qualified, so that the record formats are specified in the form filename.fmtname, the names of the formats do not have to be unique within the program.

To rename a record-format name, use the RENAME keyword on the file description specifications for the externally described file as shown in Figure 155 on page 329. The format is RENAME(*old name*:*new name*). Remember that even if the file is qualified, you do not use the qualified form of the name with the RENAME keyword.

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... *
FFilename++IPEASFRlen+LKlen+AIDevice+.Keywords+++++++++++++++++++++++++++++
FITMMSTL   IP   E           K DISK    RENAME(ITEMFORMAT:MSTITM)
 *
```

Figure 155. 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 155 on page 329, the record format ITEMFORMAT in the externally described file ITMMSTL is renamed MSTITM for use in this program. An alternate solution to the problem of having record formats from different files with the same name is to define the files as qualified, using the QUALIFIED keyword. The record formats of a qualified file are specified in the form filename.formatname, so it does not matter if the format name is the same as another name within the program.

Renaming Field Names

You can partially rename all fields in an externally described file by using the PREFIX keyword on the file-description specification for the file. You can either add a prefix to the existing field name or you can replace part of the existing field name with a sequence of characters. The format is PREFIX(*prefix-string*:{*nbr_of_char_replaced*}). Figure 156 on page 330 shows some examples of the use of PREFIX.

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... *
FFilename++IPEASFRlen+LKlen+AIDevice+.Keywords+++++++++++++++++++++++++++++
 * Add the prefix MST to each name in the record format
FITMMSTL   IP   E           K DISK    PREFIX(MST)
 *
 * Change the prefix YTD to YE for each name in the record format
FSALESMSTR IP   E           K DISK    PREFIX(YE:3)
```

Figure 156. Prefix Keyword for Record Format Names in an Externally Described File

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 157 on page 330.

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. Remember that even if the file is qualified, you do not use the qualified form of the name with the IGNORE or INCLUDE keywords.

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 157. IGNORE Keyword for Record Formats in an Externally Described File

Using Input Specifications to Modify an External Description

For a global unqualified file, 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 158 on page 331.
- Rename a field as shown in Figure 158 on page 331.
- Assign control-level indicators to fields as shown in Figure 158 on page 331.
- Assign match-field values to fields for matching record processing as shown in Figure 159 on page 332.
- Assign field indicators as shown in Figure 159 on page 332.

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.

Note: You can explicitly rename a field on an input specification, even when the PREFIX keyword is specified for a file. The compiler will recognize (and require) the name that is first *used* in your program. For example, if you specify the prefixed name on an input specification to associate the field with an indicator, and you then try to rename the field referencing the unprefixed name, you will get an error. Conversely, if you first rename the field to something other than the prefixed name, and you then use the prefixed name on a specification, you will get an error.

```
*.. 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 158. 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 159. 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; they are not allowed for local files in subprocedures, or qualified files. 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, refer to the *Db2 for i* section of the *Database and File Systems* category in the **IBM i Information Center** at this Web site - http://www.ibm.com/systems/i/infocenter/.

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 160 on page 333.

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 160. 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).

Level Checking

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. For this reason, 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 "Handling Exceptions" on page 290.

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, refer to the *Db2 for i* section of the *Database and File Systems* category in the **IBM i Information Center** at this Web site - http://www.ibm.com/systems/i/infocenter/.

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 a global 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" on page 334 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 a global 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 and APIs* section of the *Programming* category in the **IBM i Information Center** at this Web site - http://www.ibm.com/systems/i/infocenter/.

Data Management Operations and ILE RPG 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 69 on page 334 shows the data management operations provided by the IBM i and their corresponding ILE RPG operation. It also shows which operations are allowed for which ILE RPG device type.

Table 69. Data Management Operations and the Corresponding RPG I/O Operation

Data Management Operation	ILE RPG I/O Operation
OPEN	OPEN
READ	
By relative record number	READ, CHAIN
By key	READ, READE, CHAIN, primary and secondary file
Sequential	READ
Previous	READP, READPE
Next	READ, READE
Invited Device	READ
WRITE-READ	EXFMT

Table 69. Data Management Operations and the Corresponding RPG I/O Operation (continued)

Data Management Operation	**ILE RPG I/O Operation**
WRITE By relative record number By key Sequential	WRITE WRITE, EXCEPT, primary and secondary file WRITE, EXCEPT
FEOD	FEOD
UPDATE By relative record number By key	UPDATE, primary and secondary file UPDATE, primary and secondary file
DELETE By relative record number By key	DELETE, primary and secondary file DELETE, primary and secondary file
ACQUIRE	ACQ
RELEASE	REL
COMMIT	COMMIT
ROLLBACK	ROLBK
CLOSE	CLOSE, LR RETURN

General File Considerations

This chapter provides information on the following aspects of file processing on the IBM i 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
- IBM i spooling functions
- using SRTSEQ/ALTSEQ in an RPG program versus a DDS file

Overriding and Redirecting File Input and Output

IBM i 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):

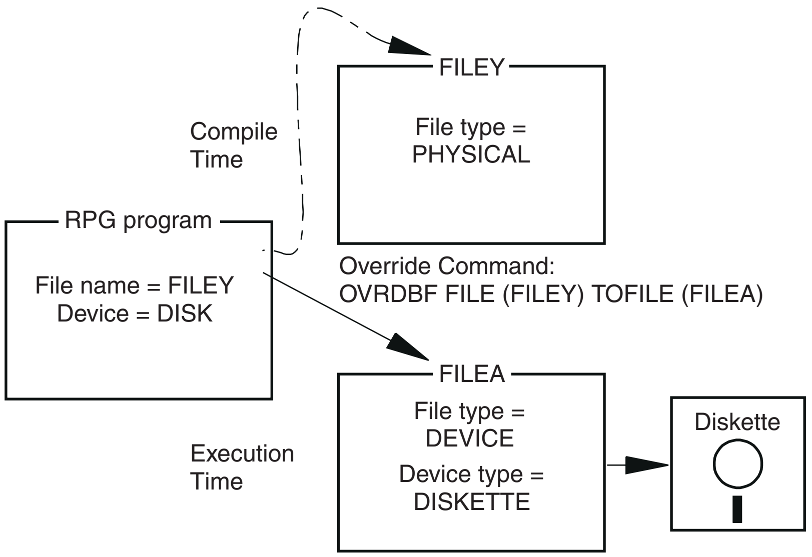


Figure 161. Overriding File Input and Output Example

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.

If you use the OVRDBF command with SHARE(*YES), and one of your programs defines the file with DATA(*NOCVT) in effect, then all the programs that open the file must define the file with DATA(*NOCVT). If the file is opened with different data-conversion options, you may receive CPF417C at runtime.

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 IBM i 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 IBM i system ignores the RPG print spacing and skipping specifications.

There are other file redirections that the IBM i 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.

For more detailed information on valid file redirections and file overrides, refer to the *Db2 for i* section of the *Database and File Systems* category in the IBM i Information Center at this Web site - http://www.ibm.com/systems/i/infocenter/.

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.

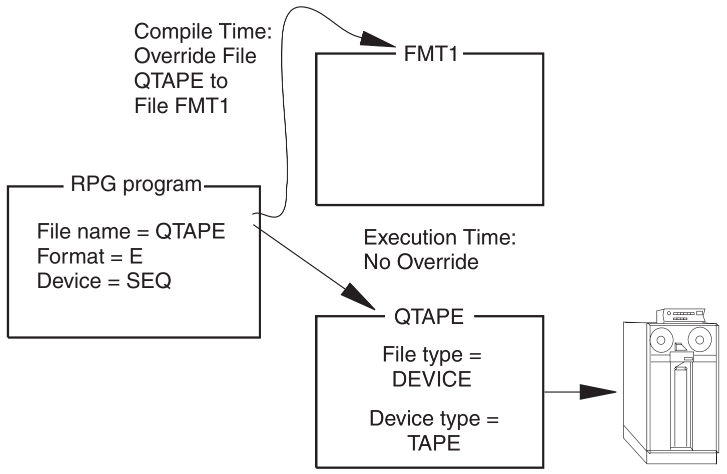


Figure 162. Redirecting File Input and Output Example

File Locking

The IBM i 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, refer to the *Db2 for i* section of the *Database and File Systems* category in the **IBM i Information Center** at this Web site - http://www.ibm.com/systems/i/infocenter/.

The IBM i 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

File Type	Lock State
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 339). 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.

Note: Subprocedures do not get inquiry message, and so this situation should be handled by using an error indicator on the read operation and checking for status 1218 following the read.

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 363 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.

If you are sharing your files so that you can use them in different programs or modules, consider passing the files between your programs and modules as parameters instead. See Passing File Parameters.

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 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.

If a program performs a sequential input operation, and it results in an end-of-file condition, the normal operation is for any subsequent sequential input operation in the same module to immediately result in an end-of-file condition without any physical input request to the database. However, if the file is shared, the RPG runtime will always send a physical input request to the database, and the input operation will be successful if the file has been repositioned by a call to another program or module using the shared file.

Sharing an open data path improves performance because the IBM i 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 70 on page 340 details the system open options allowed for each of the open options you can specify.

Table 70. 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 and activation group versus job level scope, see the *ILE Concepts* manual.

Spooling

Spooling is a system function that puts data into a storage area to wait for processing. The IBM i provides for the use of input and output spooling functions. Each 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, refer to the *Db2 for i* section of the *Database and File Systems* category in the **IBM i Information Center** at this Web site - http://www.ibm.com/systems/i/infocenter/.

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:

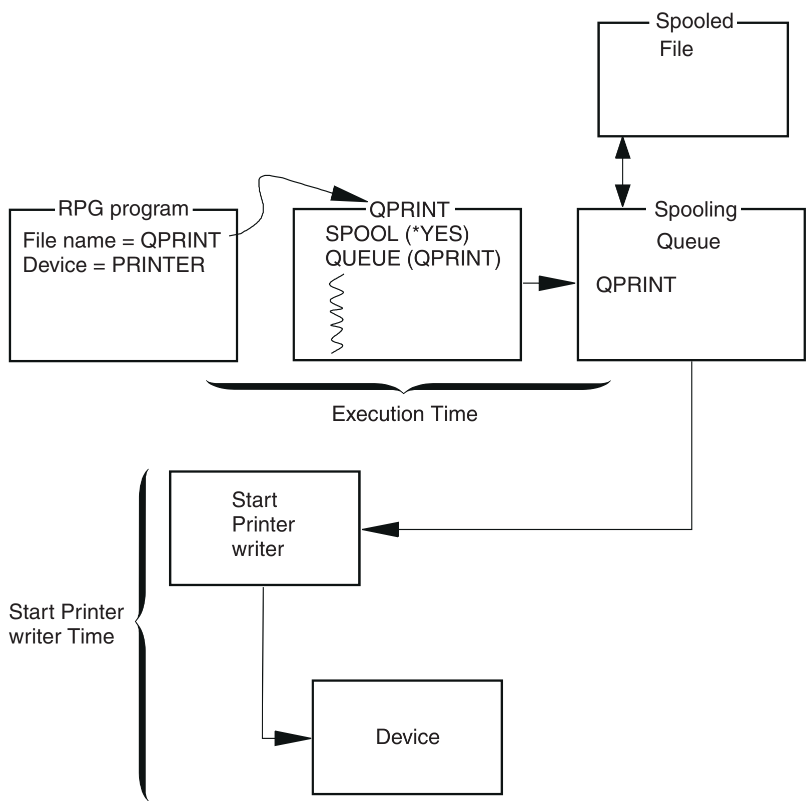


Figure 163. Output Spooling Example

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, IBM i 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.

SRTSEQ/ALTSEQ in an RPG Program versus 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.

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 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 IBM i. 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 IBM i Create File commands. For more information on describing and creating database files, refer to the *Db2 for i* section of the *Database and File Systems* category in the **IBM i Information Center** at this Web site - http://www.ibm.com/systems/i/infocenter/.

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 IBM i 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, refer to the *Db2 for i* section of the *Database and File Systems* category in the **IBM i Information Center** at this Web site - http://www.ibm.com/systems/i/infocenter/.

Figure 164 on page 343 shows an example of the DDS for a database file, and Figure 165 on page 344 for a field-reference file that defines the attributes for the fields used in the database file. See the above Web site for more information on field-reference files.

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 164 on page 343, 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, refer to the *Db2 for i* section of the *Database and File Systems* category in the **IBM i Information Center** at this Web site - http://www.ibm.com/systems/i/infocenter/.

```
*.. 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 164. 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 165 on page 344.

```
*.. 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')
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 165. 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 164 on page 343. 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 program, the edit code used is the one specified in this field-reference file; it cannot be overridden in the ILE RPG program. If the field is used in a program-described output file for an ILE RPG 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. 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 program, you can specify a search argument on certain file operation codes to identify the record you want to process. The ILE RPG 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 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.

Note: Null-capable key fields cannot be used with ALWNULL(*YES) or ALWNULL(*INPUTONLY).

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 or more fields, you can specify a KLIST, a figurative constant, and in free-form calculations only, a list of expressions (enclosed by parentheses) or a %KDS. If the search argument consists of only one field, in addition to the above, you can also specify a literal or variable name.

To process null-valued keys, you can:

- code the search argument using KLIST, in which case the null indicator can be specified in Factor 2 of the KFLD opcode
- code a null-capable field as the search argument in a list (enclosed by parentheses)
- code a null-capable field in the data structure specified in %KDS

For the latter two, the current value of the %NULLIND() for the search argument is used in the search.

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" on page 476. For search arguments in a list or %KDS used in an I/O operation in free-form calculations, the search argument only needs to match in type. Length and format may be different than the key defined in the file.

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

To specify a partial key, you can use a KLIST with fewer KFLD specifications. In free-form calculations, you can also use %KDS with a second parameter indicating the number of keys, or a list of expressions with as many keys as you want. For example, if the file has three keys, but you only want to specify two keys, you can specify the partial key in any of the following ways.

```
DName+++++++++++ETDsFrom+++To/L+++IDc.Keywords+++++++++++++++++++++++++
D keys            DS                  LIKEREC(rec : *KEY)
CL0N01Factor1+++++++Opcode&ExtFactor2+++++++Result++++++++Len++D+HiLoEq
C     klist2        KLIST
C                   KFLD                    k1
C                   KFLD                    k2
 /free
        CHAIN klist2 rec;                // KLIST with two KFLD entries
        CHAIN %KDS(keys : 2) rec;        // %KDS with two keys
        CHAIN (name : %char(id_no)) rec; // a list of two expressions
```

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 list of keys 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. For search arguments in a list or %KDS used in an I/O operation in free-form calculations, the search argument only needs to match in type. Length and format may be different than the key defined in the file. 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 *IBM Rational Development Studio for i: ILE RPG Reference*.

Record Blocking and Unblocking

By default, the RPG compiler unblocks input records and blocks output records to improve run-time performance in SEQ or DISK files when the following conditions are met:

1. The file is program-described or, if externally described, it has only one record format.
2. The keyword RECNO is not used in the file-description specification.

   Note: If RECNO is used, the ILE RPG compiler will not allow record blocking. However, if the file is an input file and RECNO is used, Data Management may still block records if fast sequential access is set. This means that updated records might not be seen right away.
3. One of the following is true:
   a. The file is an output file.
   b. If the file is a combined file, then it is an array or table file.
   c. The file is an input-only file; it is not a record-address file or processed by a record-address file; and uses only the OPEN, CLOSE FEOD, and READ file operations. (In other words, the following file operations are not allowed: READE, READPE, SETGT, SETLL, and CHAIN.)

The RPG compiler generates object program code to block and unblock records for all SEQ or DISK files that satisfy the above conditions. Certain IBM i system restrictions may prevent blocking and unblocking. In those cases, performance is not improved.

You can explicitly request record blocking by specifying the keyword BLOCK(*YES) on the file-description specification for the file. The only difference between the default record blocking and user-requested record blocking is that when BLOCK(*YES) is specified for input files, then the operations SETLL, SETGT and CHAIN can be used with the input file (see condition "3.c" on page 347 above) and blocking will still occur. If the BLOCK keyword is not specified and these operations are used, no record blocking will occur.

You can also prevent the default blocking of records by specifying the keyword BLOCK(*NO) on the file-description specification. If BLOCK(*NO) is specified, then no record blocking is done by the compiler, nor by data management. If the keyword BLOCK is not specified, then default blocking occurs as described above.

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 *IBM Rational Development Studio for i: ILE RPG Reference*.)

You can obtain valid updated feedback information by preventing the file from being blocked and unblocked. Use one of the following ways to prevent blocking:

- Specify BLOCK(*NO) on the file description specification.
- At run time, use the CL command OVRDBF (Override with Database File) with SEQONLY(*NO) specified.

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 166 on page 348 and Figure 167 on page 349 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 166. DDS and corresponding File-Description Specification Detail Flow of RPG IV Exception/Error Handling

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 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 167. (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 168. (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 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 IBM i environment, specify the record-address file length as 4, since each field is 4 bytes in length. When using a record-address file created for 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

Table 71 on page 351 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.

Table 71. Processing Methods for DISK Files

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, Z, or F	I
Randomly	Blank	A, D, G, P, T, Z, or F	I
Sequential within limits (by record-address file)	L	A, D, G, P, T, Z, or F	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, F, or Blank	Blank

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 compiler defaults to or operates as though SEQONLY(*YES) had been specified on the CL command OVRDBF (Override with Database File). (The ILE RPG 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 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 can 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, refer to the *Db2 for i* section of the *Database and File Systems* category in the **IBM i Information Center** at this Web site - http://www.ibm.com/systems/i/infocenter/.

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 compiler defaults to or operates as though SEQONLY(*YES) had been specified on the CL command OVRDBF. (The ILE RPG 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 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, refer to the *Db2 for i* section of the *Database and File Systems* category in the **IBM i Information Center** at this Web site - http://www.ibm.com/systems/i/infocenter/.

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 169 on page 352 and Figure 170 on page 353 shows the data description specifications (DDS) for the physical files used by the examples. Figure 171 on page 353 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 169. 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 170. 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 171. 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*

      *  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 172. 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
```

```
     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 173. 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 174. Sequential-by-Key Processing, Example 3

```
     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
```

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 176 on page 358 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 348).

Example of Random-by-Key Processing

The following is an example of how to use the random-by-key method of processing data. Figure 169 on page 352 and Figure 175 on page 357 show the data description specifications (DDS) for the physical files used by EMSTUPD (Figure 176 on page 358).

```
     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 175. 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 176. 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 177 on page 359 shows an example of an indexed file being processed sequentially within limits. Figure 179 on page 360 shows the same example with externally described files instead of program-described files.

Figure 169 on page 352 shows the data description specifications (DDS) for the physical file used by the program ESWLIM1 (Figure 177 on page 359) and ESWLIM2 (Figure 179 on page 360).

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
      *  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 177. Sequential-within-Limits Processing of an Externally Described File

EXAMPLE PROGRAM 2 (Sequential-within-Limits Processing)

Figure 178 on page 359 shows the data description specifications (DDS) for the record-address limits file used by the program ESWLIM2 (Figure 179 on page 360).

```
     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 178. 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 179. 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 must be a valid existing record; for a new record, that place must 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

Table 72 on page 361 shows the valid file operation codes allowed for DISK files processed by keys and Table 73 on page 362 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.

Table 72. 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	28[1]	34[2]	26-35
I	P/S			K/A/P/G/ D/T/Z/F	CLOSE, FEOD, FORCE
I	P/S	A		K/A/P/G/ D/T/Z/F	WRITE, CLOSE, FEOD, FORCE
I	P/S		L	K/A/P/G/ D/T/Z/F	CLOSE, FEOD, FORCE
U	P/S			K/A/P/G/ D/T/Z/F	UPDATE, DELETE, CLOSE, FEOD, FORCE
U	P/S	A		K/A/P/G/ D/T/Z/F	UPDATE, DELETE, WRITE, CLOSE, FEOD, FORCE
U	P/S		L	K/A/P/G/ D/T/Z/F	UPDATE, DELETE, CLOSE, FEOD, FORCE
I	F			K/A/P/G/ D/T/Z/F	READ, READE, READPE, READP, SETLL, SETGT, CHAIN, OPEN, CLOSE, FEOD
I	F	A		K/A/P/G/ D/T/Z/F	WRITE, READ, READPE, READE, READP, SETLL, SETGT, CHAIN, OPEN, CLOSE, FEOD
I	F		L	K/A/P/G/ D/T/Z/F	READ, OPEN, CLOSE, FEOD

Table 72. Valid File Operations for Keyed Processing Methods (Random by Key, Sequential by Key, Sequential within Limits) (continued)

File-Description Specifications Positions					**Calculation Specifications Positions**
U	F			K/A/P/G/ D/T/Z/F	READ, READE, READPE, READP, SETLL, SETGT, CHAIN, UPDATE, DELETE, OPEN, CLOSE, FEOD
U	F	A		K/A/P/G/ D/T/Z/F	WRITE, UPDATE, DELETE, READ, READE, READPE, READP, SETLL, SETGT, CHAIN, OPEN, CLOSE, FEOD
U	F		L	K/A/P/G/ D/T/Z/F	READ, UPDATE, DELETE, OPEN, CLOSE, FEOD
O	Blank	A		K/A/P/G/ D/T/Z/F	WRITE (add new records to a file), OPEN, CLOSE, FEOD
O	Blank			K/A/P/G/ D/T/Z/F	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.
2. Externally described files require a K in position 34; program-described files require an `A,P,G,D,T,Z,` or F in position 34 and an `I` in position 35.
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 IBM i CREATE FILE command.

Table 73. Valid File Operations for Non-keyed Processing Methods (Sequential, Random by Relative Record Number, and Consecutive)

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

Table 73. Valid File Operations for Non-keyed Processing Methods (Sequential, Random by Relative Record Number, and Consecutive) (continued)

File-Description Specifications Positions					**Calculation Specifications Positions**
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/ F/ 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.
2. A record-address file containing relative record numbers requires a T in position 35.
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.
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 IBM i file creation commands.

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 IBM i:
  1. Prepare for using commitment control:. Use the CL commands CRTJRN (Create Journal), CRTJRNRCV (Create Journal Receiver) and STRJRNPF (Start Journal Physical File).
  2. Notify the IBM i when to start and end commitment control: Use the CL commands STRCMTCTL (Start Commitment Control) and ENDCMTCTL (End Commitment Control). For information on these commands, see the *CL and APIs* section of the *Programming* category in the **IBM i Information Center** at this Web site - http://www.ibm.com/systems/i/infocenter/.

- 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. For information on how the rollback function is performed by the system, refer to the *Recovering your system* manual.

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 364 and the *CL Programming* manual 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 367.

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. For further information on the STRCMTCTL and ENDCMTCTL commands, see the *CL and APIs* section of the *Programming* category in the **IBM i Information Center** at this Web site - http://www.ibm.com/systems/i/infocenter/.

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. For details on lock levels, see the *CL and APIs* section of the *Programming* category in the **IBM i Information Center** at this Web site - http://www.ibm.com/systems/i/infocenter/.

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 the *Recovering your system* manual.

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*.

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 *IBM Rational Development Studio for i: ILE RPG 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 IBM i 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                   :
 *
 *  Use the COMMIT operation to complete a group of operations if
 *  they were successful or rollback the changes if they were not
 *  successful.
 *
C                   UPDATE    MAST_REC                             90
C                   UPDATE    TRAN_REC                             91
C                   IF        *IN90 OR *IN91
C                   ROLBK
C                   ELSE
C                   COMMIT
C                   ENDIF
```

Figure 180. 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 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 181 on page 367 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....
 *  If COMITFLAG = '1' the files are opened under commitment control,
 *  otherwise they are not.
C     *ENTRY        PLIST
C                   PARM                    COMITFLAG
C                   :
C                   :
 *
 *  Use the COMMIT operation to complete a group of operations if
 *  they were successful or rollback the changes if they were not
 *  successful.  You only issue the COMIT or ROLBK if the files
 *  were opened for commitment control (ie. COMITFLAG = '1')
 *
C                   UPDATE    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 181. 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.

Unexpected Results Using Keyed Files

When using READE, READPE, SETLL for equality, or Sequential-within-limits processing by a record address file, normally the key comparisons are done at the data management level. However, there are some situations that do not allow the key comparison to be done at the data management level. When data management cannot perform the key comparison, the comparison is done using the hexadecimal collation sequence. This may cause unexpected results. For example, if ABSVAL is used on a numeric key, both -1 and 1 would be seen as valid search arguments for a key in the file with a value of 1. Using the hexadecimal collating sequence, a search argument of -1 will not succeed for an actual key of 1.

Some of the features that cause the key comparison to differ are:

- A Get Next Key Equal following a Read Multiple does not require a search key to be provided. To circumvent this situation, issue an OVRDBF command with either SEQONLY(*NO) or SEQONLY(*YES 1) specified so a Read multiple will read only one record.
- Keyed feedback was not requested for the file at open time.
- The Read request was performed via a group-by view of the data. To circumvent this situation, use a physical copy of the group-by data.
- The file is a Distributed Data Management (DDM) file and the remote file was created before Version 3 Release 1 Modification 0.

Some of the features that will cause a hexadecimal key comparison to differ from a key comparison performed by data management are:

- ALTSEQ was specified for the file
- ABSVAL, ZONE, UNSIGNED or DIGIT keywords on key fields
- Variable length, Date, Time or Timestamp key fields
- ALWNULL(*USRCTL) is specified as a keyword on a control specification or as a command parameter and a key in the record or search argument has a null value. The key in the file or search argument has null values. This applies only to externally described files.
- SRTSEQ for the file is not hexadecimal
- A numeric sign is different from the system-preferred sign
- The CCSID of a key in the file is different from the CCSID of the job

DDM Files

ILE RPG 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 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 the *Db2 for i* section of the *Database and File Systems* category in the **IBM i Information Center** at this Web site - http://www.ibm.com/systems/i/infocenter/.

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 or READPE operation, EQ indicator for SETLL, or during sequential-within-limits processing by a record address file. The READE or READPE operation, EQ indicator for SETLL, or during 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 DDS features are used that cause more than one search argument to match a given key in the file. For example, if ABSVAL is used on a numeric key, both -1 and 1 would succeed as search arguments for a key in the file with a value of 1. Using the hexadecimal collating sequence, a search argument of -1 will not succeed for an actual key of 1. 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
- ALWNULL(*USRCTL) was specified on the creation command and a key in the record or the search argument has a null value (this applies only to externally described files)

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 or READPE operation, EQ indicator for SETLL, 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.

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 "Using WORKSTN Files" on page 382

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 74 on page 369 to access the associated externally attached devices:

Table 74. IBM i 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

Table 74. IBM i Device Files, Related CL commands, and RPG Device Name (continued)

Device File	Associated Externally Attached Device	CL commands	RPG Device Name
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 programs associate with the printer files on the IBM i:

Printer files allow you to print output files. This chapter provides information on how to specify and use printer files in ILE RPG 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 *IBM Rational Development Studio for i: ILE RPG Reference* for the valid output specification entries for an externally-described file.

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, or you can use the IBM-supplied file names.

For information on the CRTPRTF command, see the *CL and APIs* section of the *Programming* category in the **IBM i Information Center** at this Web site — http://www.ibm.com/systems/i/infocenter/.

For information on IBM-supplied file names and the DDS for externally-described printer files, refer to the *Db2 for i* section of the *Database and File Systems* category in the **Information Center** at the above Web site.

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 *IBM Rational Development Studio for i: ILE RPG 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 *IBM Rational Development Studio for i: ILE RPG 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 "Handling Exceptions" on page 290 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 75 on page 372 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 183 on page 373.
- 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 75. 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 182 on page 373 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 182. Printing a Heading on Every Page

Example of Printing a Field on Every Page

Figure 183 on page 373shows 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 183. 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 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 184 on page 374 shows the normal processing of overflow printing when fetch overflow is set on and when it is set off.

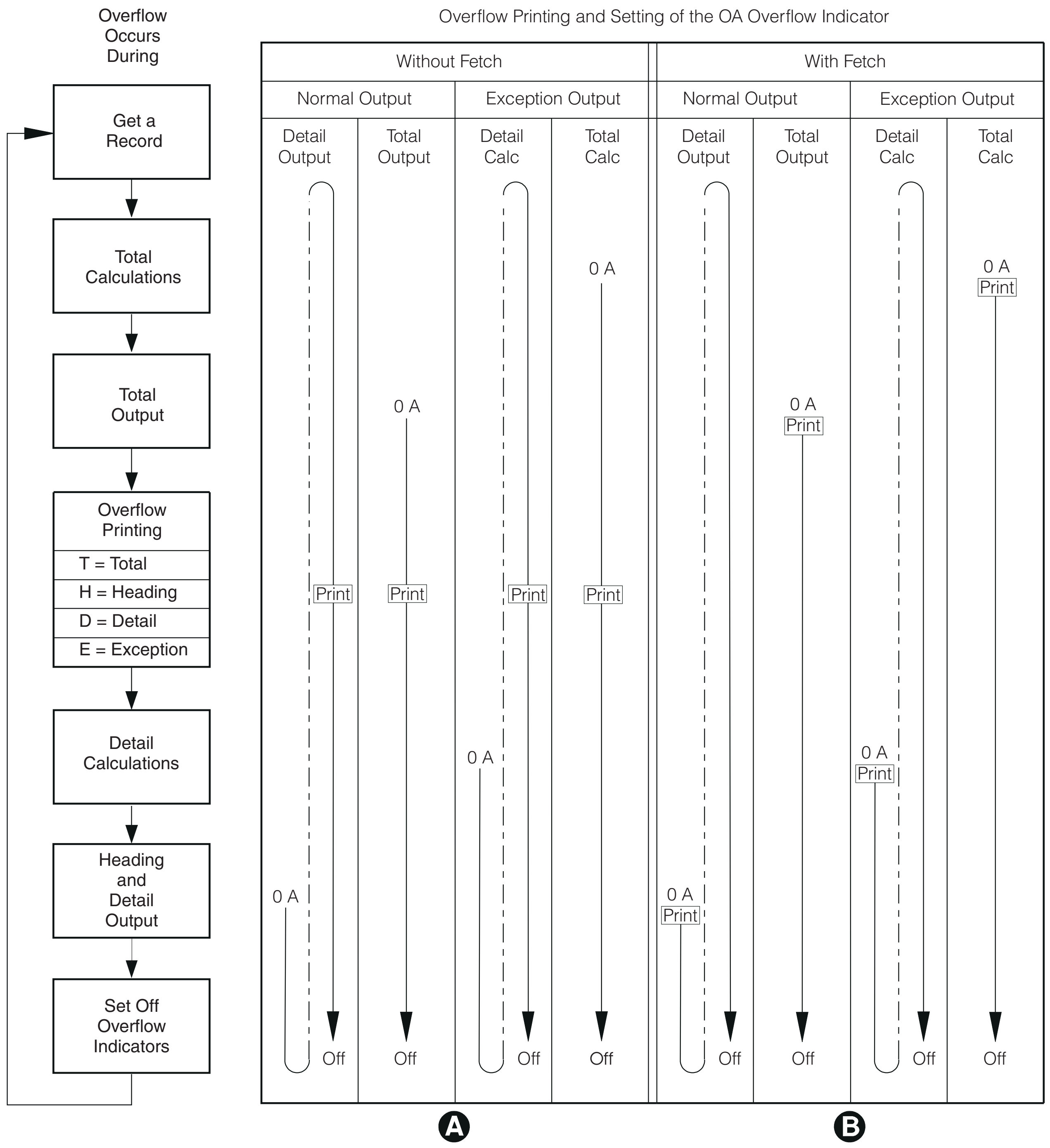


Figure 184. Overflow Printing: Setting of the Overflow Indicator

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 185 on page 375 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
O..............N01N02N03Field+++++++++YB.End++PConstant/editword/DTformat++
O                                           15 'EMPLOYEE TOTAL'
O          TF   L1                     1
O                       EMPLTOT             25
O          T    L1                     1
O                       EMPLTOT             35
O          T    L1                     1
O                       EMPLTOT             45
O          TF   L1                     1
O                       EMPLTOT             55
O          T    L1                     1
O                       EMPLTOT             65
O          T    L1                     1
O                       EMPLTOT             75
O          T    L1                     1
O*
```

Figure 185. 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 76 on page

376. To use the OPM-defined data structure layout, specify PRTCTL(*data-structure name*:*COMPAT). The OPM PRTCTL data structure layout is shown in Table 77 on page 376.

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 76. 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 77. 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 186 on page 377 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 186. 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 "Using WORKSTN Files" on page 382 for a discussion on how to use WORKSTN files.

Using Sequential Files

Sequential files in an ILE RPG 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, such as tape, printer or 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 187 on page 378. The actual device to be associated with the file while running the program can be specified by a IBM i 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.

Table 78. Valid File Operation Codes for a Sequential File

File Description Specifications Positions		**Calculation Specifications Positions**
17	**18**	**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.

Example of Specifying a Sequential File

Figure 187 on page 378 shows an example of how to specify a SEQ file in an ILE RPG source member.

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+ ...*
FFilename++IPEASFRlen+LKlen+AIDevice+.Keywords+++++++++++++++++++++++++++++
FTIMECDS   IP   E             DISK
FPAYOTIME  O    F  132        SEQ
 *
```

Figure 187. SEQ Device

A SEQ device is specified for the PAYOTIME file. When the program is run, you can use a IBM i 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 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 calls this user-written routine to open the file, read and write the records, and close the file. ILE RPG 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 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:

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 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 program. Status must contain one of the following return values when the user-written routine returns control to the ILE RPG 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 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 188 on page 381. 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.

Table 79 on page 380 shows the file operation codes that are valid for a SPECIAL file.

Table 79. Valid File Operations for a SPECIAL File

File Description Specifications Positions		Calculation Specifications Positions
17	**18**	**26-35**
I	P/S	CLOSE, FEOD
C	P/S	WRITE, CLOSE, FEOD
U	P/S	UPDATE, DELETE, CLOSE, FEOD
O		WRITE, OPEN, CLOSE, FEOD
I	F	READ, OPEN, CLOSE, FEOD
C	F	READ, WRITE, OPEN, CLOSE, FEOD
U	F	READ, UPDATE, DELETE, OPEN, CLOSE, FEOD

Example of Using a Special File

Figure 188 on page 381 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 188. SPECIAL Device

Figure 189 on page 381 shows the user-written program USERIO.

```
     *.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... *
     DName+++++++++++ETDsFrom+++To/L+++IDc.Functions++++++++++++++++++++++++++++
     D ERROR           S              5S 0
     *.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... *
     CL0N01Factor1+++++++Opcode(E)+Factor2+++++++Result++++++++Len++D+HiLoEq....

      *----------------------------------------------------------------*
      * The first 4 parameters are ILE RPG 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 189. 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 program and the user-written routine USERIO; whereas the RPG-created parameter list can be accessed only by internal ILE RPG logic and the user-written routine.

Using WORKSTN Files

Interactive applications on the IBM i 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 IBM i. 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 IBM i. You use ICF files to communicate with (send data to and receive data from) other application programs on remote systems (IBM i or other operating systems). 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 IBM i 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* manual.

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 *Db2 for i* section of the *Database and File Systems* category in the **IBM i Information Center** at this Web site - http://www.ibm.com/systems/i/infocenter/.)

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, refer to the *Db2 for i* section of the *Database and File Systems* category in the **IBM i Information Center** at this Web site - http://www.ibm.com/systems/i/infocenter/.

Figure 190 on page 383 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 190. 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 IBM i 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 IBM i 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 IBM i 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 IBM i system provides device-control information for processing input/output operations for the device. When an input record is requested from the device, the IBM i system issues the request, and then removes device-control information from the data before passing the data to the program. In addition, the IBM i 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 IBM i system. The IBM i 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 390.

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 191 on page 386.) 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 192 on page 387 shows an example of the DDS for a subfile-record format, and Figure 193 on page 388 shows an example of the DDS for a subfile control-record format.

For a description of how to use subfile keywords, refer to the *Db2 for i* section of the *Database and File Systems* category in the **IBM i Information Center** at this Web site - http://www.ibm.com/systems/i/infocenter/..

```
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 191. 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 *Db2 for i* section of the *Database and File Systems* category in the **IBM i Information Center** at this Web site - http://www.ibm.com/systems/i/infocenter/.) 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 192. 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 406.

```
*.. 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 193. 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 Table 80 on page 390. 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 IBM i 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 IBM i 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 IBM i 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, refer to the *Db2 for i* section of the *Database and File Systems* category in the **IBM i Information Center** at this Web site - http://www.ibm.com/systems/i/infocenter/.

Valid WORKSTN File Operations

Table 80 on page 390 shows the valid file operation codes for a WORKSTN file.

Table 80. 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]

Table 80. Valid File Operation Codes for a WORKSTN File (continued)

File Description Specifications Positions		Calculation Specifications Positions
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.
2. If the EXFMT operation is used, the file must be externally described (an E in position 19 of the file description specifications).
3. For subfile record formats, the UPDATE, CHAIN, and READC operations are also valid.

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 *IBM Rational Development Studio for i: ILE RPG 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 the *Db2 for i* section of the *Database and File Systems* category in the **IBM i Information Center** at this Web site - http://www.ibm.com/systems/i/infocenter/. You can also refer to information on ICF files in *ICF Programming* manual.

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 section on inviting a program device in *ICF Programming* manual.
- 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* manual.
- 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 *IBM Rational Development Studio for i: ILE RPG 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.

Example of an Interactive Application

This chapter illustrates some common workstation applications and their ILE RPG 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 81. Description of Each Module in the Interactive Application Example

Module	**Description**
"Main Menu Inquiry" on page 394	An example of a basic menu inquiry program that uses a WORKSTN file to display menu choices and accept input.
"File Maintenance" on page 397	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 406	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 412	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 194 on page 394 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 194. 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 195. 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.Maintain (add, update, delete and display)   *
       //                   customer records.                            *
       //                 2.Search customer record by zip code.          *
       //                 3.Search customer record by name.              *
       //*****************************************************************

      Fmainmenu  cf   e             workstn indds(indicators)

       // Prototype definitions:
      D CustMaintain    pr                  extproc('CUSMNT')
      D SearchZip       pr                  extproc('SCHZIP')
      D SearchName      pr                  extproc('SCHNAM')

       // Field definitions:
      D indicators      ds
      D    exitKey                      n   overlay(indicators:3)
      D    maintainKey                  n   overlay(indicators:5)
      D    srchZipKey                   n   overlay(indicators:6)
      D    srchCustKey                  n   overlay(indicators:7)

       /free
            // Keep looping until exit key is pressed
            dow  '1';
               // Display main menu
               exfmt hdrscn;

               // Perform requested action
               if exitKey;
                  // Exit program
                  leave;

               elseif maintainKey;
                  // Maintain customer data
                  CustMaintain();

               elseif srchZipKey;
                  // Search customer data on ZIP code
                  SearchZip();

               elseif srchCustKey;
                  // Search customer data on customer name
                  SearchName();
               endif;
            enddo;

            *inlr = *on;
       /end-free
```

Figure 196. 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 197 on page 397.

```
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 197. 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 198. 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
     A          R HDRSCN
     A                                      TEXT('PROMPT FOR CUST NUMBER')
     A                                      CA03(03 'END MAINTENANCE')
     A                                      CF05(05 'ADD MODE')
     A                                      CF06(06 'UPDATE MODE')
     A                                      CF07(07 'DELETE MODE')
     A                                      CF08(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'
```

Figure 199. DDS for display device file MNTMENU

```
     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)
     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'
```

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 indds(indicators)

 // Field definitions:

D indicators      ds
D   exitKey                     n   overlay(indicators:3)
D   disableInput                n   overlay(indicators:4)
D   addKey                      n   overlay(indicators:5)
D   updateKey                   n   overlay(indicators:6)
D   deleteKey                   n   overlay(indicators:7)
D   displayKey                  n   overlay(indicators:8)
D   prevKey                     n   overlay(indicators:12)
D   custExists                  n   overlay(indicators:51)
D   custNotFound                n   overlay(indicators:52)

 // Key list definitions:

C     CSTKEY        KLIST
C                   KFLD                    CUST
```

Figure 200. Source for module CUSMNT

```
 //*****************************************************************
 //       MAINLINE                                                 *
 //*****************************************************************

 /free

  mode = 'DISPLAY';
  exfmt hdrscn;

  // Loop until exit key is pressed
  dow not exitKey;
     exsr SetMaintenanceMode;

     if cust <> 0;
        if mode = 'ADD';
           exsr AddSub;
        elseif mode = 'UPDATE';
           exsr UpdateSub;
        elseif mode = 'DELETE';
           exsr DeleteSub;
        elseif mode = 'DISPLAY';
           exsr InquirySub;
        endif;
     endif;

     exfmt hdrscn;
     custExists   = *off;   // turn off error messages
     CustNotFound = *off;
  enddo;

  *inlr = *on;
```

```
//*****************************************************************
//    SUBROUTINE - AddSub                                         *
//    PURPOSE    - Add new customer to file                       *
//*****************************************************************
begsr AddSub;

   // Is customer number already in file?
   chain CstKey cmlrec1;
   if %found(cusmstl1);
      // Customer number is already being used
      custExists = *on;
      leavesr;
   endif;

   // Initialize new customer data
   custExists   = *off;   // turn off error messages
   CustNotFound = *off;
   name = *blank;
   addr1 = *blank;
   addr2 = *blank;
   city = *blank;
   state = *blank;
   zip = 0;

   // Prompt for updated data for this customer record
   exfmt cstbld;

   // If OK, add customer to the customer file
   if not *in12;
      write cmlrec1;
   endif;
endsr;  // end of subroutine AddSub



//*****************************************************************
//    SUBROUTINE - UpdateSub                                      *
//    PURPOSE    - Update customer master record                  *
//*****************************************************************
begsr UpdateSub;

   // Lookup customer number
   chain cstkey cmlrec1;
   if not %found(cusmstl1);
      // Customer is not found in file
      custNotFound = *on;
      leavesr;
   endif;

   // Display information for this customer
   disableInput = *off;
   exfmt cstinq;
   if not prevKey;
      // Update information in file
      update cmlrec1;
   else;
      // If we don't want to update, at least unlock
      // the record.
      unlock cusmstl1;
   endif;
endsr;  // end of subroutine UpdateSub;
```

```
        //****************************************************************
        //    SUBROUTINE - DeleteSub                                     *
        //    PURPOSE    - Delete customer master record                 *
        //****************************************************************
        begsr DeleteSub;

           // Lookup customer number
           chain cstkey cmlrec1;
           if not %found(cusmstl1);
              // Customer is not found in file
              custNotFound = *on;
              leavesr;
           endif;

           // Display information for this customer
           disableInput = *on;
           exfmt cstinq;
           if not prevKey;
              // Delete customer record
              delete cmlrec1;
           else;
              // If we don't want to delete, at least unlock
              // the record.
              unlock cusmstl1;
           endif;
        endsr;  // end of subroutine DeleteSub



        //****************************************************************
        //    SUBROUTINE - InquirySub                                    *
        //    PURPOSE    - Display customer master record                *
        //****************************************************************
        begsr InquirySub;

           // Lookup customer number
           chain(n) cstkey cmlrec1;  // don't lock record
           if not %found(cusmstl1);
              // Customer is not found in file
              custNotFound = *on;
              leavesr;
           endif;

           // Display information for this customer
           disableInput = *on;
           exfmt cstinq;
        endsr;  // end of subroutine InquirySub;
```

```
        //****************************************************************
        //    SUBROUTINE - SetMaintenanceMode                            *
        //    PURPOSE    - Set maintenance mode                          *
        //****************************************************************
        begsr SetMaintenanceMode;
           if addKey;
              mode = 'ADD';
           elseif updateKey;
              mode = 'UPDATE';
           elseif deleteKey;
              mode = 'DELETE';
           elseif displayKey;
              mode = 'DISPLAY';
           endif;
        endsr;  // end of subroutine SetMaintenanceMode

       /end-free
```

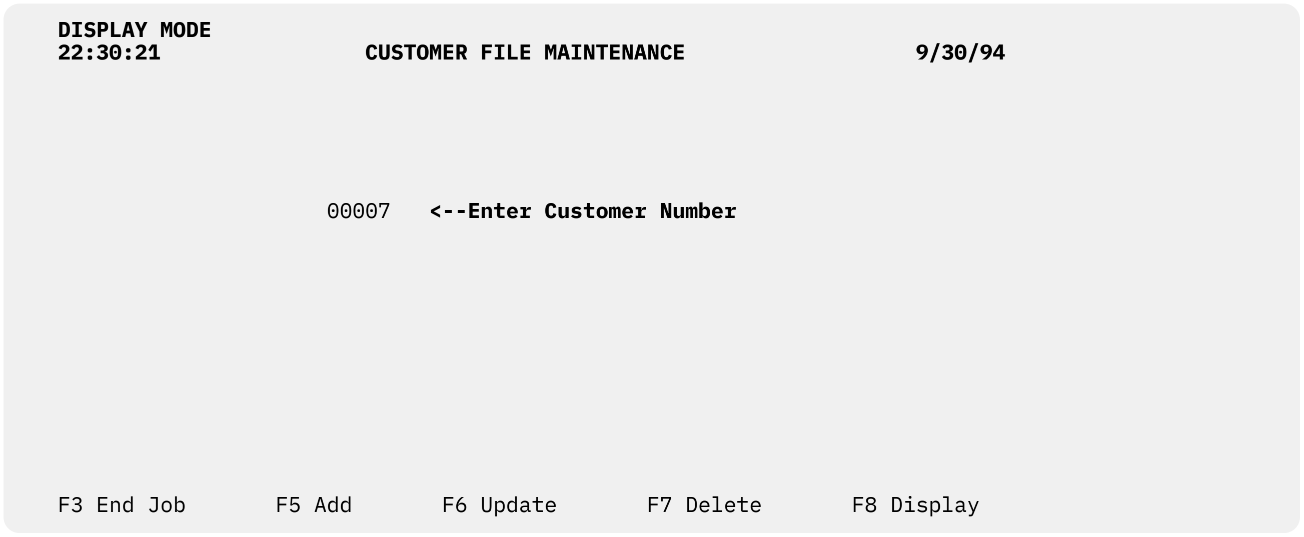
This program maintains a customer master file for additions, changes, and deletions. The program can also be used for inquiry.

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.

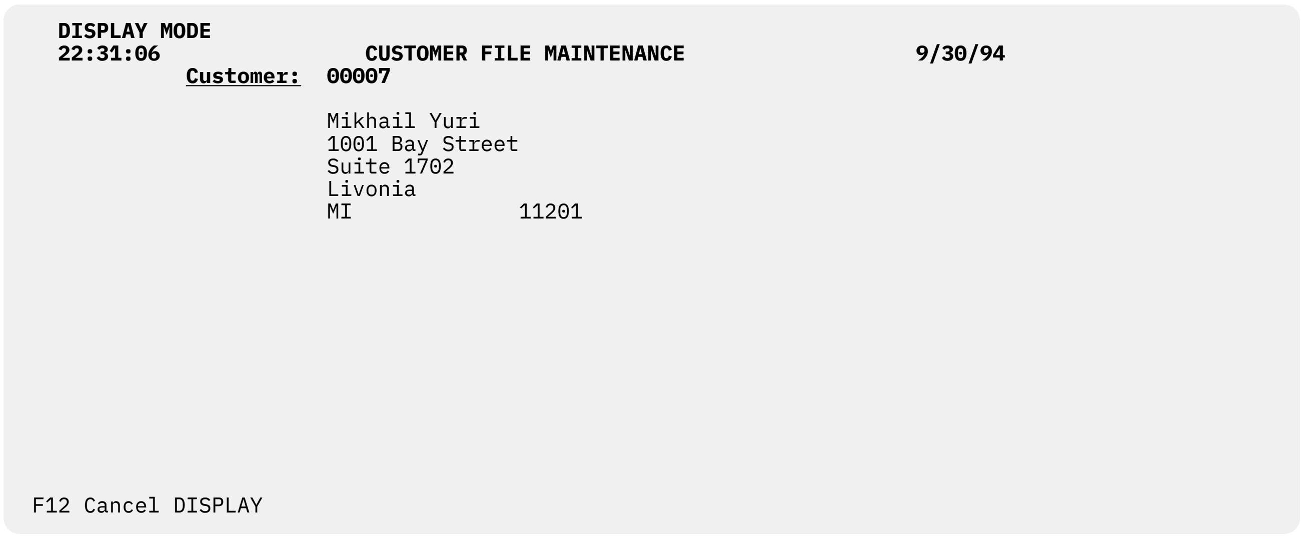
In Figure 201 on page 403, the workstation user responds to the prompt by entering customer number 00007 to display the customer record.



```
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 201. 'Customer File Maintenance' Display Mode prompt screen

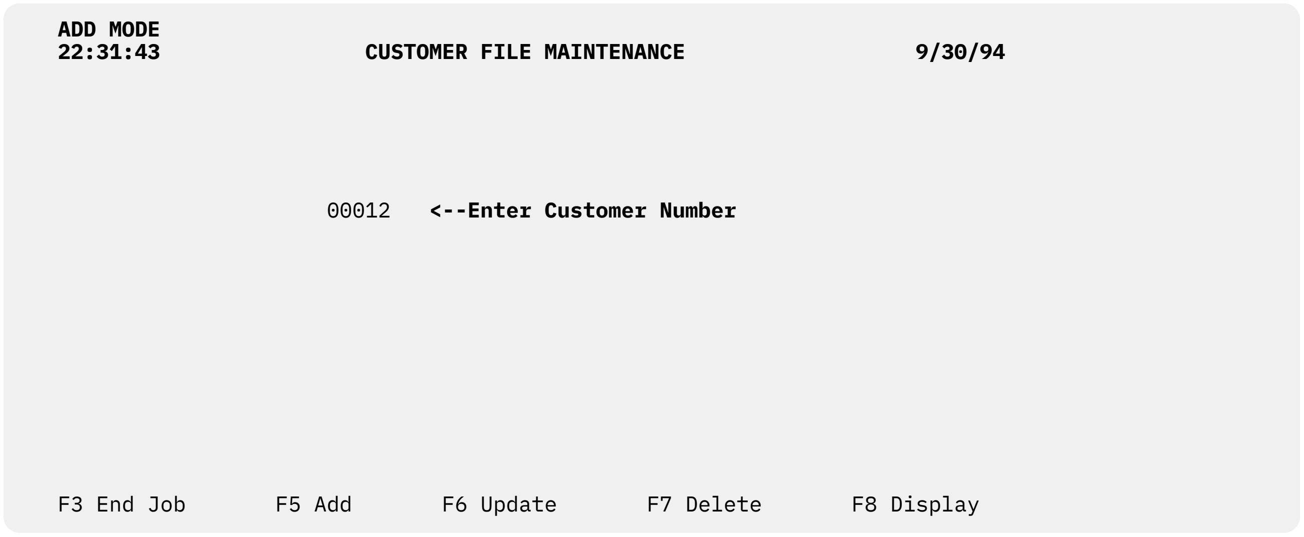
Because the customer record for customer number 00007 exists in the Master File, the data is displayed as show in Figure 202 on page 404.



```
  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 202. 'Customer File Maintenance' Display Mode screen

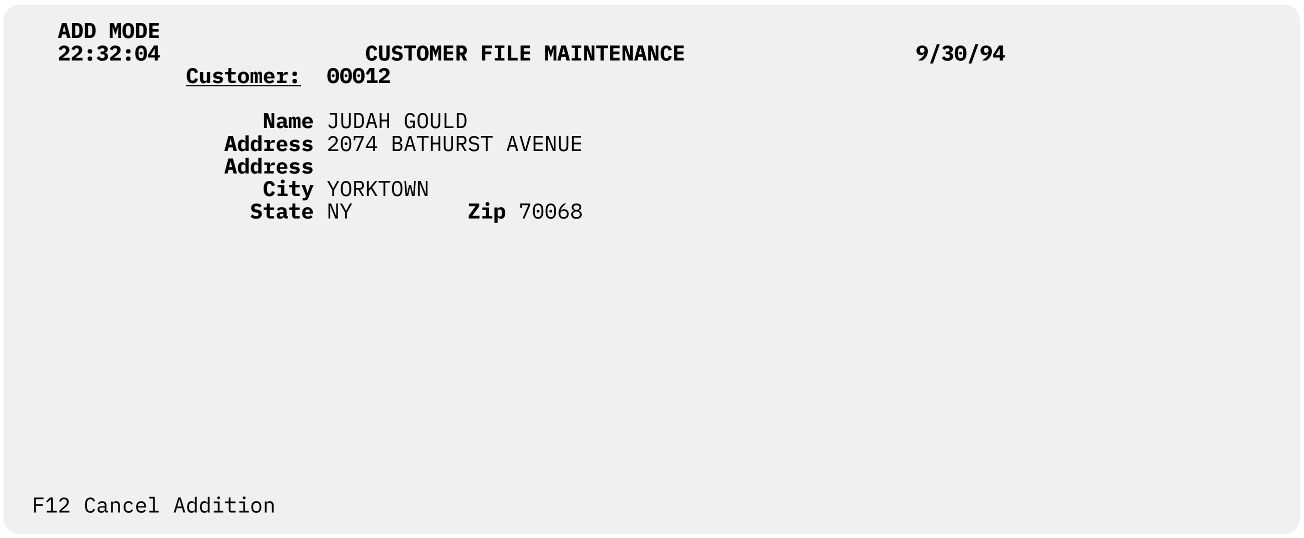
The workstation user responds to the add prompt by entering a new customer number as shown in Figure 203 on page 404.



```
  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 203. 'Customer File Maintenance' Add Mode prompt screen

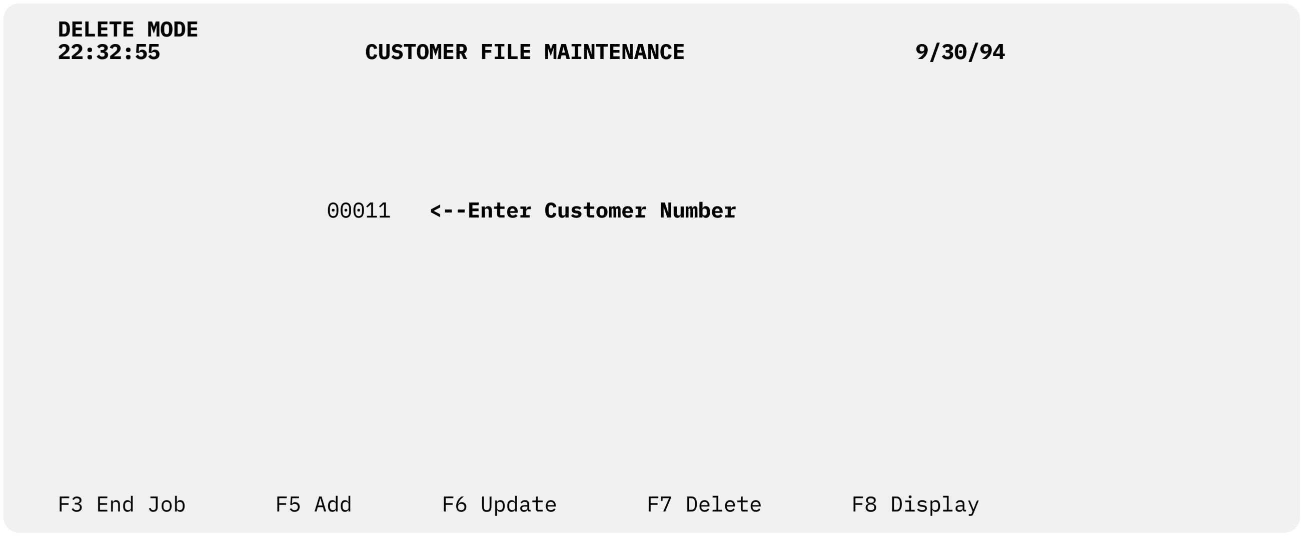
In Figure 204 on page 405 a new customer is added to the Customer Master File.



```
  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 204. 'Customer File Maintenance' Add Mode prompt screen

The workstation user responds to the delete prompt by entering a customer number as shown in Figure 205 on page 405.



```
  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 205. 'Customer File Maintenance' Delete Mode prompt screen

The workstation user responds to the update prompt by entering a customer number as shown in Figure 206 on page 406.

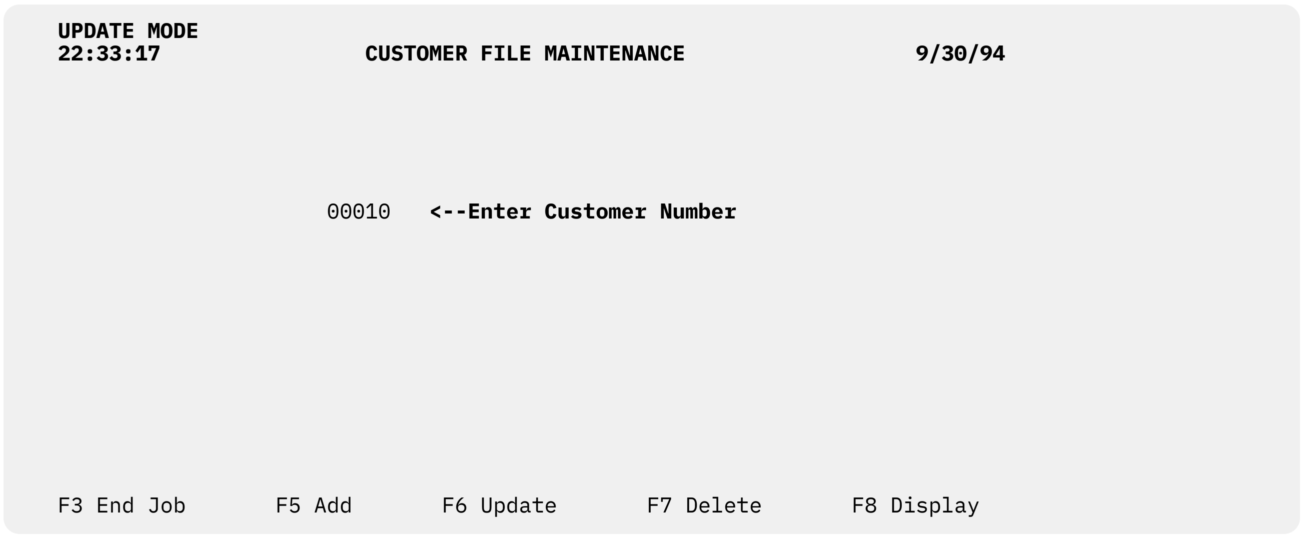

```
UPDATE MODE
22:33:17               CUSTOMER FILE MAINTENANCE                 9/30/94




                     00010   <--Enter Customer Number








F3 End Job       F5 Add       F6 Update       F7 Delete       F8 Display
```

Figure 206. '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 207. 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.

SCHZIPD: DDS for a Display Device File

```
A*****************************************************************
A*     FILE NAME:  SCHZIPD                                       *
A*  RELATED PGMS:  SCHZIP                                        *
A* RELATED FILES:  CUSMSTL2   (LOGICAL FILE)                     *
A*   DESCRIPTION:  THIS IS THE DISPLAY FILE SCHZIPD. IT HAS 6    *
A*                 RECORD FORMATS.                               *
A*****************************************************************
A                                      REF(CUSMSTL2)
A                                      CHGINPDFT(CS)
A                                      PRINT(QSYSPRT)
A                                      INDARA
A                                      CA03(03 'END OF JOB')
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'
A            ZIP       R        O  4 14DSPATR(HI)
A                                  7  4'Customer Name'
A                                      DSPATR(HI UL)
A                                  7 27'A/R Balance'
A                                      DSPATR(HI UL)
```

Figure 208. DDS for display device file SCHZIPD

The DDS for the SCHZIPD 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 IBM i 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)                       *
       //                 SCHZIPD (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 records by   *
       //                 zip code.                                     *
       //                 Roll up key can be used to look at another    *
       //                 page. PF3 us used to quit the program.        *
       //****************************************************************
      Fcusmstl2  if   e           k disk
      Fschzipd   cf   e             workstn sfile(subfile:recnum)
      F                                     indds(indicators)


       // Field definitions:
      D recnum          s              5p 0
      D recordFound     s               n

      D indicators      ds
      D    exitKey                      n   overlay(indicators:3)
      D    restartKey                   n   overlay(indicators:4)
      D    sflClear                     n   overlay(indicators:55)
      D    zipNotFound                  n   overlay(indicators:61)
      D    rollupKey                    n   overlay(indicators:95)

       // Key list definitions:
      C     cstkey        klist
      C                   kfld                    zip
```

Figure 209. Source for module SCHZIP

```
      //*******************************************************************
       //    MAINLINE                                                     *
       //*******************************************************************

       /free

        // Write out initial menu
        write foot1;
        write head;
        exfmt prompt;

        // loop until PF03 is pressed
        dow not exitKey;
           setll cstkey cmlrec2;
           recordFound = %equal(cusmstl2);
           if recordFound;
              exsr ProcessSubfile;
           endif;

           // Quit loop if PF03 was pressed in the subfile display
           if exitKey;
              leave;
           endif;

           // If PF04 was pressed, then redo search with the same
           // zip code.
           if restartKey;
              iter;
           endif;

           // Prompt for new zip code.
           if not recordFound;
              // If we didn't find a zip code, don't write header
              // and footer again
              write foot1;
              write head;
           endif;
           zipNotFound = not recordFound;
           exfmt prompt;
        enddo;

        *inlr = *on;
        //*****************************************************************
        //    SUBROUTINE - ProcessSubfile                                *
        //    PURPOSE    - Process subfile and display it                *
        //*****************************************************************
        begsr ProcessSubfile;

           // Keep looping while roll up key is pressed
           dou not rollupKey;
              // Do we have more information to add to subfile?
              if not %eof(cusmstl2);
                 // Clear and fill subfile with customer data
                 exsr ClearSubfile;
                 exsr FillSubfile;
              endif;

              // Write out subfile and wait for response
              write foot2;
              exfmt subctl;
           enddo;

        endsr;  // end of subroutine ProcessSubfile



        //*****************************************************************
        //    SUBROUTINE - FillSubfile                                   *
        //    PURPOSE    - Fill subfile with customer records matching   *
        //                 specified zip code.                           *
        //*****************************************************************
        begsr FillSubfile;

           // Loop through all customer records with specified zip code
           recnum = 0;
           dou %eof(schzipd);
              // Read next record with specified zip code
              reade zip cmlrec2;
              if %eof(cusmstl2);
                 // If no more records, we're done
                 leavesr;
              endif;

              // Add information about this record to the subfile
              recnum = recnum + 1;
              write subfile;
           enddo;
        endsr;  // end of subroutine FillSubfile;



        //*****************************************************************
        //    SUBROUTINE - ClearSubfile                                  *
        //    PURPOSE    - Clear subfile records                         *
        //*****************************************************************
        begsr ClearSubfile;

           sflClear = *on;
           write subctl;
           sflClear = *off;

        endsr;  // end of subroutine ClearSubfile

       /end-free
```

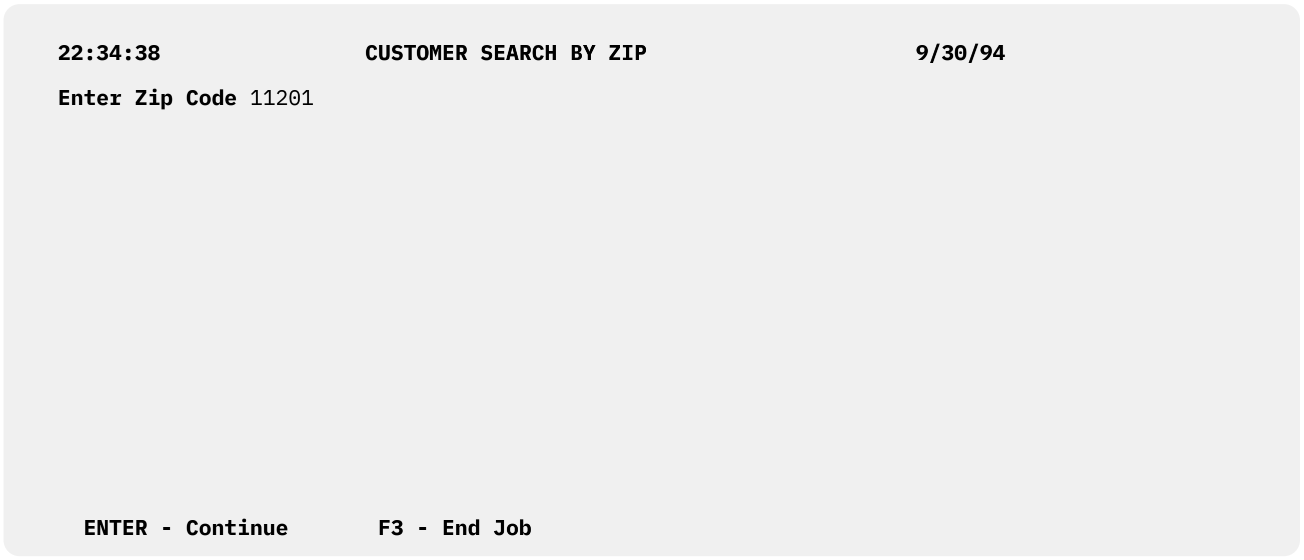
The file description specifications identify the disk file to be searched and the display device file to be used (SCHZIPD). The SFILE keyword for the WORKSTN file identifies the record format (SUBFILE) that is to be used as a subfile. The relative-record-number field (RECNUM) specified controls which 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.

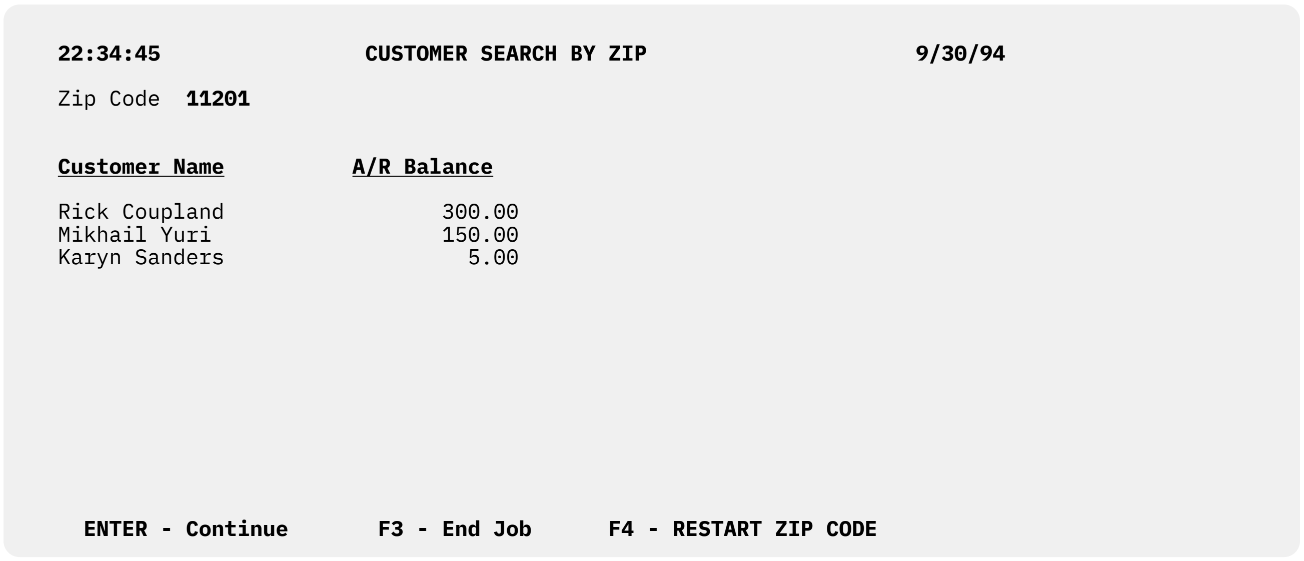
In Figure 210 on page 411, the user enters a zip code in response to the prompt.



```
22:34:38               CUSTOMER SEARCH BY ZIP                    9/30/94

Enter Zip Code 11201
















  ENTER - Continue       F3 - End Job
```

Figure 210. 'Customer Search by Zip' prompt screen

The subfile is written to the screen as shown in Figure 211 on page 412.



```
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 211. '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 212. 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.

SCHNAMD: DDS for a Display Device File

```
      A*****************************************************************
      A*     FILE NAME:  SCHNAMD                                       *
      A*  RELATED PGMS:  SCHNAM                                        *
      A* RELATED FILES:  CUSMSTL3   (LOGICAL FILE)                     *
      A*   DESCRIPTION:  THIS IS THE DISPLAY FILE SCHNAMD. IT HAS 7    *
      A*                 RECORD FORMATS.                               *
      A*****************************************************************
      A                                      REF(CUSMSTL3)
      A                                      CHGINPDFT(CS)
      A                                      PRINT(QSYSPRT)
      A                                      INDARA
      A                                      CA03(03 'END OF JOB')
      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'
```

Figure 213. DDS for display device file SCHNAMD

```
      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                                      CF04(04 'RESTART SEARCH NAME')
      A                                  5  4'Search Name'
      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                                  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)
```

The DDS for the SCHNAMD 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 IBM i 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)                       *
       //                 SCHNAMD (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 rollup key.   *
       //                 To display customer detail, enter 'X' beside  *
       //                 that customer and press enter.                *
       //                 To quit the program, press PF3.               *
       //****************************************************************

      Fcusmstl3  if   e           k disk
      Fschnamd   cf   e             workstn sfile(subfile:recnum)
      F                                     indds(indicators)

       // Field definitions:
      D recnum          s              5p 0

      D indicators      ds
      D    exitKey                      n   overlay(indicators:3)
      D    restartKey                   n   overlay(indicators:4)
      D    sflClear                     n   overlay(indicators:55)
      D    rollupKey                    n   overlay(indicators:95)

       // Key list definitions:
      C     cstkey        klist
      C                   kfld                    srcnam
      C     namekey       klist
      C                   kfld                    name
```

Figure 214. Source for module SCHNAM

```
      //*******************************************************************
      //   MAINLINE                                                        *
      //*******************************************************************

      /free

       write foot1;
       write head;
       exfmt prompt;

       // loop until exit key is pressed
       dow not exitKey;
          setll cstkey cusrec;
          exsr ProcessSubfile;
          exsr DisplayCustomerDetail;

          // If exit key pressed in subfile display, leave loop
          if exitKey;
             leave;
          endif;

          // If restart key pressed in subfile display, repeat loop
          if restartKey;
             iter;
          endif;

          write foot1;
          write head;
          exfmt prompt;

       enddo;

       *inlr = *on;



       //******************************************************************
       //  SUBROUTINE - ProcessSubfile                                     *
       //  PURPOSE    - Process subfile and display                        *
       //******************************************************************
       begsr ProcessSubfile;

          // Keep looping while roll up key is pressed
          dou not rollupKey;
             // Do we have more information to add to subfile?
             if not %eof(cusmstl3);
                // Clear and fill subfile with customer data
                exsr ClearSubfile;
                exsr FillSubfile;
             endif;

             // Write out subfile and wait for response
             write foot2;
             exfmt subctl;
          enddo;

       endsr;  // end of subroutine ProcessSubfile
       //*******************************************************************
       //   SUBROUTINE - FillSubfile                                        *
       //   PURPOSE    - Fill subfile                                       *
       //*******************************************************************
       begsr FillSubfile;

          // Loop through all customer records
          recnum = 0;
          dou %eof(schnamd);
             // Read next record
             read cusrec;
             if %eof(cusmstl3);
                // If no more records, we're done
                leavesr;
             endif;

             // Add information about this record to the subfile
             recnum = recnum + 1;
             sel = *blank;
             write subfile;
          enddo;

       endsr;  // end of subroutine FillSubfile;



       //*****************************************************************
       //    SUBROUTINE - ClearSubfile                                   *
       //    PURPOSE    - Clear subfile records                          *
       //*****************************************************************
       begsr ClearSubfile;

          sflClear = *on;
          write subctl;
          sflClear = *off;

       endsr;  // end of subroutine ClearSubfile
```

```
      //******************************************************************
      //  SUBROUTINE - DisplayCustomerDetail                             *
      //  PURPOSE    - Display selected customer records                 *
      //******************************************************************
      begsr DisplayCustomerDetail;

         // Loop through all changed record in subfile
         readc subfile;
         dow not %eof(schnamd);
            // Restart the display of requested customer records
            restartKey = *on;

            // Lookup customer record and display it
            chain namekey cusrec;
            exfmt cusdsp;

            // If exit key pressed, exit loop
            if exitKey;
               leave;
            endif;

            readc subfile;
         enddo;


      endsr;  // end of subroutine ChangeSubfile

     /end-free
```

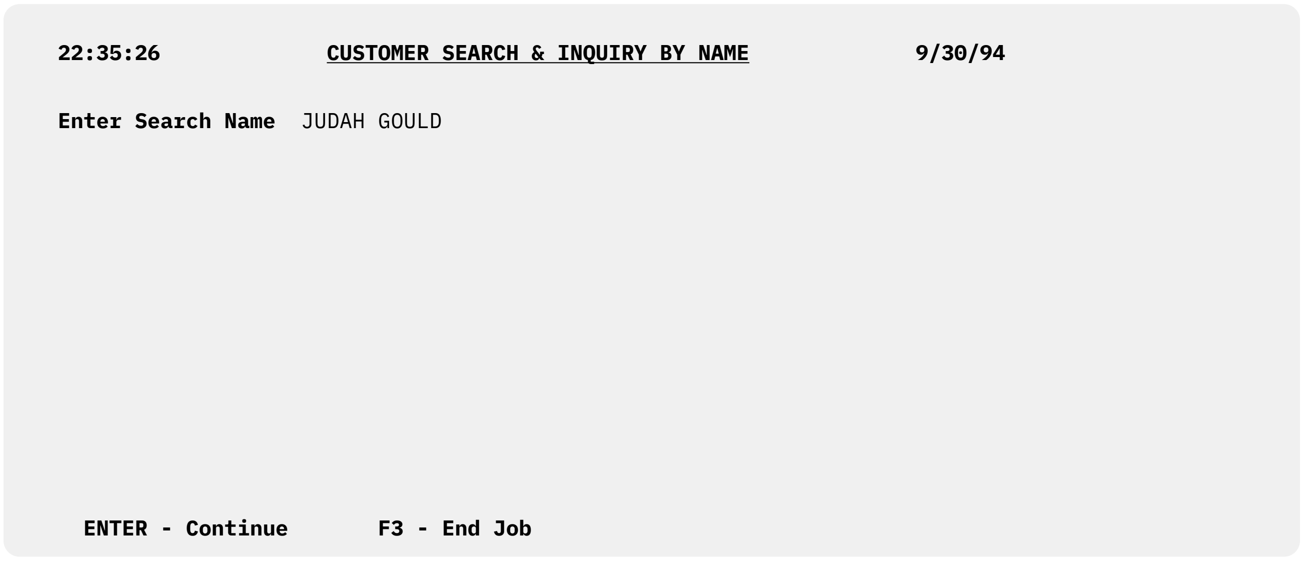
The file description specifications identify the disk file to be searched and the display device file to be used (SCHNAMD). The SFILE keyword for the WORKSTN file identifies the record format (SUBFILE) to be used as a subfile. The relative-record-number field (RECNUM) specifies which record within the 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.

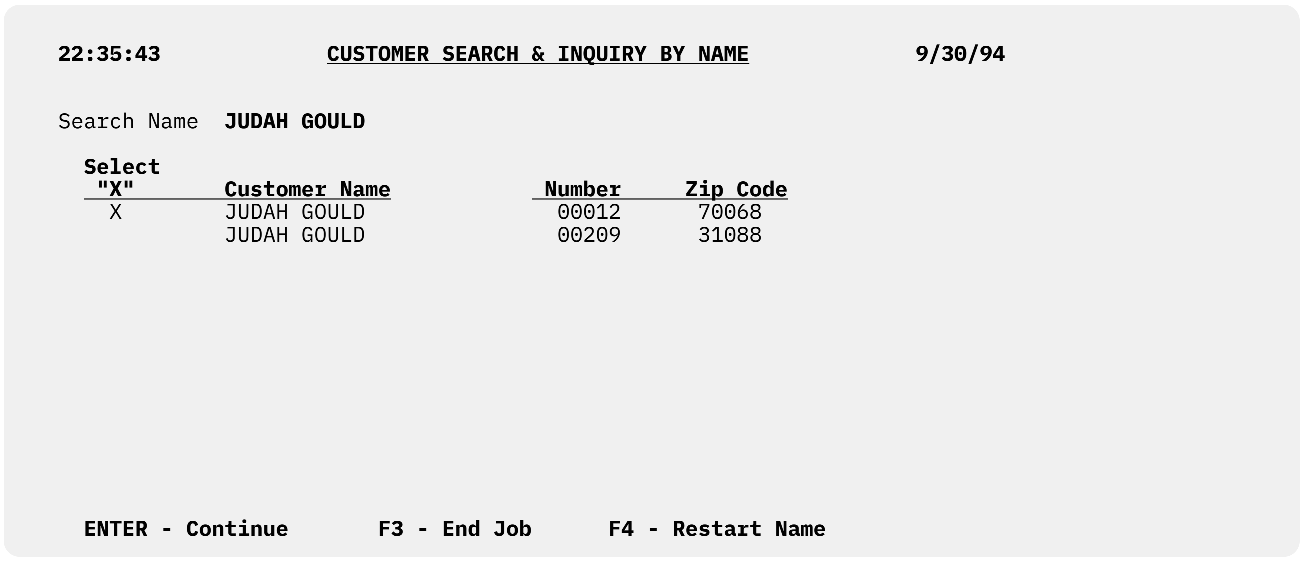
In Figure 215 on page 418, the user responds to the initial prompt by entering a customer name.



```
22:35:26            CUSTOMER SEARCH & INQUIRY BY NAME            9/30/94

Enter Search Name  JUDAH GOULD









  ENTER - Continue       F3 - End Job
```

Figure 215. 'Customer Search and Inquiry by Name' prompt screen

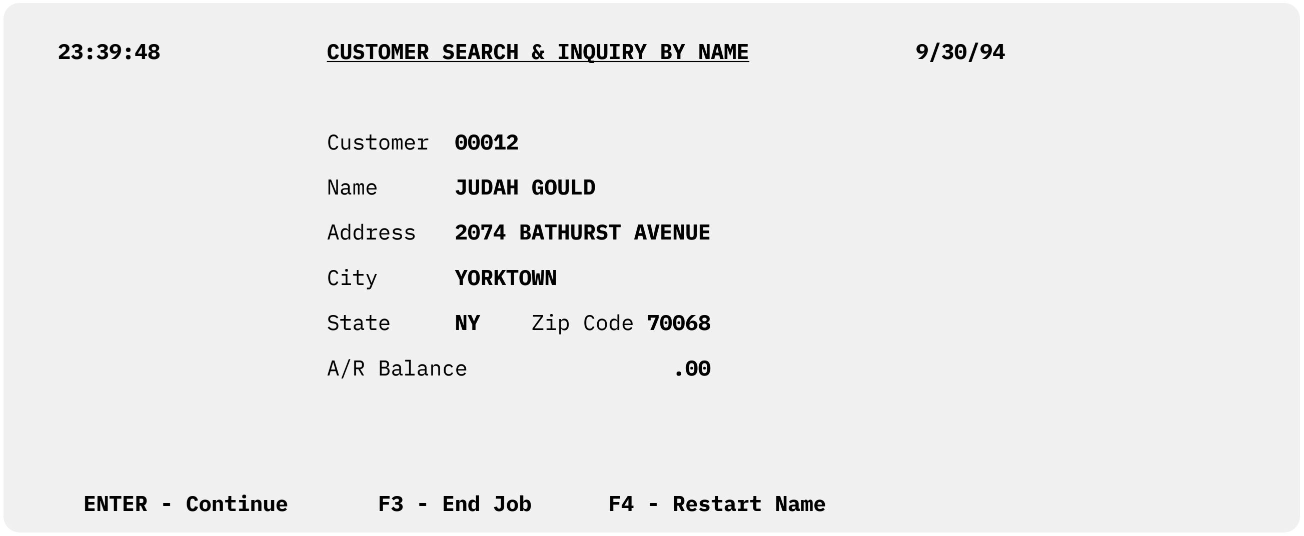
The user requests more information by entering an X as shown in Figure 216 on page 419.



```
 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 216. 'Customer Search and Inquiry by Name' information screen

The detailed information for the customer selected is shown in Figure 217 on page 419. 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 217. 'Customer Search and Inquiry by Name' detailed information screen

Chapter 7. Appendixes

Appendix A. Behavioral Differences Between OPM RPG/400 and ILE RPG for AS/400

The following lists note differences in the behavior of the OPM RPG/400 compiler and ILE RPG.

Compiling

1. If you specify CVTOPT(*NONE) in OPM RPG, all externally described fields that are of a type or with attributes not supported by RPG will be ignored. If you specify CVTOPT(*NONE) in ILE RPG, all externally described fields will be brought into the program with the same type as specified in the external description.
2. In RPG IV there is no dependency between DATEDIT and DECEDIT in the control specification.
3. Regarding the ILE RPG create commands (CRTBNDRPG and CRTRPGMOD):
   - The IGNDECERR parameter on the CRTRPGPGM command has been replaced by the FIXNBR parameter on the ILE RPG create commands. IGNDECDTA ignores any decimal data errors and continues with the next machine instruction. In some cases, this can cause fields to be updated with incorrect and sometimes unpredictable values. FIXNBR corrects the data in a predictable manner before it is used.
   - There is a new parameter, TRUNCNBR, for controlling whether numeric overflow is allowed.
   - There are no auto report features or commands in RPG IV.
   - You cannot request an MI listing from the compiler.
4. In a compiler listing, line numbers start at 1 and increment by 1 for each line of source or generated specifications, when the default OPTION(*NOSRCSTMT) is specified. If OPTION(*SRCSTMT) is specified, sequence numbers are printed instead of line numbers. Source IDs are numeric, that is, there are no more AA000100 line numbers for /COPY members or expanded DDS.
5. 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.
6. ILE RPG is more rigorous in detecting field overlap in data structures. For some calculation operations involving overlapping operands, ILE RPG issues a message while the OPM compiler does not.
7. In ILE RPG the word NOT cannot be used as a variable name. NOT is a special word that is used as an operator in expressions.
8. At compile time, the source is read using the CCSID specified by the TGTCCSID parameter, while for OPM RPG, the source is read using the CCSID of the job.

Running

1. The FREE operation is not supported by RPG IV. See Unsupported RPG III Features.
2. Certain MCH messages may appear in the job log that do not appear 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 nonbindable 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 if you used the special value of '*' for the call message queue and a value of greater than 0 for the stack offset.

4. ILE RPG does not interpret return codes that are not 0 or 1 for calls to programs or procedures that end without an exception.
5. When the cancel handler for an ILE RPG program receives control, it will set the system return code to 2. The cancel handler for an OPM RPG program does not modify the setting of the system return code.
6. 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. Note that this only applies to cycle-main procedures. Recursion is allowed for subprocedures.
7. When the cycle-main procedure of an ILE RPG module is cancelled from the program stack without reaching the part of the RPG cycle that checks *INLR, the *TERM processing will be done.

   When an OPM RPG program is cancelled from the program stack without reaching the part of the RPG cycle that checks *INLR, the *TERM processing will not be done.

   *TERM processing includes the following:

   - opened global files are closed
   - data areas locked by the program are released
   - the program or module is set so that program variables will be refreshed for the next call.

   If *INLR was on when an ILE RPG cycle-main procedure was canceled, *INLR will not be on for the next call to the procedure, and the RPG cycle will begin normally with *INIT.

   If *INLR was on when an OPM RPG program was cancelled, it will still be on for the next call to the program and the RPG cycle will proceed to *TERM without performing the *DETC part of the cycle.
8. If decimal-data errors occur during the initialization of a zoned-decimal or packed-decimal subfield, then the reset values (those values use to restore the subfield with the RESET operation) may not be valid. For example, it may be that the subfield was not initialized, or that it was overlaid on another initialized subfield of a different type. If a RESET operation is attempted for that subfield, then in OPM RPG/400, a decimal-data error would occur. However, a RESET to the same subfield in ILE RPG will complete successfully; after the RESET, the subfield has the same invalid value. As a result, attempts to use the value will get a decimal data error.
9. In ILE RPG, positions 254-263 of the program status data structure (PSDS) contain the user name of the originating job. In OPM RPG, these positions reflect the current user profile. The current user profile in ILE RPG can be found in positions 358-367.

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 left in the job log by both OPM RPG and ILE RPG. However, in ILE RPG, if you have coded an error indicator, 'E' extender, or *PSSR error routine, then the function check will not appear.

   You should remove any code that deletes function checks, since the presence of the indicator, 'E' extender, or *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 that fills the PSDS after 80 bytes is costly to obtain. If the PSDS is not coded, or is too short to contain the date and time the program started, these two values will not be available in a formatted dump. All other PSDS values will be available, no matter how long the PSDS is.
5. The prefix for ILE RPG inquiry messages is RNQ, so if you use the default reply list, you must add RNQ entries similar to your existing RPG entries.
6. In OPM, if a CL program calls your RPG program followed by a MONMSG, and the RPG program receives a notify or status message, the CL MONMSG will not handle the notify or status message. If you are calling ILE RPG from ILE CL and both are in the same activation group, the ILE CL MONMSG will handle the notify or status message and the RPG procedure will halt immediately without an RPG

error message being issued. For more information see "Problems when ILE CL Monitors for Notify and Status Messages" on page 316.

7. When displaying a variable using the ILE source debugger, you will get unreliable results if:
   - the ILE RPG program uses an externally described file and
   - the variable is defined in the data base file but not referenced in the ILE RPG program.
8. If your RPG III program has a parameter-mismatch problem (for example, it passes a parameter of length 10 to a program that expects a parameter of length 20, and the called program changes all 20 bytes), your program will experience a storage corruption problem. This problem may not always result in an error, if the storage that is corrupted is not important to the running of the program.

   When this program is converted to RPG IV, the layout of storage may be different, so that the corrupted storage is used by the program. This can cause an unexpected exception to occur, for example exception MCH3601 on a file operation such as a SETLL. If you experience mysterious errors that seem unrelated to your application, you should check the parameters of all your call operations to ensure the parameters all have the correct length.

I/O

1. In ILE RPG you can read a record in a file opened for update, and created or overridden with SHARE(*YES), and then update this locked record in another program that has opened the same file for update.
2. If a program performs a sequential input operation, and it results in an end-of-file condition, the normal operation is for any subsequent sequential input operation in the same module to immediately result in an end-of-file condition without any physical input request to the database. However, if the file is shared, the RPG runtime will always send a physical input request to the database, and the input operation will be successful if the file has been repositioned by a call to another program or module using the shared file.
3. You cannot modify the MR indicator using the MOVE or SETON operations. (RPG III only prevents using SETON with MR.)
4. The File Type entry on the File specification no longer dictates the type of I/O operations that 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.
5. ILE RPG will allow record blocking even if the COMMIT keyword is specified on the file description specification.
6. 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.
7. In RPG IV, you do not have to code an actual number for the number of devices that 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 that 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, the total number of program devices that can be acquired by the program cannot be different from the maximum number of devices defined in the device file. OPM RPG/400 allows this through the NUM option.
8. In ILE RPG, the ACQ and REL operation codes can be used with single device files.
9. In ILE RPG, 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.

10. When a referential constraint error occurs in OPM RPG/400, the status code is set to "01299" (I/O error). In ILE RPG, the status code is set to "01022", "01222", or "01299", depending on the type of referential constraint error that occurs:

    - If data management is not able to allocate a record due to a referential constraint error, a CPF502E notify message is issued. ILE RPG will set the status code to "01222" and OPM RPG/400 will set the status code to "01299".

      If you have no error indicator, 'E' extender, or INFSR error subroutine, ILE RPG 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 that has caused it to issue either a CPF503A, CPF502D, or CPF502F notify message, ILE RPG will set the status code to "01022" and OPM RPG/400 will set the status code to "01299".

      If you have no error indicator, 'E' extender, or INFSR error subroutine, ILE RPG 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 to set the status code to "01299".
11. In ILE RPG, the database-specific feedback section of the INFDS is updated regardless of the outcome of the I/O operation. In OPM RPG/400, this feedback section is not updated if the record-not-found condition is encountered.
12. ILE RPG 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 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 and OPM RPG/400 set the status code to "01211". ILE RPG detects this situation when data management issues a CPF501B notify message and places it in the job log.
    - When handling WORKSTN files and trying to do I/O to a device that has not been acquired or defined, both ILE and OPM RPG will set the status to "01281". ILE RPG detects this situation when data management issues a CPF5068 escape message and places it in the job log.
13. 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 DDS features are used that cause more than one search argument to match a given key in the file.

    For example, if ABSVAL is used on a numeric key, both -1 and 1 would succeed as search arguments for a key in the file with a value of 1. Using the hexadecimal collating sequence, a search argument of -1 will not succeed for an actual key of 1.

    ILE RPG does key comparisons using *HEX collating sequence only for pre-V3R1 DDM files. See "Using Pre-V3R1 DDM Files" on page 368 for more information.
14. ILE RPG allows the To File and the From File specified for prerun-time arrays and tables to be different. In OPM RPG, both file names must be the same; if they are different the diagnostic message QRG3038 is issued.
15. When translation of a RAF-Controlled file is specified, the results using ILE RPG may differ from OPM RPG/400, depending on the translation table. This is due to the different sequence of operations. In OPM RPG/400 the sequence is: retrieve record, translate and compare; in ILE RPG the sequence is: translate, compare and retrieve record.
16. The RPG/400 compiler considers the DELET operation to be an output operation. If an update-capable record format has a DELET operation and a CLEAR or RESET operation, but no UPDAT operation, the RPG/400 compiler will clear or reset the fields of the record format, but the ILE RPG compiler will not clear or reset the fields. To have the ILE RPG compiler clear or reset the fields, *ALL can be specified in Factor 2 of the operation, or an UPDATE operation can be added to the program.

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 that 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, 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:

   - A matching shift-in for each shift-out (that is, the shift-out and shift-in control characters should be balanced)
   - An even number (minimally two) between the shift-in and the shift-out
   - The absence of an 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 that fail the transparency check performed by the OPM RPG compiler, such programs will get compilation errors in ILE RPG.
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 character literal will not be a transparent literal if:

   - 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 a transparency check should be performed by the compiler

   In ILE RPG, 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 control characters.
3. In ILE RPG, if you want to avoid the checking of literals for shift-out characters (that is, you do not want a shift-out character to be interpreted as such), then you should specify the entire literal as a hexadecimal literal. For example, if you have a literal 'AoB' where 'o' represents a shift-out control character, 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 428 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

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 also optionally include specification templates 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 on the status of the conversion if you specified a log file on the command. You can also obtain a conversion report that 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. In some cases, there may be a small amount of code that 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 that cannot be converted, see "Resolving Conversion Problems" on page 442.

File Considerations

The Conversion Aid operates on file members. This section presents information on different aspects of files that must be taken into consideration when using the Conversion Aid.

Source Member Types

Table 82 on page 426 lists the various source member types, indicates whether the member type can be converted, and indicates the output source member type.

Table 82. 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 does not support auto report source members.

For more information on converting auto report source members, see "Converting Auto Report Source Members" on page 435.

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 218 on page 427. The recommended record length of 112 characters also corresponds to the maximum amount of information that fits on a line of a compiler listing.

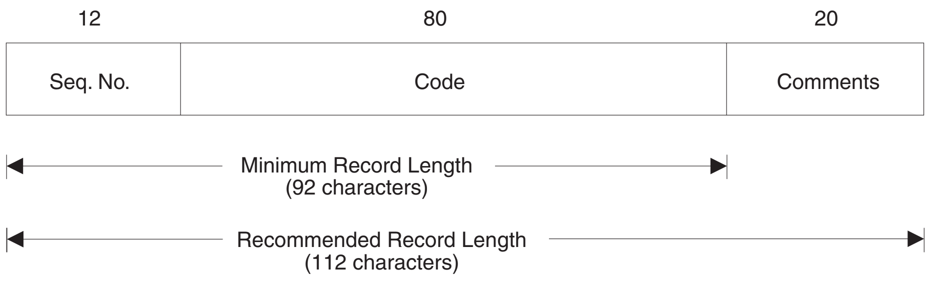


Figure 218. RPG IV Record Length Breakdown

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 225 on page 441 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 that is accessed by the Conversion Aid.

For information on using the log file see “Using the Log File” on page 440.

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 that contains the source file and source members
- *CHANGE authority to the new library that 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 percent 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 conversion 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 explains how to convert source programs to the RPG IV format. It discusses 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 data area in the RPG IV format. Refer to the chapter on control specifications in *IBM Rational Development Studio for i: ILE RPG 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 427 and "Using the Log File" on page 440.
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 426.
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 429 for more information.
5. Check the log file or the error report for any errors. For more information, see "Analyzing Your Conversion" on page 438.
6. If there are errors, correct them and go to step "4" on page 428.
7. If there are no errors, create your program. For information on how to create ILE RPG programs, see "Creating a Program with the CRTBNDRPG Command" on page 101.
8. If your converted source member still has compilation problems, these 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 442 for further information.

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. "CVTRPGSRC Parameters and Their Default Values Grouped by Function" on page 429 shows the parameters of the command based on their function.

CVTRPGSRC Parameters and Their Default Values Grouped by Function

Table 83. File Identification

Parameter	Description
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

Table 84. Conversion Processing

Parameter	Description
TOMBR	If *NONE is specified, then no file members are saved
EXPCPY(*NO)	Determines if/COPY statements are included in converted output
INSRTPL(*NO)	Indicates if specification templates are to be included in converted output

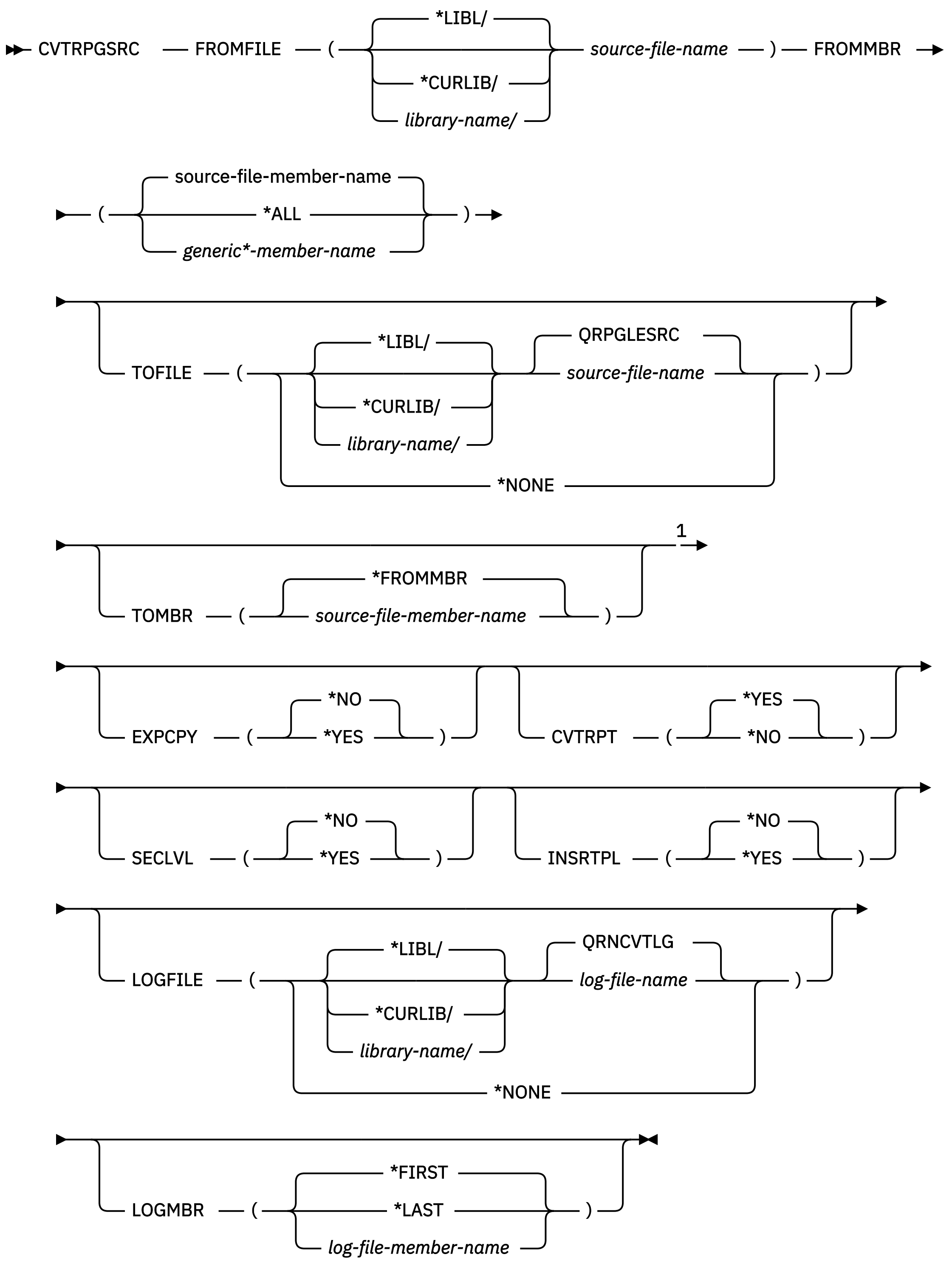
Table 85. Conversion Feedback

Parameter	Description
CVTRPT(*YES)	Determines whether to produce conversion report
SECLVL(*NO)	Determines 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

CVTRPGSRC command syntax

The syntax for the CVTRPGSRC command is shown below.

Job: B,I Pgm: B,I REXX: B,I Exec



Notes:

[1] 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 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 second-level text is printed in the conversion report in the message summary section.

<u>*NO</u>
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 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.

<u>*NO</u>
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 CPYF 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 files 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 82 on page 426).

Converting Some Members in a File

If you need to convert only some members that are in a source physical file, and these members 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 that 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 438.

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 that no conversion has occurred. However, no information regarding these members is placed in the report.

The conversion report includes the following information:

- CVTRPGSRC command options
- Source section that 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.)

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.

In particular, you may want to check the auto report spooled file for an error message 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.

Auto Report supports compile-time data in /COPY members. RPG IV does not support this. If you are keeping compile-time data in /COPY members so that several programs can use the data, consider moving the compile-time data to a user-space and accessing it through the user-space APIs.

Converting Source Members with Embedded SQL

When converting code that contains embedded SQL and the SQL code is continued over multiple lines, the following will occur:

- If there are continuation lines but column 74 is blank, the line is simply copied to the ILE member.

  Note: This could be a problem if column 74 happens to be a blank character inside a character string.
- If column 74 is not blank, all of the SQL code from that line to the /END-EXEC will be concatenated and copied to the ILE member filling up all 80 columns. If this occurs:
  - Any comments in column 75 on, will be ignored.
  - Any embedded comment lines (C*) will be copied to the ILE member before the concatenated code is copied.
  - Problems could arise if DBCS literals are split.

  If you do not want this concatenation and re-formatting to occur, ensure that column 74 is blank.

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 218 on page 427.) 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 219 on page 436 shows the source of the RPG III version.

```
     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
     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 219. 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 220 on page 437.

```
 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 220. 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 standalone 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 that 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 the 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 221 on page 439 shows the command summary for a sample conversion.

```
5769WDS V5R2M0  020719 RN         IBM ILE RPG    ISERIES1    08/15/02 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 221. Command Summary of Sample Conversion Report

The source section includes lines that 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 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 222 on page 439 shows the source section for the sample conversion.

```
5769WDS V5R2M0  020719 RN         IBM ILE RPG                  ISERIES1    08/15/02 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 222. Sample Source Section of Conversion Report

The message summary of the listing shows you the different messages that were issued. If you specify SECLVL(*YES), second-level messages will appear in the message summary. Figure 223 on page 440 shows the messages section for the sample conversion, including second-level messages.

```
5769WDS V5R2M0  020719 RN         IBM ILE RPG                             ISERIES1    08/15/02 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 for AS/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 223. 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 224 on page 440 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 224. 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 the 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 225 on page 441 shows the DDS for this file.

Use the following CPYF 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.

```
CPYF FROMFILE(QRPGLE/QARNCVTLG) TOFILE(MYLIB/QRNCVTLG)
     CRTFILE(*YES)
```

```
      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 225. 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) prior to performing the conversion if the type is RPT or RPT38.

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 that identifies most of these types of problems.

For further information on converting auto report members, see "Converting Auto Report Source Members" on page 435. For further information on differences between RPG III and RPG IV, see "Appendix A. Behavioral Differences Between OPM RPG/400 and ILE RPG for AS/400" on page 421.

Converting the FREE operation code

To replace the function of the FREE operation, you must first determine why the FREE operation was being used.

- If the FREE operation was being used to ensure that the program would be initialized on the next call to the program, change the called program so that it may be called with a special parameter (or no parameter), indicating that it should simply set on LR and return. Then, instead of coding the FREE operation, call the program with the special "free" parameter.

```
 * RPG III coding
C                     CALL 'MYPGM'
C                     PARM           P1
...
C                     FREE 'MYPGM'
...
C                     CALL 'MYPGM'
C                     PARM           P1

* Replacement RPG IV coding for the "reresolve" function of FREE

C                   call      MYPGM_var
C                   parm                    p1
 ...
 * Cause MYPGM to initialize on the next call
C                   call      MYPGM_VAR
 ...
```

```
C                   call      MYPGM_var
C                   parm                    p1

* Modified version of MYPGM.  It ends itself when it is called with no parameters.
D                SDS
D PARMS             *PARMS
C     *ENTRY        PLIST
C                   PARM                    NAME             10
c     PARMS         IFEQ      0
C                   SETON                                        LR
C                   RETURN
C                   ENDIF
 ...
```

- If the FREE operation was being used to cause the program containing the FREE operation to resolve to the program again on the next call to the program, then you can change your calling program so that you call using a character variable; to cause your called program to be resolved again, you must use the character variable to call a different program at the point where you would do your FREE; then when you use the character variable on the next CALL operation, the system would perform the resolve to your program again. Create a very quick-running program to be called for the "FREE" function, such as an ILE RPG program that simply has a RETURN operation.

```
 * RPG III coding
C                     CALL 'MYPGM'
C                     PARM            P1
...
C                     FREE 'MYPGM'
...
C                     CALL 'MYPGM'
C                     PARM            P1

* Replacement RPG IV coding for the "reresolve" function of FREE

D MYPGM_var       s             21a   INZ('MYPGM')
C                   call      MYPGM_var
C                   parm                    p1
 ...
 * Cause a reresolve to MYPGM for the next call
C                   eval      MYPGM_var = 'MYLIB/FREEPGM'
C                   call      MYPGM_VAR
C                   reset                   MYPGM_var
 ...
C                   call      MYPGM_var
C                   parm                    p1
```

To replace the function of the DEBUG operation, use an interactive debugger. For information on program debugging see "Debugging Programs" on page 241.

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 errors 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 435 for the RPG III source member TEST1. If you compare the placement of the data structure DS1 in Figure 219 on page 436 and in Figure 220 on page 437, you can see that the data structure DS1 was moved so that it precedes 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. This copy member is included in source TEST2. Figure 226 on page 444 and Figure 227 on page 444 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 226. RPG III 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 227. RPG III 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 located. As a result, all source lines in the copy member COPYDS1 will get a "source record is out of sequence" error. 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.

There are two methods of correcting 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 compiler listing and the *IBM Rational Development Studio for i: ILE RPG 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 435, you are not allowed to define a standalone array and a data structure subfield with the same name in RPG IV. Therefore, as shown in the example TEST1 (Figure 220 on page 437), the Conversion Aid must merge these two definitions. However, if 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 220 on page 437), if compile-time arrays are merged with data structure subfield definitions, the loading of array data may be affected. 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:

- In data structure subfields or program-described file fields

```
      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 228. RPG III /COPY file with input fields only

- In renaming an externally described data structure field or 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 229. RPG III Source with a renamed field

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 228 on page 445 to the code shown in Figure 230 on page 446. 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 230. RPG IV source after converting source with input fields only

Remember that you have two ways of correcting 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.

- The merging of an array with an externally described DS subfield
- The renaming and initializing of an externally described DS subfield

These problems 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 compiler listing and the *IBM Rational Development Studio for i: ILE RPG Reference* to correct them.

Merging an Array with an Externally Described DS Subfield

As mentioned earlier, you are not allowed to define a standalone 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 231 on page 446 is included twice in Figure 232 on page 446. It is included once as a standalone array and once in the externally described data structure EXTREC. When converted, the RPG IV source generated is shown in Figure 233 on page 447. This code will not compile since ARRAY is defined twice. In order to correct this problem, delete the standalone array and add a subfield with the keywords to data structure DSONE as shown in Figure 234 on page 447.

```
      A           R RECORD
      A             CHARACTER     10
      A             ARRAY         10
```

Figure 231. DDS for external data structure

```
      E                       ARRAY      10  1
      IDSONE     E DSEXTREC
      C            CHAR      DSPLY
      C                       SETON                     LR
```

Figure 232. RPG III source using external data structure with array

```
     D ARRAY           S               1    DIM(10)
     D DSONE         E DS                   EXTNAME(EXTREC)
     C     CHAR          DSPLY
     C                   SETON                                      LR
```

Figure 233. RPG IV source with two definitions for the array

```
     D DSONE         E DS                   EXTNAME(EXTREC)
     D  ARRAY        E                      DIM(10)
     C     CHAR          DSPLY
     C                   SETON                                      LR
```

Figure 234. Corrected RPG IV source with a single definition for the array

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 235 on page 447. The converted source also contains two source lines, as shown in Figure 236 on page 447. This use of two source lines for a field 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 237 on page 447, 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 235. RPG III source with renamed and initialized external subfield

```
     D DSONE         E DS                   EXTNAME(EXTREC)
     D  CHAR         E                      EXTFLD(CHARACTER)
     D  CHAR         E                      INZ('XYZ')
     C     CHAR          DSPLY
     C                   SETON                                      LR
```

Figure 236. RPG IV source with two definitions for renamed subfield

```
     D DSONE         E DS                   EXTNAME(EXTREC)
     D  CHAR         E                      EXTFLD(CHARACTER) INZ('XYZ')
     C     CHAR          DSPLY
     C                   SETON                                      LR
```

Figure 237. Corrected RPG IV source with a single definition

Run-time Differences

If you have prerun-time arrays that overlap in data structures, the order of loading these arrays at run time may be different in RPG III and in RPG IV. This difference in order can cause the data in the overlapping section to differ. The order in which the arrays are loaded is the order in which they are encountered in the source. This order may have changed when the arrays were been merged with the subfields during conversion.

In general, you should avoid situations where an application consists of OPM and ILE programs that 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. Refer to “Program Creation Strategies” on page 71 or *ILE Concepts* for further information.

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

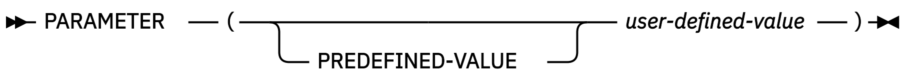
For information on the Create Program and Create Service Program commands, see the *CL and APIs* section of the *Programming* category in the **IBM i Information Center** at this Web site - http://www.ibm.com/systems/i/infocenter/.

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:



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.

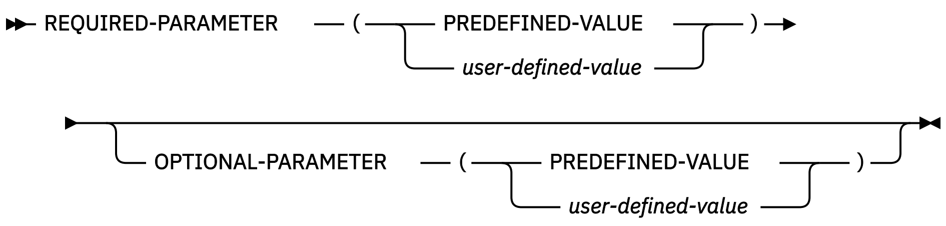
The ──►◄ symbol indicates the end of the syntax diagram.

The ───► symbol indicates that the statement syntax is continued on the next line.

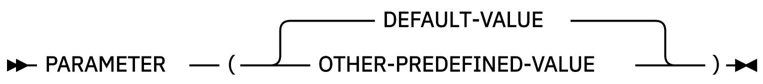
The ►─── symbol indicates that a statement is continued from the previous line.

The ──(──)── symbol indicates that the parameter or value must be entered in parentheses.

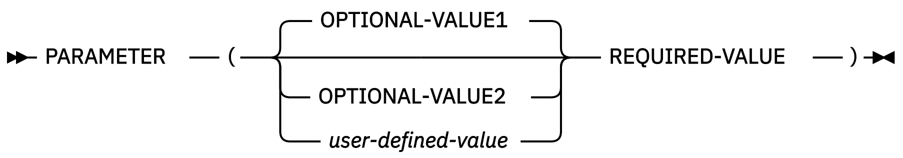
Required parameters appear on the base line and must be entered. **Optional parameters** appear below the base line and do not need to be entered. In the following sample, you must enter REQUIRED-PARAMETER and a value for it, but you do not need to enter OPTIONAL-PARAMETER or a value for it.



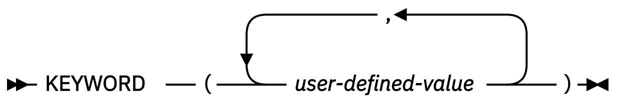
Default values appear above the base line and do not need to be entered. They are used when you do not specify a parameter. In the following sample, you can enter DEFAULT-VALUE, OTHER-PREDEFINED-VALUE, or nothing. If you enter nothing, DEFAULT-VALUE is assumed.



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).



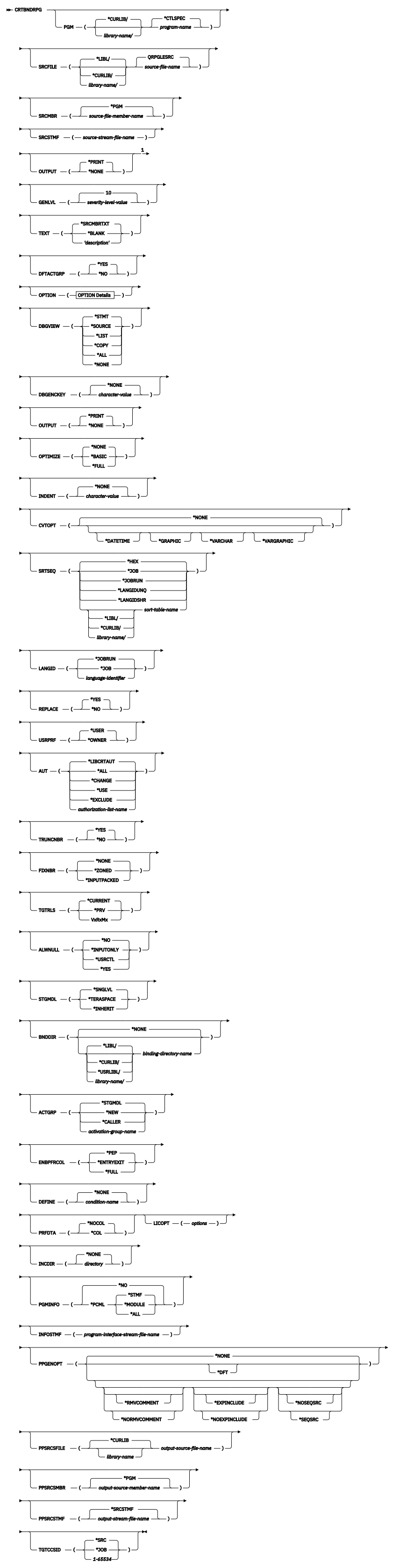
Repeated values can be specified for some parameters. The comma (**,**) in the following sample indicates that each *user-defined-value* must be separated by a comma.



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.

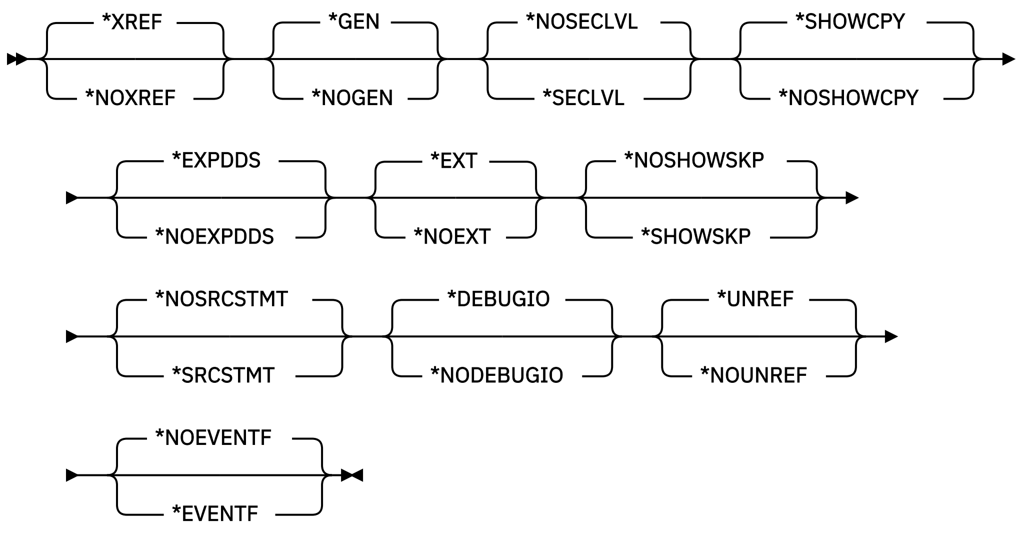
Job: B,I Pgm: B,I REXX: B,I Exec


CRTBNDRPG
PGM (*CURLIB/ library-name/ *CTLSPEC program-name)
SRCFILE (*LIBL/ *CURLIB/ library-name/ QRPGLESRC source-file-name)
SRCMBR (*PGM source-file-member-name)
SRCSTMF (source-stream-file-name)
OUTPUT (*PRINT *NONE) 1
GENLVL (10 severity-level-value)
TEXT (*SRCMBRTXT *BLANK 'description')
DFTACTGRP (*YES *NO)
OPTION (OPTION Details)
DBGVIEW (*STMT *SOURCE *LIST *COPY *ALL *NONE)
DBGENCKEY (*NONE character-value)
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 *INPUTPACKED)
TGTRLS (*CURRENT *PRV VxRxMx)
ALWNULL (*NO *INPUTONLY *USRCTL *YES)
STGMDL (*SNGLVL *TERASPACE *INHERIT)
BNDDIR (*NONE *LIBL/ *CURLIB/ *USRLIBL/ library-name/ binding-directory-name)
ACTGRP (*STGMDL *NEW *CALLER activation-group-name)
ENBPFRCOL (*PEP *ENTRYEXIT *FULL)
DEFINE (*NONE condition-name)
PRFDTA (*NOCOL *COL)
LICOPT (options)
INCDIR (*NONE directory)
PGMINFO (*NO *PCML *STMF *MODULE *ALL)
INFOSTMF (program-interface-stream-file-name)
PPGENOPT (*NONE *DFT *RMVCOMMENT *NORMVCOMMENT *EXPINCLUDE *NOEXPINCLUDE *NOSEQSRC *SEQSRC)
PPSRCFILE (*CURLIB library-name output-source-file-name)
PPSRCMBR (*PGM output-source-member-name)
PPSRCSTMF (*SRCSTMF output-stream-file-name)
TGTCCSID (*SRC *JOB 1-65534)

Notes:

1 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 IBM i 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.

SRCSTMF
Specifies the path name of the stream file containing the ILE RPG source code to be compiled.

The path name can be either absolutely or relatively qualified. An absolute path name starts with '/'; a relative path name starts with a character other than '/'.

If absolutely-qualified, the path name is complete. If relatively-qualified, the path name is completed by appending the job's current working directory to the path name.

The SRCMBR and SRCFILE parameters cannot be specified with the SRCSTMF parameter.

GENLVL
Controls the creation of the program object. The program object is created if all errors encountered during compilation have a severity level less than or equal to the generation severity level specified.

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. For errors greater than severity 20, the program object will not be generated.

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.

***YES**
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 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 you cannot use the BNDDIR or ACTGRP parameters when creating this program. In addition, any call operation in your source must call a program and not a procedure.

DFTACTGRP(*YES) is useful when attempting to move an application on a program-by-program basis to ILE RPG.

***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.

<u>*GEN</u>
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.

<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 and display key field information.

***NOEXPDDS**
Do not show the expansion of externally described files in the listing or display key field information.

<u>*EXT</u>
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.

<u>*NOSHOWSKP</u>
Do not show ignored statements in the source part of the listing. The compiler ignores statements as a result of /IF, /ELSEIF or /ELSE directives.

***SHOWSKP**
Show all statements in the source part of the listing, regardless of whether or not the compiler has skipped them.

***NOSRCSTMT**
Line Numbers in the listing are assigned sequentially; these numbers are used when debugging using statement numbers. Line Numbers are shown on the left-most column of the listing. The source IDs and SEU Sequence Numbers are shown on the two right-most columns of the listing.

***SRCSTMT**
Statement numbers for debugging are generated using SEU sequence numbers and source IDs as follows:

```
Statement_Number = source_ID * 1000000 + source_SEU_sequence_number
```

SEU Sequence Numbers are shown on the left-most column of the listing. Statement Numbers are shown on the right-most column of the listing; these numbers are used when debugging using statement numbers.

Note: When OPTION(*SRCSTMT) is specified, all sequence numbers in the source files must contain valid numeric values. If there are duplicate sequence numbers in the same source file, the behavior of the debugger may be unpredictable and statement numbers for diagnostic messages or cross reference entries may not be meaningful.

***DEBUGIO**
Generate breakpoints for all input and output specifications.

***NODEBUGIO**
Do not generate breakpoints for input and output specifications.

***UNREF**
Unreferenced data items are included in the compiled module.

***NOUNREF**
Unreferenced data items are not included in the compiled module. This reduces the amount of storage used, allowing a larger program to be compiled. You cannot look at or assign to an unreferenced data item during debugging when the *NOUNREF option is chosen. The unreferenced data items still appear in the cross-reference listings produced by specifying OPTION(*XREF).

***NOEVENTF**
Do not create an Event File for use by Rational Developer for i. Rational Developer for i uses this file to provide error feedback integrated with the editor. An Event File is normally created when you create a module or program from within Rational Developer for i.

***EVENTF**
Create an Event File for use by Rational Developer for i. 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.

Rational Developer for i uses this file to provide error feedback integrated with the editor. An Event File is normally created when you create a module or program from within the Rational Developer for i.

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 or Statement Numbers of the compiler listing. Line Numbers are shown on the left-most column of the source section of the compiler listing when OPTION(*NOSRCSTMT) is specified. Statement Numbers are shown on the right-most column of the source section of the compiler listing when OPTION(*SRCSTMT) is specified.

***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, *EXPDDS, and *SRCSTMT are specified for the OPTION parameter.

Note: The listing view will not show any indentation that 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, *EXPDDS, and *SRCSTMT are specified for the OPTION parameter.

***NONE**
Disables all of the debug options for debugging the compiled program object.

DBGENCKEY
Specifies the encryption key to be used to encrypt program source that is embedded in debug views.

<u>*NONE</u>
No encryption key is specified.

character-value
Specify the key to be used to encrypt program source that is embedded in debug views stored in the module object. The length of the key can be between 1 and 16 bytes. A key of length 1 to 15 bytes will be padded to 16 bytes with blanks for the encryption. Specifying a key of length zero is the same as specifying *NONE.

If the key contains any characters which are not invariant over all code pages, it will be up to the user to ensure that the target system uses the same code page as the source system, otherwise the key may not match and the decryption may fail. If the encryption key must be entered on systems with differing code pages, it is recommended that the key be made of characters which are invariant for all EBCDIC code pages.

OUTPUT
Specifies if a compiler listing is generated.

<u>*PRINT</u>
Produces a compiler listing, consisting of the ILE RPG program source and all compile-time messages. The information contained in the listing is dependent on whether *XREF, *SECLVL, *SHOWCPY, *EXPDDS, *EXT, *SHOWSKP, and *SRCSTMT are specified for the OPTION parameter.

***NONE**
Do not generate the compiler listing.

OPTIMIZE
Specifies the level of optimization, if any, of the program.

<u>*NONE</u>
Generated code is not optimized. This is the fastest in terms of translation time. It allows you to display and modify variables while in debug mode.

***BASIC**
Some optimization is performed on the generated code. This allows user variables to be displayed but not modified while the program is in debug mode.

***FULL**
Optimization which generates the most efficient code. Translation time is the longest. In debug mode, 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 that you request here will not be reflected in the listing debug view that 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 the string 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.

<u>*NONE</u>
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-weight table. This special value is used in conjunction with the LANGID parameter to determine the proper sort sequence table.

***LANGIDSHR**
Use a shared-weight 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.

If you want to use the SRTSEQ and LANGID parameters to determine the alternate collating sequence, you must also specify ALTSEQ(*EXT) on the control specification.

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.

***JOBRUN**
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.

***YES**
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 to 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 its value, you must delete the program and recompile using the new value (or, if the constituent *MODULE objects exist, you may choose to invoke the CRTPGM command).

***USER**
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 objects 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 users 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 and APIs* section of the *Programming* category in the **IBM i Information Center** at this Web site - http://www.ibm.com/systems/i/infocenter/.

***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; that is, 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.

Note: The TRUNCNBR option does not apply to calculations performed within expressions. (Expressions are found in the Extended-Factor 2 field.) If overflow occurs for these calculations, an error will always occur. In addition, overflow is always signalled for any operation where the value that is assigned to an integer or unsigned field is out of range.

***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 decimal data that is not valid is fixed by the compiler.

***NONE**

Indicates that decimal data that is not valid will result in decimal data errors during run time if used.

***ZONED**

Zoned-decimal data that 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.

***INPUTPACKED**

Indicates that if packed decimal data that is not valid is encountered while processing input specifications, the internal variable will be set to zero.

TGTRLS

Specifies the release level of the operating system on which you intend to use the object being created. In the examples given for the *CURRENT and *PRV values, and when specifying the *target-release* value, the format VxRxMx is used to specify the release, where Vx is the version, Rx is the release, and Mx is the modification level. For example, V2R3M0 is version 2, release 3, modification level 0.

Valid values for this parameter change every release. The possible values are:

<u>*CURRENT</u>

The object is to be used on the release of the operating system currently running on your system. For example, if V2R3M5 is running on the system, *CURRENT means that you intend to use the object on a system with V2R3M5 installed. You can also use the object on a system with any subsequent release of the operating system installed.

Note: If V2R3M5 is running on the system, and the object is to be used on a system with V2R3M0 installed, specify TGTRLS(V2R3M0), not TGTRLS(*CURRENT).

***PRV**

The object is to be used on the previous release with modification level 0 of the operating system. For example, if V2R3M5 is running on your system, *PRV means you intend to use the object on a system with V2R2M0 installed. You can also use the object on a system with any subsequent release of the operating system installed.

target-release

Specify the release in the format VxRxMx. You can use the object on a system with the specified release or with any subsequent release of the operating system installed.

Valid values depend on the current version, release, and modification level, and they change with each new release. If you specify a *target-release* that is earlier than the earliest release level supported by this command, an error message is sent indicating the earliest supported release.

Note: The current version of the command may support options that are not available in previous releases of the command. If the command is used to create objects that are to be used on a previous release, it will be processed by the compiler appropriate to that release, and any unsupported options will not be recognized. The compiler will not necessarily issue any warnings regarding options that it is unable to process.

ALWNULL

Specifies how the ILE RPG module will be allowed to use records containing null-capable fields from externally described database files.

<u>*NO</u>

Specifies that the ILE RPG module 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 module and a data-mapping error occurs.

***INPUTONLY**

Specifies that the ILE RPG module can successfully read records with null-capable fields containing null values from externally-described input-only database files. When a record containing null values is retrieved, no data-mapping errors occur and the database default values are placed into any fields that contain null values. The module cannot do any of the following:

- use null-capable key fields
- create or update records containing null-capable fields

- determine whether a null-capable field is actually null while the module is running
- set a null-capable field to be null.

***USRCTL**
Specifies that the ILE RPG module can read, write, and update records with null values from externally-described database files. Records with null keys can be retrieved using keyed operations. The module can determine whether a null-capable field is actually null, and it can set a null-capable field to be null for output or update. The programmer is responsible for ensuring that fields containing null values are used correctly within the module.

***YES**
Same as *INPUTONLY.

STGMDL
Specifies the storage model attribute of the program.

<u>*SNGLVL</u>
The program is created with single-level storage model. When a single-level storage model program is activated and run, it is supplied single-level storage for automatic and static storage. A single-level storage program runs only in a single-level storage activation group.

***TERASPACE**
The program is created with teraspace storage model. When a teraspace storage model program is activated and run, it is supplied teraspace storage for automatic and static storage. A teraspace storage program runs only in a teraspace storage activation group.

***INHERIT**
The program is created with inherit storage model. When activated, the program adopts the storage model of the activation group into which it is activated. An equivalent view is that it inherits the storage model of its caller. When the *INHERIT storage model is selected, *CALLER must be specified for the Activation group (ACTGRP) parameter.

BNDDIR
Specifies the list of binding directories that are used in symbol resolution.

<u>*NONE</u>
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.

<u>*STGMDL</u>
If STGMDL(*TERASPACE) is specified, the program will be activated into the QILETS activation group when it is called. Otherwise, this program will be activated into the QILE activation group when it is called.

***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.

ENBPFRCOL
Specifies whether performance collection is enabled.

***PEP**
Performance statistics are gathered on the entry and exit of the program entry procedure only. This applies to the actual program-entry procedure for a program, not to the main procedure of the modules within the program. This is the default.

***NEW**
When this program is called, it is activated into a new activation group.

***ENTRYEXIT**
Performance statistics are gathered on the entry and exit of all procedures of the program.

***FULL**
Performance statistics are gathered on entry and exit of all procedures. Also, statistics are gathered before and after each call to an external procedure.

DEFINE
Specifies condition names that are defined before the compilation begins. Using the parameter DEFINE(condition-name) is equivalent to coding the /DEFINE condition-name directive on the first line of the source file.

***NONE**
No condition names are defined. This is the default.

condition-name
Up to 32 condition names can be specified. Each name can be up to 50 characters long. The condition names will be considered to be defined at the start of compilation.

PRFDTA
Specifies the program profiling data attribute for the program. Program profiling is an advanced optimization technique used to reorder procedures and code within the procedures based on statistical data (profiling data).

***NOCOL**
This program is not enabled to collect profiling data. This is the default.

***COL**
The program is enabled to collect profiling data. *COL can be specified only when the optimization level of the module is *FULL, and when compiling with a target release of *CURRENT.

LICOPT
Specifies one or more Licensed Internal Code compile-time options. This parameter allows individual compile-time options to be selected, and is intended for the advanced programmer who understands the potential benefits and drawbacks of each selected type of compiler option.

INCDIR
Specifies one or more directories to add to the search path used by the compiler to find copy files.

The compiler will search the directories specified here if the relatively specified copy files in the source program can not be resolved by looking in the current directory.If the copy file cannot be found in the current directory or the directories specified in the INCDIR parameter, the directories specified in the RPGINCDIR environment variable will be searched, followed by the directory containing the main source file.

***NONE**
No directories are specified.

directory
Specify up to 32 directories in which to search for copy files.

PGMINFO Parameter:

This option specifies whether program interface information should be generated and where it should be generated. Specify the option values in the following order:

generate

Specifies whether program interface information should be generated. The possible values are:

<u>*NO</u>

Program interface information will not be generated.

[*PCML]

Specifies that PCML (Program Call Markup Language) should be generated. The generated PCML makes it easier for Java methods to call the procedures in this RPG module, with less Java code.

location

Specifies the location for the generated program information if the generate parameter is *PCML. The possible values are:

<u>*STMF</u>

Specifies that the program information should be generated into a stream file. The name of a stream file that will contain the generated information must be specified on the INFOSTMF option.

[*MODULE]

Specifies that the program information should be stored in the RPG module. For CRTBNDRPG, a module is created as the first step before creating a program.

[*ALL]

Specifies that the program information should be generated into a stream file and also stored in the module. The name of a stream file that will contain the generated information must be specified on the INFOSTMF option.

INFOSTMF

Specifies the path name of the stream file to contain the generated program interface information specifed on the PGMINFO option.

The path name can be either absolutely or relatively qualified. An absolute path name starts with '/'; a relative path name starts with a character other than '/'.

If absolutely-qualified, the path name is complete. If relatively-qualified, the path name is completed by appending the job's current working directory to the path name.

This parameter can only be specified when the PGMINFO parameter has a value other than *NO.

PPGENOPT

Specifies the preprocessor generation options to use when the source code is compiled.

The possible options are:

<u>*NONE</u>

Run the entire compiler against the source file. Do not copy the preprocessor output to a file.

***DFT**

Run the preprocessor against the input source. *RMVCOMMENT, *EXPINCLUDE and *NOSEQSRC will be used as the options for generating the preprocessor output. Use PPSRCFILE and PPSRCMBR to specify an output source file and member, or PPSRCSTMF to specify a stream file to contain the preprocessor output.

***RMVCOMMENT**

Remove comments, blank lines, and most directives during preprocessing. Retain only the RPG specifications and any directives necessary for the correct interpretation of the specifications..

***NORMVCOMMENT**

Preserve comments, blank lines and listing-control directives (for example /EJECT, /TITLE) during preprocessing. Transform source-control directives (for example /COPY, /IF) to comments during preprocessing.

***EXPINCLUDE**
Expand /INCLUDE directives in the generated output file.

***NOEXPINCLUDE**
/INCLUDE directives are placed unchanged in the generated output file.

Note: /COPY directives are always expanded

***SEQSRC**
If PPSRCFILE is specified, the generated output member has sequential sequence numbers, starting at 000001 and incremented by 000001.

***NOSEQSRC**
If PPSRCFILE is specified, the generated output member has the same sequence numbers as the original source read by the preprocessor

PPSRCFILE
Specifies the source file name and library for the preprocessor output.

source-file-name
Specify the name of the source file for the preprocessor output.

The possible library values are:

***CURLIB**
The preprocessor output is created in the current library. If a job does not have a current library, the preprocessor output file is created in the QGPL library.

library-name
Specify the name of the library for the preprocessor output.

PPSRCMBR
Specifies the name of the source file member for the preprocessor output.

***PGM**
The name supplied on the PGM parameter is used as the preprocessor output member name.

member-name
Specify the name of the member for the preprocessor output.

PPSRCSTMF
Specifies the path name of the stream file for the preprocessor output.

***SRCSTMF**
The path name supplied on the SRCSTMF parameter is used as the preprocessor output path name. The file will have the extension '.i'.

'path-name'
Specify the path name for the preprocessor output stream file.

The path name can be either absolutely or relatively-qualified. An absolute path name starts with '/'; a relative path name starts with a character other than '/'.

If absolutely-qualified, the path name is complete. If relatively-qualified, the path name is completed by appending the job's current working directory to the path name.

TGTCCSID
Specifies the CCSID that the compiler uses to read the source files.

***SRC**
The source is read in the CCSID of the primary source file, or if the file is an IFS file with an ASCII CCSID, the EBCDIC CCSID related to the ASCII CCSID. This is the default.

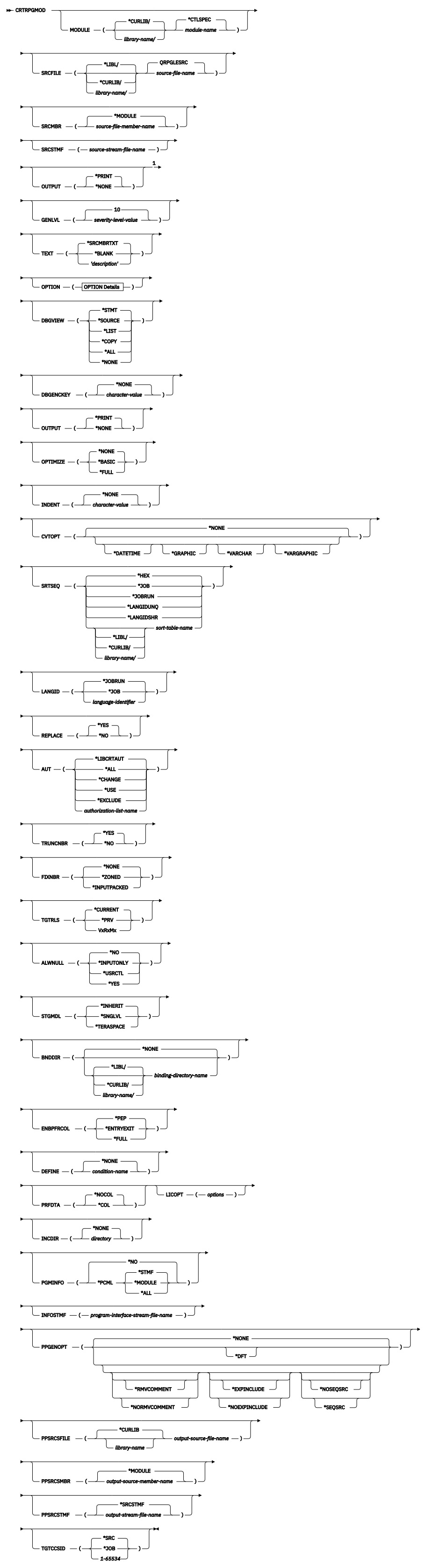
***JOB**
The source is read in the job CCSID. If the job CCSID is 65535, the source is read in the default CCSID of the job.

1-65534
The source is read in the specified CCSID. The CCSID must be a single-byte or mixed-byte EBCDIC CCSID.

CRTRPGMOD Command

The Create RPG Module (CRTRPGMOD) command compiles ILE RPG source code to create a module object (*MODULE). The entire syntax diagram for the CRTRPGMOD command is shown below.

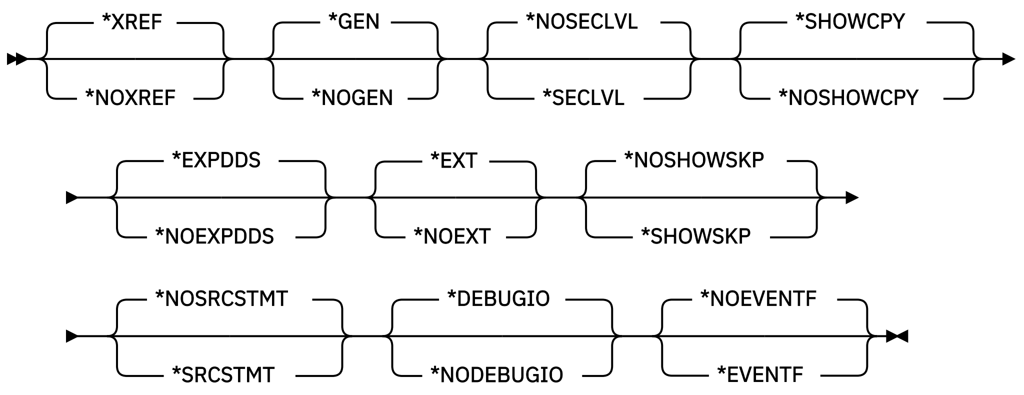
Job: B,I Pgm: B,I REXX: B,I Exec


CRTRPGMOD
MODULE (*CURLIB/ library-name/ *CTLSPEC module-name)
SRCFILE (*LIBL/ *CURLIB/ library-name/ QRPGLESRC source-file-name)
SRCMBR (*MODULE source-file-member-name)
SRCSTMF (source-stream-file-name)
1
OUTPUT (*PRINT *NONE)
GENLVL (10 severity-level-value)
TEXT (*SRCMBRTXT *BLANK 'description')
OPTION (OPTION Details)
DBGVIEW (*STMT *SOURCE *LIST *COPY *ALL *NONE)
DBGENCKEY (*NONE character-value)
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)
AUT (*LIBCRTAUT *ALL *CHANGE *USE *EXCLUDE authorization-list-name)
TRUNCNBR (*YES *NO)
FIXNBR (*NONE *ZONED *INPUTPACKED)
TGTRLS (*CURRENT *PRV VxRxMx)
ALWNULL (*NO *INPUTONLY *USRCTL *YES)
STGMDL (*INHERIT *SNGLVL *TERASPACE)
BNDDIR (*NONE *LIBL/ *CURLIB/ library-name/ binding-directory-name)
ENBPFRCOL (*PEP *ENTRYEXIT *FULL)
DEFINE (*NONE condition-name)
PRFDTA (*NOCOL *COL)
LICOPT (options)
INCDIR (*NONE directory)
PGMINFO (*NO *PCML *STMF *MODULE *ALL)
INFOSTMF (program-interface-stream-file-name)
PPGENOPT (*NONE *DFT *RMVCOMMENT *NORMVCOMMENT *EXPINCLUDE *NOEXPINCLUDE *NOSEQSRC *SEQSRC)
PPSRCSFILE (*CURLIB library-name output-source-file-name)
PPSRCSMBR (*MODULE output-source-member-name)
PPSRCSTMF (*SRCSTMF output-stream-file-name)
TGTCCSID (*SRC *JOB 1-65534)

Notes:

[1] All parameters preceding this point can be specified by position.

OPTION Details



Description of the CRTRPGMOD command

For a description of the parameters, options and variables for the CRTRPGMOD command see the corresponding description in the CRTBNDRPG command. They correspond exactly, except that those in CRTRPGMOD refer to modules and not to programs. (When looking at the CRTBNDRPG descriptions, keep in mind that CRTRPGMOD does not have the following parameters: ACTGRP, DFTACTGRP, USRPRF.)

The meaning of the STGMDL parameter for the CRTRPGMOD command differs from the meaning for the CRTBNDRPG command.

STGMDL
Specifies the type of storage to be used by the module.

***INHERIT**
The module is created with inherit storage model. An inherit storage model module can be bound into programs and service programs with a storage model of single-level, teraspace or inherit. The type of storage used for automatic and static storage for single-level and teraspace storage model programs matches the storage model of the object. An inherit storage model object will inherit the storage model of its caller.

***SNGLVL**
The module is created with single-level storage model. A single level storage model module can only be bound into programs and service programs that use single level storage. These programs and service programs use single-level storage for automatic and static storage.

***TERASPACE**
The module is created with teraspace storage model. A teraspace storage model module can only be bound into programs and service programs that use teraspace storage. These programs and service programs use teraspace storage for automatic and static storage.

A description of CRTRPGMOD is also 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.

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 compiler listing. See "Using a Compiler Listing" on page 107 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 86 on page 467 summarizes the keyword specifications and their associated compiler listing information.

Table 86. Sections of the Compiler Listing

Listing Section[1]	OPTION[2]	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
Skipped statements	*SHOWSKP	Source lines excluded by conditional compilation directives.
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
EVAL-CORR Summary	*XREF[4]	Summary of subfields for EVAL-CORR operations
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 errors[3]		Errors (if any) which occur during code generation phase.
Binding section[3]		Errors (if any) which occur during binding phase for CRTBNDRPG command

Table 86. Sections of the Compiler Listing (continued)

Listing Section[1]	OPTION[2]	Description

Note:

1. The information contained in the listing section is dependent on whether *SRCSTMT or *NOSRCSTMT is specified for the OPTION parameter. For details on how this information changes, see "*NOSRCSTMT Source Heading" and "*SRCSTMT Source Heading". *SRCSTMT allows you to request that the compiler use SEU sequence numbers and source IDs when generating statement numbers for debugging. Otherwise, statement numbers are associated with the Line Numbers of the listing and the numbers are assigned sequentially.
2. 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.
3. The sections containing the code generation errors and binding errors appear only if there are errors. There is no option to suppress these sections.
4. If OPTION(*XREF) is specified, the summary lists information about all subfields, whether or not they are handled by the EVAL-CORR operation. If OPTION(*NOXREF) is specified, the summary lists only information about subfields that are not handled by the EVAL-CORR operation. The EVAL-CORR summary section is not printed if there are no EVAL-CORR operations.

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 238 on page 469 illustrates how to interpret the Prologue section of the listing for the program MYSRC, which was compiled using the CRTBNDRPG command.

```
Title from first source line    1a
5722WDS V5R2M0  020719 RN   IBM ILE RPG    MYLIB/MYSRC  1b  ISERIES1 02/08/15 12:58:46
Page 1

  Command  . . . . . . . . . . . . :   CRTBNDRPG
    Issued by  . . . . . . . . . . :     MYUSERID
  Program  . . . . . . . . . . . . :   MYSRC    2
    Library  . . . . . . . . . . . :     MYLIB
  Text 'description' . . . . . . . :   Text specified on the Command
  Source Member  . . . . . . . . . :   MYSRC    3
  Source File  . . . . . . . . . . :   QRPGLESRC    4
    Library  . . . . . . . . . . . :     MYLIB
    CCSID  . . . . . . . . . . . . :     37
  Text 'description' . . . . . . . :   Text specified on the Source Member
  Last Change  . . . . . . . . . . :   98/07/27  12:50:13
  Generation severity level  . . . :   10
  Default activation group . . . . :   *NO
  Compiler options . . . . . . . . :   *XREF      *GEN       *SECLVL    *SHOWCPY    5
                                       *EXPDDS    *EXT       *SHOWSKP   *NOSRCSTMT
                                       *DEBUGIO   *NOEVENTF
  Debugging views  . . . . . . . . :   *ALL
  Output . . . . . . . . . . . . . :   *PRINT
  Optimization level . . . . . . . :   *NONE
  Source listing indentation . . . :   '| '  6
  Type conversion options  . . . . :   *NONE
  Sort sequence  . . . . . . . . . :   *HEX
  Language identifier  . . . . . . :   *JOBRUN
  Replace program  . . . . . . . . :   *YES
  User profile . . . . . . . . . . :   *USER
  Authority  . . . . . . . . . . . :   *LIBCRTAUT
  Truncate numeric . . . . . . . . :   *YES
  Fix numeric  . . . . . . . . . . :   *ZONED     *INPUTPACKED
  Target release . . . . . . . . . :   *CURRENT
  Allow null values  . . . . . . . :   *NO
  Binding directory  . . . . . . . :   BNDDIRA      BNDDIRB
    Library  . . . . . . . . . . . :     CMDLIBA      CMDLIBB
  Activation group . . . . . . . . :   CMDACTGRP
  Define condition names . . . . . :   ABC  7
                                       DEF
  Enable performance collection  . :   *PEP
  Profiling data . . . . . . . . . :   *NOCOL
  Generate program interface . . . :   *PCML
  Program interface stream file  . :   /home/mydir/MYSRC.pcml    8
  Include directory  . . . . . . . :   /projects/ABC Electronics Corporation/copy files/
prototypes
                                   :   /home/mydir     9
```

Figure 238. Sample Prologue for CRTBNDRPG

1 Page Heading
The page heading information includes the product information line 1b and the text supplied by a /TITLE directive 1a. "Customizing a Compiler Listing" on page 109 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.

7 Define condition names
Specifies the condition names that take effect before the source is read.

8
Specifies the IFS file that the PCML (Program Call Markup Language) is to be written to.

9
Specifies the directories that can be searched for /COPY or /INCLUDE files.

Source Section

The source section shows records that comprise the ILE RPG 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 also shows the conditional compilation process. All lines with /IF, /ELSEIF, /ELSE and /ENDIF directives and source lines selected by the /IF groups are printed and given a listing line number. If OPTION(*SHOWSKP) is specified, it shows all statements that have been excluded by the /IF, /ELSEIF, and /ELSE directives, and marks them with a '-------' in the column beside the statement. Line numbers in the listing are not incremented for excluded lines. All skipped statements are printed exactly as specified, but are not interpreted in any way. For example, an excluded statement with an /EJECT directive does not cause a page break. Similarly, /SPACE, /TITLE, /COPY and /EOF compiler directives are ignored if they are encountered in excluded lines. These statements are not shown if the default OPTION(*NOSHOWSKP) is specified; instead a message is printed giving the number of lines excluded.

The source section identifies any syntax errors in the source, and includes a match-field table, when appropriate.

If OPTION(*NOSRCSTMT) is specified, line numbers are printed sequentially on the left side of the listing to reflect the compiled source line numbers. Source IDs and SEU sequence numbers are printed on the right side of the listing to identify the source members and records respectively. For example, Figure 239 on page 470 shows a section of the listing with a /COPY statement in line 35. In the root source member, the next line is a DOWEQ operation. In the listing, however, the DOWEQ operation is on line 39. The three intervening lines shown in the listing are from the /COPY source member.

```
Line   <--------------------- Source Specifications ----------------------------------------------><---- Comments ----> Src Seq
Number ....1....+....2....+<-------- 26 - 35 -------->....4....+....5....+....6....+....7....+....8....+....9....+...10 Id  Number
    34 C                   MOVE                       '123'         BI_FLD1                                                 001500
    35 C/COPY MYCPY                                                                                              971104     001600
        *---------------------------------------------------------------------------------------------*
        * RPG member name  . . . . . :  MYCPY                                                         *                    5
        * External name  . . . . . . :  RPGGUIDE/QRPGLESRC(MYCPY)                                     *                    5
        * Last change  . . . . . . . :  98/07/24  16:20:04                                            *                    5
        * Text 'description' . . . . :  Text on copy member                                           *                    5
        *---------------------------------------------------------------------------------------------*
    36+C     Blue(1)       DSPLY                                                                                           5000100
    37+C     Green(4)      DSPLY                                                                                           5000200
    38+C     Red(2)        DSPLY                                                                                           5000300
    39 C     *in20         doweq                      *OFF                                                                  001700
```

*Figure 239. Sample Section of the Listing with OPTION(*NOSRCSTMT)*

If OPTION(*SRCSTMT) is specified, sequence numbers are printed on the left side of the listing to reflect the SEU sequence numbers. Statement numbers are printed on the right side of the listing. The statement number information is identical to the source ID and SEU sequence number information. For example, Figure 240 on page 471 shows a section of the listing that has a /COPY statement with sequence number 001600. The next line in the root source member is the same as the line with the next sequence number in the listing: sequence number 001700. The three intervening lines are assigned the SEU sequence numbers from the /COPY source member. The corresponding statement numbers are genereated from source IDs and SEU sequence numbers of the root and /COPY source members.

```
Seq    <--------------------- Source Specifications ---------------------------------------------><---- Comments ----> Statement
Number ....1....+....2....+<-------- 26 - 35 -------->....4....+....5....+....6....+....7....+....8....+....9....+...10 Number
001500 C                   MOVE                       '123'         BI_FLD1                                                 001500
001600 C/COPY MYCPY                                                                                              971104     001600
        *--------------------------------------------------------------------------------------------*
        * RPG member name  . . . . . :  MYCPY                                                        *                     5
        * External name  . . . . . . :  RPGGUIDE/QRPGLESRC(MYCPY)                                    *                     5
        * Last change  . . . . . . . :  98/07/24  16:20:04                                           *                     5
        * Text 'description' . . . . :  Text on copy member                                          *                     5
        *--------------------------------------------------------------------------------------------*
000100+C     Blue(1)       DSPLY                                                                                           5000100
000200+C     Green(4)      DSPLY                                                                                           5000200
000300+C     Red(2)        DSPLY                                                                                           5000300
001700 C     *in20         doweq                      *OFF                                                                  001700
```

*Figure 240. Sample Section of the Listing with OPTION(*SRCSTMT)*

Figure 241 on page 471 shows the entire source section for MYSRC with OPTION(*NOSRCSTMT) specified.

```
5769WDS V5R2M0  020719 RN        IBM ILE RPG            MYLIB/MYSRC            ISERIES1   02/08/15 14:21:00      Page    2
   1a
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 DFTACTGRP(*NO) ACTGRP('Srcactgrp') CCSID(*GRAPH:*SRC)                                                   980727     000100
     2 H OPTION(*NODEBUGIO)                                                                                      980727     000200
     3 H BNDDIR('SRCLIB1/BNDDIR1' : 'SRCLIB2/BNDDIR2' : '"ext.nam"')                                             971104     000300
     4 H ALTSEQ(*SRC)                                                                                            971104     000400
     5 H FIXNBR(*ZONED)                                                                                          980728     000500
     6 H TEXT('Text specified on the Control Specification')                                                     971104     000600
        *--------------------------------------------------------------------*   2
        * Compiler Options in Effect:                                        *
        *--------------------------------------------------------------------*
        *  Text 'description' . . . . . . . :                                *
        *                 Text specified on the Control Specification        *
        *  Generation severity level  . . . :   10                           *
        *  Default activation group . . . . :   *NO                          *
        *  Compiler options . . . . . . . . :   *XREF      *GEN              *
        *                                       *SECLVL    *SHOWCPY          *
        *                                       *EXPDDS    *EXT              *
        *                                       *SHOWSKP *NOSRCSTMT        *
        *                                       *NODEBUGIO   *NOEVENTF       *
        *  Optimization level . . . . . . . :   *NONE                        *
        *  Source listing indentation . . . :   '| '                         *
        *  Type conversion options  . . . . :   *NONE                        *
        *  Sort sequence  . . . . . . . . . :   *HEX                         *
        *  Language identifier  . . . . . . :   *JOBRUN                      *
        *  User profile . . . . . . . . . . :   *USER                        *
        *  Authority  . . . . . . . . . . . :   *LIBCRTAUT                   *
        *  Truncate numeric . . . . . . . . :   *YES                         *
        *  Fix numeric  . . . . . . . . . . :   *ZONED      *INPUTPACKED     *
        *  Allow null values  . . . . . . . :   *NO                          *
        *  Binding directory from Command . :   BNDDIRA      BNDDIRB         *
        *    Library  . . . . . . . . . . . :     CMDLIBA      CMDLIBB       *
        *  Binding directory from Source  . :   BNDDIR1      BNDDIR2         *
        *    Library  . . . . . . . . . . . :     SRCLIB1      SRCLIB2       *
        *                                       "ext.nam"                    *
        *                                         *LIBL                      *
        *  Activation group . . . . . . . . :   Srcactgrp                    *
        *  Enable performance collection  . :   *PEP                         *
        *  Profiling data . . . . . . . . . :   *NOCOL                       *
        *--------------------------------------------------------------------*
     7 FInFile    IF   E             DISK                                                                        971104     000700
        *--------------------------------------------------------------------------------------------*   3
        *                               RPG name        External name                                *
        * File name. . . . . . . . . :  INFILE          MYLIB/INFILE                                 *
        * Record format(s) . . . . . :  INREC           INREC                                        *
        *--------------------------------------------------------------------------------------------*
     8 FKEYL6     IF   E           K DISK                                                                        971104     000800
        *--------------------------------------------------------------------------------------------*
        *                               RPG name        External name                                *
        * File name. . . . . . . . . :  KEYL6           MYLIB/KEYL6                                  *
        * Record format(s) . . . . . :  REC1            REC1                                         *
        *                               REC2            REC2                                         *
        *--------------------------------------------------------------------------------------------*
     9 FOutfile   O    E             DISK                                                                        971104     000900
        *--------------------------------------------------------------------------------------------*
        *                               RPG name        External name                                *
        * File name. . . . . . . . . :  OUTFILE         MYLIB/OUTFILE                                *
        * Record format(s) . . . . . :  OUTREC          OUTREC                                       *
        *--------------------------------------------------------------------------------------------*
    10 D Blue            S              4    DIM(5) CTDATA PERRCD(1)                                             971104     001000
    11 D Green           S              2    DIM(5) ALT(Blue)                                                    971104     001100
    12 D Red             S              4    DIM(2) CTDATA PERRCD(1)                                             980727     001200
    13 D DSEXT1        E DS           100    PREFIX(BI_) INZ(*EXTDFT)                                            980727     001300
    14 D   FLD3        E                     INZ('111')                                                          980727     001400
```

Figure 241. Sample Source Part of the Listing

```
        *---------------------------------------------------------------------------------------------*  4                  1
        * Data structure . . . . . . :  DSEXT1                                                        *                     1
        * Prefix . . . . . . . . . . :  BI_ :    0                                                    *                     1
        * External format  . . . . . :  REC1 : MYLIB/DSEXT1                                           *                     1
        * Format text  . . . . . . . :  Record format description                                     *                     1
        *---------------------------------------------------------------------------------------------*                     1
  5
    15=D BI_FLD1                        5A   EXTFLD (FLD1)                        FLD1 description                         1000001
    16=D                                     INZ (*BLANK)                                                                  1000002
    17=D BI_FLD2                       10A   EXTFLD (FLD2)                        FLD2 description                         1000003
    18=D                                     INZ (*BLANK)                                                                  1000004
    19=D BI_FLD3                       18A   EXTFLD (FLD3)                        FLD3 description                         1000005
    20=D                                     INZ ('111')                                                                   1000006
    21=IINREC                                                                                                              2000001
        *---------------------------------------------------------------------------------------------*                     2
        * RPG record format  . . . . :  INREC                                                         *                     2
        * External format  . . . . . :  INREC : MYLIB/INFILE                                          *                     2
        *---------------------------------------------------------------------------------------------*                     2
    22=I                             A    1   25  FLDA                                                                     2000002
    23=I                             A   26   90  FLDB                                                                     2000003
    24=I                  13488 *VAR C   91  112  UCS2FLD                                                                  2000004
    25=IREC1                                                                                                               3000001
        *---------------------------------------------------------------------------------------------*                     3
        * RPG record format  . . . . :  REC1                                                          *                     3
        * External format  . . . . . :  REC1 : MYLIB/KEYL6                                            *                     3
        *---------------------------------------------------------------------------------------------*                     3
    26=I                        *ISO-D    1   10  FLD12                                                                    3000002
    27=I                             A   11   13  FLD13                                                                    3000003
    28=I                             A   14   17  FLD14                                                                    3000004
    29=I                             A   18   22  FLD15                                                                    3000005
    30=I                  13488      C   23   32  FLDC                                                                     3000006
    31=I                  13488 *VAR C   33   44  FLDCV                                                                    3000007
    32=I                  835        G   45   54  FLDG                                                                     3000008
    33=IREC2                                                                                                               4000001
        *---------------------------------------------------------------------------------------------*                     4
        * RPG record format  . . . . :  REC2                                                          *                     4
        * External format  . . . . . :  REC2 : MYLIB/KEYL6                                            *                     4
        *---------------------------------------------------------------------------------------------*                     4
    34=I                        *ISO-D    1   10  FLD22                                                                    4000002
    35=I                             A   11   13  FLD23                                                                    4000003
    36=I                             A   14   17  FLD24                                                                    4000004
    37=I                             A   18   22  FLD25                                                                    4000005
Line   <--------------------- Source Specifications ----------------------------------------------><---- Comments ----> Src Seq
Number ....1....+....2....+<-------- 26 - 35 -------->....4....+....5....+....6....+....7....+....8....+....9....+...10 Id  Number
    38 C                   MOVE                   '123'         BI_FLD1                                                 001500
    39 C/COPY MYCPY                                                                                              971104     001600
        *---------------------------------------------------------------------------------------------*  6
        * RPG member name  . . . . . :  MYCPY                                                         *                     5
        * External name  . . . . . . :  MYLIB/QRPGLESRC(MYCPY)                                        *                     5
        * Last change  . . . . . . . :  98/07/24  16:20:04                                            *                     5
        * Text 'description' . . . . :  Text specified on Copy Member                                 *                     5
        *---------------------------------------------------------------------------------------------*
  7
    40+C     Blue(1)       DSPLY                                                                                           5000100
    41+C     Green(4)      DSPLY                                                                                           5000200
    42+C     Red(2)        DSPLY                                                                                           5000300
  8
    43 C     *in20         doweq                  *OFF                                                                      001700
    44 C                   | READ                 InRec                               ----20                               001800
    45 C                   | if                   NOT *in20                                                                 001900
    46 C     FLDA          | | DSPLY                                                                                        002000
    47 C                   | endif                                                                                          002100
    48 C                   enddo                                                                                            002200
    49 C                   write                  outrec                                                                    002300
                                                                                        9
    50 C                   SETON                                                       LR----                               002400
    47 C/DEFINE ABC                                                                                              971104     002500
    51 C/IF DEFINED(ABC)                                                                                         971104     002600
    52 C                   MOVEL                  'x'           Y               10                                          002700
    54 C                   MOVEL                  'x'           Z               10                                          002800
    55 C/ELSE                                                                                                    971104     002900
  10
------ C                   MOVEL     ' '           Y              10                                             971104     003000
------ C                   MOVEL     ' '           Z              10                                             971104     003100
    56 C/ENDIF                                                                                                   971104     003200
```

```
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
    57=OOUTREC                                                                                                             6000001
        *---------------------------------------------------------------------------------------------*                     6
        * RPG record format  . . . . :  OUTREC                                                        *                     6
        * External format  . . . . . :  OUTREC : MYLIB/OUTFILE                                        *                     6
        *---------------------------------------------------------------------------------------------*                     6
    58=O                       FLDY               100A CHAR      100                                                       6000002
    59=O                       FLDZ               132A CHAR       32                                                       6000003
    60=O                       GRAPHFLD           156G GRPH       12 835                                                   6000004
       * * * * *   E N D   O F   S O U R C E   * * * * *
```

1a ***NOSRCSTMT Source Heading**

The source heading shown in the above example was generated with OPTION(*NOSRCSTMT) specified.

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 (on right side of listing)
: Shows the SEU sequence number of the record from a member in a source physical file. Shows an incremental number for records from a /COPY member or records generated from DDS.

1b *SRCSTMT Source Heading

When OPTION(*SRCSTMT) is specified, the source heading changes to:

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

The Ruler Line, Do Number, and Page Line remain unchanged.

Sequence Number (on left side of listing)
: Shows the SEU sequence number of the record from a member in a source physical file. Shows an incremental number for records from a /COPY member or records generated from DDS.

Statement Number
: Shows the statement number generated from the source ID number and the SEU sequence number as follows:

```
stmt_num = source_ID * 1000000 + source_SEU_sequence_number
```

Use this number when debugging using statement numbers.

2 Compiler Options in Effect

Identifies the compiler options in effect. Displayed when compile-option keywords are specified on the control specification.

3 File/Record Information

Identifies the externally described file and the records it contains.

4 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.

5 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. Shows the initial value as specified by the INZ keyword on the definition specification. If INZ(*EXTDFT) is specified for externally described data structure subfields, the DDS default value is displayed. Default values that are too long to fit on one line are truncated and suffixed with '...'.

6 /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.

7 /COPY Member Records

Shows the records from the /COPY member, indicated by a '+' beside the Line Number.

8 Indentation

Shows how structured operations appear when you request that they be marked.

9 Indicator Usage

Shows position of unused indicators, when an indicator is used.

10 **OPTION(*SHOWSKP) Usage**
Shows two statements excluded by an /IF directive, indicated by a '-------' beside the statements. If the OPTION(*NOSHOWSKP) was specified these two statements would be replaced by: LINES EXCLUDED: 2.

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 and sequence number of the source which is in error. Figure 242 on page 474 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 Seq     Message text
*RNF7066 00      8 000800  Record-Format REC1 not used for input or output.
*RNF7066 00      8 000800  Record-Format REC2 not used for input or output.
*RNF7086 00     60 000004  RPG handles blocking for file INFILE. INFDS is updated only
                           when blocks of data are transferred.
*RNF7086 00     60 000004  RPG handles blocking for file OUTFILE. INFDS is updated
                           only when blocks of data are transferred.
 * * * * *   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 242. Sample Additional Diagnostic Messages with OPTION(*NOSRCSTMT)

If OPTION(*SRCSTMT) is specified, the messages will have only the statement number shown. Figure 243 on page 474 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     Statement  Message text
*RNF7066 00        000800  Record-Format REC1 not used for input or output.
*RNF7066 00        000800  Record-Format REC2 not used for input or output.
*RNF7086 00       6000004  RPG handles blocking for file INFILE. INFDS is updated only
                           when blocks of data are transferred.
*RNF7086 00       6000004  RPG handles blocking for file OUTFILE. INFDS is updated
                           only when blocks of data are transferred.
 * * * * *   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 243. Sample Additional Diagnostic Messages with OPTION(*SRCSTMT)

Output Buffer Positions

The Field Positions in Output Buffer Positions 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...'). Figure 244 on page 474 shows an example of an Output Buffer Position table.

```
                O u t p u t   B u f f e r   P o s i t i o n s
Line   Start End   Field or Constant
Number Pos   Pos
    58     1   100 FLDY
    59   101   132 FLDZ
    60   133   156 GRAPHFLD
 * * * * *   E N D   O F   O U T P U T   B U F F E R   P O S I T I O N   *  * * * *
```

Figure 244. Output Buffer Position Table

/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 245 on page 475 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
    39    5 MYCPY      MYLIB      QRPGLESRC  MYCPY          37  98/07/24 16:20:04
         * * * * *   E N D   O F   / C O P Y   M E M B E R S   * * * * *
```

Figure 245. 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 246 on page 475.

```
                       C o m p i l e   T i m e   D a t a
    61 **                                                                                                        971104     003300
        *---------------------------------------------------------------------*
        * Alternate Collating Sequence Table Data:                             *
        *---------------------------------------------------------------------*
    62 ALTSEQ    1122ACAB4B7C36F83A657D73                                                                        971104     003400
Line   <---------------------- Data Records -------------------------------------------------------------->      Change Src Seq
Number ....+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8....+....9....+...10      Date   Id  Number
        *---------------------------------------------------------------------*
        * Alternate Collating Sequence Table:                                 *
        * Number of characters with an altered sequence . . . . . . :  6  1   *
        * 2   0_ 1_ 2_ 3_ 4_ 5_ 6_ 7_ 8_ 9_ A_ B_ C_ D_ E_ F_                 *
        * _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_                 *
        *---------------------------------------------------------------------*
    63 **                                                                                                        971104     003500
        *---------------------------------------------------------------------*
        * Array . . . : BLUE   4    Alternating Array  . . . . : GREEN        *
        *---------------------------------------------------------------------*
    64 1234ZZ                                                                                                    971104     003600
    65 ABCDYY                                                                                                    971104     003700
    66 5432XX                                                                                                    971104     003800
    67 EDCBWW                                                                                                    971104     003900
    68 ABCDEF                                                                                                    0980728    004000
    69 **                                                                                                        971104     00410
        *---------------------------------------------------------------------*
        * Array . . . : RED                                                   *
        *---------------------------------------------------------------------*
    70 3861                                                                                                      971104     00420
    71 TJKL                                                                                                      971104     00430
    * * * * *   E N D   O F   C O M P I L E   T I M E   D A T A   * * * * *
```

Figure 246. 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. For example, the new value for character 3A is 65, found in column 3_ and row _A.

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 247 on page 476 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
                         FLDC                    UCS2        5 13488
                         FLDCV                   VUC2        5 13488
                         FLDG                    GRPH        5 835
            REC2
                         FLD22                   DATE *ISO- 10
                         FLD23                   CHAR        3
  * * * * *   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 247. Sample Key Field Information

Cross-Reference Table

The Cross-Reference table contains at least three lists:

- files and records
- global fields
- indicators

In addition, it contains the local fields that are used by each subprocedure. Use this table to check where files, fields and indicators are used within the module/program.

Note that the informational message RNF7031, which is issued when an identifier is not referenced, will only appear in the cross-reference section of the listing and in the message summary. It does not appear in the source section of the listing.

Names longer than 122 characters, will appear in the cross-reference section of the listing split across multiple lines. The entire name will be printed with the characters '...' at the end of the lines. If the final portion of the name is longer than 17 characters, the attributes and line numbers will be listed starting on the following line. Figure 248 on page 477 shows an example for the module TRANSRPT, which has two subprocedures.

In this example, the Cross-Reference table shows the line numbers for each reference. If you specify OPTION(*SRCSTMT) instead of OPTION(*NOSRCSTMT), the statement numbers will be displayed for each reference and the cross reference listing can extend beyond the first 80 columns of the listing.

```
                          C r o s s   R e f e r e n c e
      File and Record References:
         File             Device            References (D=Defined)
           Record
         CUSTFILE         DISK                   8D
           CUSTREC                               0      44
*RNF7031 CUSTRPT          DISK                   9D
           ARREARS                               0      60      79
      Global Field References:
         Field            Attributes        References (D=Defined M=Modified)
         *INZSR           BEGSR                 63D
         AMOUNT           P(10,2)               56M     83      95
         CITY             A(20)                 53D    132
         CURDATE          D(10*ISO-)            42D     64M     92
         CUSTNAME         A(20)                 50D    122
         CUSTNUM          P(5,0)                49D    124
         DUEDATE          A(10)                 57M     84      91
         EXTREMELY_LONG_PROCEDURE_NAME_THAT_REQUIRES_MORE_THAN_ONE_LINE_IN_THE_CROSS_REFERENCE_EVEN_THOUGH_THE_ENTIRE_LINE_UP_TO_.
            COLUMN_132_IS_USED_TO_PRINT_THE_NAME...
                          I(5,0)                 9D
                          PROTOTYPE
         FMTCUST          PROTOTYPE             35D     59     113     114
                                               134
         INARREARS        A(1)                  30D     58      85      86
                          PROTOTYPE            101
         LONG_FLOAT       F(8)                  7D     11M     12M
         NUMTOCHAR        A(31)                 22D    124     130
                          PROTOTYPE
         RPTADDR          A(100)                59      82
         RPTNAME          C(100)                59      81
                          CCSID(13488)
         RPTNUM           P(5,0)                80
         SHORT_FLOAT      F(4)                  8D     10M
*RNF7031 STATE            A(2)                  54D
         STREETNAME       A(20)                 52D    131
         STREETNUM        P(5,0)                51D    130
         THIS_NAME_IS_NOT_QUITE_SO_LONG...
                          A(5)                   7D
         UDATE            S(6,0)                64
*RNF7031 ZIP              P(5,0)                55D
      INARREARS Field References:
         Field            Attributes        References (D=Defined M=Modified)
         DAYSLATE         I(10,0)               88D     92M     94
         DATEDUE          D(10*ISO-)            89D     91M     92
      FMTCUST Field References:
         Field            Attributes        References (D=Defined M=Modified)
         NAME             A(100)               115D    122M
                          BASED(_QRNL_PST+)
         ADDRESS          A(100)               116D    130M
                          BASED(_QRNL_PST+)
      Indicator References:
         Indicator                          References (D=Defined M=Modified)
*RNF7031 01                                     44D
       * * * * *   E N D   O F   C R O S S   R E F E R E N C E   * * * * *
```

*Figure 248. Sample Cross-Reference Table with OPTION(*NOSRCSTMT)*

EVAL-CORR Summary

When OPTION(*XREF) is specified, the EVAL-CORR summary lists every subfield in either the source or the target data structure indicating

- whether the subfield is assigned
- the reason the source and target subfields are not considered to correspond, if the subfield is not assigned
- for subfields that are assigned, additional information that may affect the assignment such as a difference in the number of array elements or the null-capability of the subfields

When OPTION(*NOXREF) is specified, the EVAL-CORR summary does not list any information about corresponding subfields. It only lists the subfields that do not correspond, with the reason that the subfields are not considered to correspond.

```
        EVAL-CORR summary 1   1                     13      14      19      24   2
                                                    28
            FLD1              Assigned; exact match
            FLD2              Assigned; target and source are compatible
                              Target subfield has fewer elements than source subfield 3
            FLD3              Assigned; exact match
                              Target subfield is null-capable; source subfield is
  4 *RNF7349 FLD5             Not same data type in source and target
        EVAL-CORR summary 2                         22
            FLD1              Assigned; exact match
            SUBDS  5
             SUBF1            Assigned; exact match
                              Target subfield is defined using OVERLAY
            FLD2              Assigned; exact match
     *RNF7341 FLD3            In target only.
```

Figure 249. EVAL-CORR summary

1 EVAL-CORR Summary Number
Messages in the Additional Diagnostics section refer to the relevant EVAL-CORR summary by number.

2 EVAL-CORR Statement Numbers
EVAL-CORR operations with the same (either identical or related through LIKEDS or LIKEREC) source and target data structures share the same EVAL-CORR summary. In this example, there are five EVAL-CORR operations with one pair of data structure definitions, and one EVAL-CORR operation with the other pair.

3 Additional Information for a Subfield
The subfield is assigned. Additional information is listed on separate lines.

4 Message Indicating that the Subfield is not Assigned
The subfield is not assigned. The error message and text indicate the reason the subfields are not considered to correspond is given.

5 Data Structure Subfields
If the subfield is a data structure, its subfields are listed with indentation.

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, or the name of a prototyped bound procedure called by CALLP or within an expression.

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. Figure 250 on page 479 shows an example.

```
                       E x t e r n a l   R e f e r e n c e s
      Statically bound procedures:
         Procedure                           References
         PROTOTYPED                               2       2
         PADDR_PROC                               4
         CALLB_PROC                               6
      Imported fields:
         Field            Attributes         Defined
         IMPORT_FLD       P(5,0)                 3
      Exported fields:
         Field            Attributes         Defined
         EXPORT_ARR(2)    A(5)                   2
   * * * * *   E N D   O F   E X T E R N A L   R E F E R E N C E S   * * * * *
```

Figure 250. Sample External References

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 251 on page 479 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     16 The name or indicator is not referenced.
                     Cause . . . . . :   The field, subfield, TAG, data
                       structure, PLIST, KLIST, subroutine, indicator, or
                       prototype is defined in the program, but not referenced.
                     Recovery  . . . :   Reference the item, or remove it from
                       the program.  Compile again.
*RNF7066 00      2 Record-Format name of Externally-Described file is not used.
                     Cause . . . . . :   There is a Record-Format name for an
                       Externally-Described File that is not used on a valid
                       input or output operation.
                     Recovery  . . . :   Use the Record-Format name of the
                       Externally-Described File for input or output, or specify
                       the name as a parameter for keyword IGNORE. Compile
                       again.
*RNF7086 00      2 RPG handles blocking for the file. INFDS is updated only when
                   blocks of data are transferred.
                     Cause . . . . . :   RPG specifies MLTRCD(*YES) in the UFCB
                       (User-File-Control Block). Records are passed between RPG
                       and data management in blocks. Positions 241 through the
                       end of the INFDS (File-Information-Data Structure) are
                       updated only when a block of records is read or written.
                     Recovery  . . . :   If this information is needed after
                       each read or write of a record, specify the OVRDBF
                       command for the file with SEQONLY(*NO).
       * * * * *   E N D   O F   M E S S A G E   S U M M A R Y   * * * * *
```

Figure 251. 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 252 on page 480 shows an example.

```
                          F i n a l   S u m m a r y
  Message Totals:
    Information  (00) . . . . . . . :       20
    Warning      (10) . . . . . . . :        0
    Error        (20) . . . . . . . :        0
    Severe Error (30+)  . . . . . . :        0
    ---------------------------------  -------
    Total . . . . . . . . . . . . . :       20
  Source Totals:
    Records . . . . . . . . . . . . :       71
    Specifications  . . . . . . . . :       55
    Data records  . . . . . . . . . :        8
    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 98/07/28 at 14:21:03.
          * * * * *   E N D   O F   C O M P I L A T I O N * * * * *
```

Figure 252. 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.

Appendix E. Information for Preprocessor Providers

An RPG preprocessor which merges the main source and the copy files into a new source member must observe the following rules related to code containing the special directive **FREE.

Special directive **END-FREE is available for use by preprocessors to indicate the end of fully-free code in a preprocessed source member which merges the main source file and the copy files into a single source file.

- If the preprocessor is merging copy files into the preprocessed output:
  - The preprocessor must insert **END-FREE into the preprocessor output at the end of the copy file, if the copy file was in fully free-form mode and the file containing the /COPY or /INCLUDE directive was not in fully free-form mode.
  - The preprocessor may insert additional **FREE and **END-FREE lines if it is convenient, provided that **FREE is only inserted when the preprocessed code is in column-limited source mode, and that **END-FREE is only inserted when the preprocessed code is in fully free-form source mode.
  - The preprocessor must insert **FREE into the preprocessor output at the end of the copy file, if the copy file was not in fully free-form mode and the file containing the /COPY or /INCLUDE directive was in fully free-form mode.
  - The remainder of inserted lines containing **FREE or **END-FREE must be blank.
- The **FREE and **END-FREE special directives must be balanced in the output source, except that it is not necessary for the source to end with **END-FREE if the source ends in full-free mode.
- When to accept a special directive:
  - **FREE and **END-FREE may only be coded between statements.
  - **FREE cannot be specified when the source is in full free-form mode.
  - **END-FREE cannot be specified when the source is in column-limited mode.
  - **FREE can only be specified in the first line of a copy file.

  - **END-FREE cannot be specified in a copy file.
- If the special directive is not accepted, and the preprocessor chooses to continue preprocessing, the preprocessor must not copy the invalid special directive to the output of the preprocessor.

Chapter 8. Bibliography

For additional information about topics related to ILE RPG programming on the IBM i, refer to the following IBM publications:

- *ADTS/400: Programming Development Manager, SC09-1771-00*, provides information about using the 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 for AS/400: Source Entry Utility, SC09-2605-00*, provides information about using the Application Development ToolSet 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, SC41-5715-02*, 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
- *Recovering your system, SC41-5304-10*, 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, SC41-5721-06*, provides a wide-ranging discussion of IBM i 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.
- *Communications Management, SC41-5406-02*, 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.
- *GDDM Programming Guide, SC41-0536-00*, provides information about using IBM i 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, SC41-3718-00*, provides information about using IBM i 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, SC41-5442-00*, provides information needed to write application programs that use IBM i communications and the IBM i intersystem communications function (IBM i-ICF). Also contains

information on data description specifications (DDS) keywords, system-supplied formats, return codes, file transfer support, and program examples.

- *IDDU Use, SC41-5704-00*, describes how to use the IBM i 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.
- *IBM Rational Development Studio for i: ILE C/C++ Programmer's Guide, SC09-2712-07*, provides information on how to develop applications using the ILE C 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.
- *IBM Rational Development Studio for i: ILE COBOL Programmer's Guide, SC09-2540-07*, provides information about how to write, compile, bind, run, debug, and maintain ILE COBOL programs on the IBM i. It provides programming information on how to call other ILE COBOL and non-ILE COBOL 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, SC41-5606-09*, explains concepts and terminology pertaining to the Integrated Language Environment (ILE) architecture of the IBM i licensed program. Topics covered include creating modules, binding, running programs, debugging programs, and handling exceptions.
- *IBM Rational Development Studio for i: ILE RPG Reference, SC09-2508-08*, provides information about the ILE RPG 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.
- *Printer Device Programming, SC41-5713-06*, provides information to help you understand and control printing. Provides specific information on printing elements and concepts of the IBM i, 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 IBM i 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 IBM i are also provided. Font substitution tables provide a cross-reference of substituted fonts if attached printers do not support application-specified fonts.
- *Security reference, SC41-5302-11*, 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.
- *Installing, upgrading, or deleting IBM i and related software, SC41-5120-11*, 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 want to install a new release.
- *Who Knew You Could Do That with RPG IV? A Sorcerer's Guide to System Access and More* provides hints and tips for IBM i programmers who want to take full advantage of RPG IV and the Integrated Language Environment (ILE). It is available from the IBM Redbooks® Web Site:

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

Notices

Programming interface information

This ILE RPG Programmer's Guide publication documents intended Programming Interfaces that allow the customer to write programs to obtain the services of IBM i.

Trademarks

Terms and conditions

Index

Special Characters

*CALLER 145
*CANCL 307
*DETC 307
*DETL 307
*ENTRY PLIST 182
*EXTDFT
  example 471
  in compiler listing 473
*GETIN 307
*JOB
  sort sequence, SRTSEQ 456
*JOBRUN
  language identifier, LANGID 457
  sort sequence, SRTSEQ 456
*NEW 144
*OFL 307
*OMIT 171
*TOTC 307
*TOTL 307
*USER
  user profile, USRPRF 457
%ADDR (Get Address of Variable)
  omitted parameters 171
%ADDR debug built-in 280
%ALLOC built-in function 148
%INDEX debug built-in 280
%PARMS (Return Number of Parameters)
  checking for number of parameters 172
%REALLOC built-in function 148
%SUBSTR debug built-in
  changing values 282
  examples 281
%VARS debug built-in 280
/COPY statement
  conversion problems 435, 443
  COPY debug view 245
  in a conversion report 439
  table in compiler listing 475
  using source files 98

Numerics

01-99 indicators
  displaying while debugging 276
  in a sample formatted dump 324

A

abnormal program/procedure end 184
access path
  example of 348
  for externally described DISK file 343
  for indexed file 348
ACTGRP parameter
  CRTBNDRPG command 102, 460
ACTGRP parameter *(continued)*
  CRTPGM command 124
  specifying 144
  using 105
activation group
  *CALLER
    running in OPM default 145
    specifying 145
  *NEW
    ending 144
    specifying 144
  definition 144
  deleting 145
  identifying 124, 144
  managing 144
  named
    deleting 144
    specifying 144
  OPM default 145
  QILE 124, 144
  role in exception handling 291
activation, program 144
Add Reply List Entry (ADDRPLYE) command
  adding to system reply list 143
adding objects to a debug session 249
additional diagnostic messages section of compiler listing 474
all- thread static variables 191
ALLOC (allocate storage) operation code 148
allocating storage for a run-time array 153
alternate collating sequence
  debug considerations 258
ALWNULL parameter
  CRTBNDRPG command 102, 459
  CRTRPGMOD command 116
analyzing your conversion 438
application programming interface (API)
  calling non-bindable 160
  QMHSNDPM 421
  Retrieve Message (QMHRTVM) API 187
area parameter for SPECIAL PLIST 380
array
  conversion problems 446
  displaying while debugging 273
  loading 447
  prerun-time arrays 447
arrival sequence access path 343
ATTR debug command
  definition 242
  example 284
  using 284
audit file 427
AUT parameter
  CRTBNDRPG command 102, 458
  CRTRPGMOD command 116
authority to commands 3
auto report program

auto report program *(continued)*
  converting to ILE RPG 435
avoiding a loop in an error subroutine 306

B

behavior of bound ILE RPG modules 122
behavioral differences between OPM RPG/400 and ILE RPG 421
bibliography 483
bindable APIs
  calling conventions 186
  CEE4ABN 185
  CEECRHP (Create Heap) 69, 153
  CEECZST (Reallocate Storage) 69
  CEEDSHP (Discard Heap) 69, 153
  CEEFRST (Free Storage) 69
  CEEGTST (Get Heap Storage) 69, 153
  CEEHDLR (Register ILE Condition Handler) 308
  CEEHDLU (Unregister ILE Condition Handler) 308
  CEERTX (Register Call Stack Entry Termination User Exit Procedure) 313
  CEETREC 185
  CEETSTA (Check for Omitted Argument) 171
  CEEUTX (Call Stack Entry Termination User Exit Procedure) 313
  Create Heap (CEECRHP) 69, 153
  description 186
  Discard Heap (CEEDSHP) 69, 153
  Free Storage (CEEFRST) 69
  Get Descriptive Information About a String Argument (CEESGI) 170
  Get Heap Storage (CEEGTST) 69, 153
  overview 69
  passing operational descriptors to 170
  Reallocate Storage (CEECZST) 69
  Retrieve Operational Descriptor Information (CEEDOD) 170
  returning from a procedure 185
  sample coding 186
binder language
  example 134
  reasons for using 130
binder listing
  as maintenance resource 127
  basic 138
  creating 127
  determining exports in service program 130
  sections of 127
binding
  after modifying a module 127
  definition 122
  modules into a program 122
  service program to a program 135
binding errors in compiler listing 480
binding multiple modules 125
blocking/unblocking records 347
BNDDIR parameter on CRTBNDRPG
  CRTBNDRPG command 102, 460
  CRTRPGMOD command 116
  static binding 105
BREAK debug command
  definition 242
  example 257
BREAK debug command *(continued)*
  using 254, 256, 259
breakpoints
  conditional
    setting and removing for job 255
    setting and removing for thread 261
  removing all 262
  setting and removing 253
  setting using statement numbers 259
  testing 253
  unconditional
    setting and removing for job 253
    setting and removing for thread 255
browsing a compiler listing using SEU 112
built-in functions
  %ADDR 171

C

calculation specifications
  general description 57
  program-described WORKSTN file 389
CALL (call a program) operation code
  in a conversion report 439
  using 180
CALL CL command
  example passing parameters 139
  passing parameters 139
  running a program 139
call operations
  calling programs 180
  DSPPGMREF 181
  fixed-form call 180
  free-form call 166
  query names of called procedures 181
  special routines 187
  using 166
call stack 161, 291
Call Stack Entry Termination User Exit Procedure (CEEUTX) 313
CALLB (call a bound procedure) operation code
  calling programs 180
  using 180
calling
  Java from RPG 214
  RPG from Java 220
calling a graphics routine 187
calling programs/procedures
  abnormal program/procedure end 184
  call stack 161
  calling bindable APIs 186
  calling graphics 187
  calling procedures 160
  calling programs 160
  calling special routines 187
  fixed-form call 180
  free-form call 166
  interlanguage calls 178
  normal program/procedure end 183
  overview 160
  parameter passing methods 167
  recursive calls 161
  returning from a called program or procedure 183
  returning values 166

calling programs/procedures *(continued)*
    returning without ending 184
    static calls 160
    using the CALL operation 180
    using the CALLB operation 180
    within ILE 69
CALLP (call a prototyped program or procedure) operation code
    using 166
cancel handler
    CEERTX (Register Call Stack Entry Termination User Exit Procedure) 313
    CEEUTX (Call Stack Entry Termination User Exit Procedure) 313
    example 313
    using 313
CCSIDs
    indicated in compiler listing 471
CEE4ABN 185
CEECRHP (Create Heap) bindable API 69, 153
CEECZST (Reallocate Storage) bindable API 69
CEEDOD (Retrieve Operational Descriptor Information)
    example 171
    operational descriptors 170
CEEDSHP (Discard Heap) bindable API 69, 153
CEEFRST (Free Storage) bindable API 69
CEEGTST (Get Heap Storage) bindable API 69, 153
CEEHDLR (Register ILE Condition Handler) 308
CEEHDLU (Unregister ILE Condition Handler) 308
CEERTX (Register Call Stack Entry Termination User Exit Procedure) 313
CEESGI (Get Descriptive Information About a String Argument) 170
CEETREC 185
CEETSTA (Check for Omitted Argument) 171
CEEUTX (Call Stack Entry Termination User Exit Procedure) 313
Change Module (CHGMOD) command
    removing observability 129
Change Program (CHGPGM) command
    optimization parameters 128
    removing observability 129
Change Service Program (CHGSRVPGM) command 137
changing a module 127
changing a program 127
changing a service program 131
changing field values while debugging 282
changing optimization level
    of a program or module 128
changing the debug view of a module 252
character format
    character CCSID
        indicated in compiler listing 471
    displaying while debugging 277
Check for Omitted Argument (CEETSTA) 171
checking for the number of passed parameters 172
checking, level 333
CL commands
    Add Program (ADDPGM) 249
    additional service program commands 131
    ADDRPLYE 143
    authority 3
    CALL 139
    Change Module (CHGMOD) 128
CL commands *(continued)*
    CHGPGM 129
    commonly used commands 65
    CRTPGM command 124
    CRTRPGMOD 116
    CVTRPGSRC 429
    Display Module Source (DSPMODSRC) 248–251
    DSPMOD 181
    DSPPGMREF 181
    End Debug (ENDDBG) 247
    module-related 122
    MONMSG 316
    program-related 126
    RCLACTGR 144
    RCLRSC 146
    reading syntax diagrams 448
    Remove Program (RMVPGM) 249
    Start Debug (STRDBG) 247, 248
    UPDPGM 128
    using 448
    WRKRPLYE 143
clear command 384
CLEAR debug command
    definition 243
    removing all 262
    using 254, 256, 261
code conversion constraints 442
code generation errors in compiler listing 480
combined file 390
command attention (CA) keys 382
command definition 142
command function (CF) keys 382
COMMIT (commit) operation code
    commitment control 365
    system considerations 365
    with multiple devices 365
commitment control
    COMMIT operation 365
    conditional 367
    example 366
    in program cycle 367
    locks 364
    scoping 365
    specifying files 365
    starting and ending 364
communication
    accessing other programs and systems 382
compatibility differences between OPM RPG/400 and ILE RPG 421
compilation errors, correcting 110
compile time array or table
    section in compiler listing 475
compiler directives
    changing a listing heading 109
compiler listing
    additional diagnostic messages 111
    browsing using SEU 112
    coordinating listing options with debug view options 112
    correcting compilation errors 110
    correcting run-time errors 112
    default information 108
    in-line diagnostic messages 111
    indenting structured operations 109
    obtaining 108

compiler listing *(continued)*
  reading 466
  sample listing 468
  sections of 108, 468
  specifying the format of 109
  using 107
  using as documentation 112
compiling
  creating modules 115
  differences between ILE RPG and OPM RPG/400 421
  in ILE 67
  using CRTBNDRPG command 102
compressing an object 129
condition handler
  example 308
  overview 307
  percolate an exception 308
  recursive calls 308
  registering 308
conditional breakpoint
  definition 253
  setting 256
  setting and removing for job 255
  setting and removing for thread 261
  using statement numbers 259
conditional commitment control, specifying 367
conditioning output
  overflow indicators 371
consecutive processing 351
control boundary 291
control break
  example 372
control language (CL) program
  as module in ILE program 75
  commands used with ILE RPG 65
  commonly used commands 65
  in OPM-compatible application 71
control specification keywords
  compile-option keywords
    compiler listing example 468
control specifications
  conversion considerations 428
  example 61
  general description 57
control-record format, subfile 385
conversion reports
  obtaining 434
  sections of 438
  using 438
conversion, analyzing 438
converting to RPG IV
  analyzing your conversion 438
  constraints 428
  conversion problems 442
  converting 428
  converting all file members 433
  converting auto report source members 435
  converting some file members 434
  converting source from a data file 435
  converting source members with embedded SQL 435
  CVTRPGSRC command 429
  example 435
  file and member names 427
  file considerations 426
converting to RPG IV *(continued)*
  file record length 427
  log file 427
  obtaining conversion reports 434
  overview 425
  performing a trial conversion 434
  requirements 427
  using a conversion error report 438
  using the log file 440
  valid source member types 426
coordinating listing options with debug view options 112
correcting compilation errors 110
correcting run-time errors 112
CPF417C
  due to override 336
Create Bound RPG Program (CRTBNDRPG) command
  and ILE 68
  coordinating listing options with debug view 112
  creating programs 101
  default parameter values 102
  examples
    OPM-compatible program 105
    program for source debugging 104
    program with static binding 105
  parameter description 451
  parameters grouped by function 102
  program creation strategy 71, 73
  RETURNCODE data area 113
  syntax diagram 449
  using 102
Create Heap (CEECRHP) bindable API 69, 153
Create Program (CRTPGM) command
  and ILE 68
  creating a program 115
  examples
    binding multiple modules 125
  parameters 125
  system actions 125
  using 124
Create RPG Module (CRTRPGMOD) command
  and ILE 68
  default values of parameters 116
  defaults 118
  examples 135
  parameter description 466
  parameter grouping by function 116
  program creation strategy 75
  syntax diagram 464
  using 116
Create Service Program (CRTSRVPGM) command
  and ILE 67
  example 135
  parameters 131
creating a binder listing 127
creating a debug view
  COPY 245
  listing 246
  root source 245
  statement 246
creating a library 93
creating a module
  general discussion 115
  using CRTRPGMOD 116
  using CRTRPGMOD defaults 118

creating a program with the CRTBNDRPG command 101
creating a source physical file 93
creating programs
    coding considerations 89
    examples of 104, 105, 125
    OPM-compatible
        creating 71
        strategy to avoid 78
    strategies for
        CRTPGM command 124
        ILE application using CRTRPGMOD 75
        OPM-compatible 71
        strategy to avoid 78
        using CRTBNDRPG 73
        using CRTRPGMOD and CRTPGM 115
        using the one-step process 101
creating service programs
    about 129
    strategies 130
cross-reference listing 476
CRTBNDRPG command 452
CRTRPTPGM (create auto report program) command
    converting auto report members 435
CVTOPT parameter
    CRTBNDRPG command 102, 456
    CRTRPGMOD command 116
CVTRPGSRC (Convert RPG Source) command
    default parameter values 429
    example 433
    parameter description 430
    syntax diagram 429
    using the command defaults 433
CVTRPT parameter 432, 434, 438
cycle-free module 118
cycle, program
    commitment control 367
    fetch overflow logic 373
    general description 58
    last cycle 59

D

data areas
    RETURNCODE 113
data file, converting source from 435
data management operations 334
data structures
    multiple-occurrence
        displaying while debugging 274
    subfields
        conversion problems 446
        displaying while debugging 274
    using EVAL debug command 274
database data
    updating while debugging 248
database file
    data file 342
    field level description 342
    general discussion 342
    physical and logical files 342
    record level description 342
    source member 342
DB2 for AS/400 SQL
    entering SQL statements 97
DBCS
    in RPG IV character fields 425
    NLSS debug considerations 258
DBGVIEW parameter
    coordinating with listing options 112
    CRTBNDRPG command 102, 454
    CRTRPGMOD command 116
    preparing a program for debugging 244
    using 104
    values for viewing source 250
DEALLOC (free storage) operation code 148
debug commands
    ATTR 284
    CLEAR 254
    DISPLAY 251
    EQUATE 285
    equating with a name while debugging 285
    EVAL 271, 282
    general discussion 242
    STEP 266, 267
    STEP INTO 268
    STEP OVER 267
    WATCH 262
debug data
    creating 244
    effect on object size 244
    none 244
    removing from a module 129
debug view
    changing while debugging 252
    COPY source 245
    default 246
    definition 244
    listing 246
    root source 245
    statement 246
debugging
    adding an object to a session 249
    built-in functions
        %ADDR 280
        %INDEX 280
        %SUBSTR 280
        %VARS 280
        changing values using %SUBSTR 282
        examples 281
        general discussion 280
    changing field values 282
    changing modules while debugging 251
    coordinating with listing options 112
    creating a program for debugging 104
    differences between ILE RPG and OPM RPG/400 422
    displaying attributes of 284
    displaying data addressed by pointers 278
    displaying data and expressions 271
    displaying fields as hexadecimal values 277
    displaying fields in character format 277
    displaying fields in UCS-2 format 277
    displaying fields in variable-length format 278
    displaying indicators 276
    displaying multiple-occurrence data structures 274
    displaying the contents of a table 274
    displaying the contents of an array 273
    general discussion 241
    National Language Support 286

debugging *(continued)*
    NLSS considerations 258
    obtaining a formatted dump 317
    OPM program limit in debug session 249
    optimization effects 128, 242
    overview 69
    preparing a program 244
    removing an object from a session 249, 250
    rules for assigning values using EVAL 282
    setting and removing breakpoints 253
    setting debug options 248
    setting watch conditions 262
    starting the source 247
    stepping through 266
    unexpected results 273
    updating production files 248
    viewing shorthand names 286
    viewing source 250
decimal positions
    input specifications
        program-described WORKSTN file 389
        with external descriptions 331
decompressing an object 129
default activation group
    running in 145
default exception handler, RPG 293
default heap 148
DEFINE parameter
    CRTBNDRPG command 102, 461
    CRTRPGMOD command 116
definition specifications
    general description 57
deleting an activation group 145
description of parameters
    CRTBNDRPG command 451
    CRTRPGMOD command 466
    CVTRPGSRC command 430
descriptors, operational
    definition 170
    example 132
DETAIL parameter
    creating a binder listing 127
DETC 307
detecting errors in a program 241
DETL 307
device files
    device dependence 325
    device independence 325
    DISK files 341
    general discussion 369
    multiple-device 391
    PRINTER files 370
    SEQ files 378
    workstation files 382
device name, function of 326
devices
    WORKSTN 382
DFTACTGRP parameter on CRTBNDRPG
    CRTBNDRPG command 102
    description 452
    running in OPM default 145
    using 101, 105
diagnosing errors in a program 241
differences between OPM and ILE RPG
differences between OPM and ILE RPG *(continued)*
    behavioral differences 421
    exception handling 294
different views of a module 252
Discard Heap (CEEDSHP) bindable API 69, 153
DISK file
    externally described
        access path 343
        as program-described 328
        examples 343
        general description 342
        record format specifications 342
    file operation codes allowed
        for keyed processing methods 362, 363
        for non-keyed processing methods 362, 363
    general description 341
    processing methods
        consecutive processing 351
        overview 350
        random-by-key processing 357
        relative-record-number processing 360
        sequential-by-key processing 352
        sequential-within-limits processing 358
    program-described
        indexed file 348
        processing 350
        record-address file 350
        sequential file 349
    record-format specifications 342
DISPLAY debug command
    definition 243
    using 251
    viewing shorthand names 286
Display Module (DSPMOD) command 181
Display Module Source (DSPMODSRC) command 248–251
Display Program (DSPPGM) command
    determining optimization level 128
Display Program References (DSPPGMREF) command 181
Display Service Program (DSPSRVPGM) command 130
displaying attributes of a field 284
displaying data and expressions while debugging 271
distributed data management (DDM)
    files 368
documentation of programs 112
double byte character set
    in RPG IV character fields 425
    NLSS debug considerations 258
DUMP (program dump) operation code
    obtaining a formatted dump 317
    using 318
dump, formatted 317
dynamic array
    allocating storage during run-time 153
dynamic calls 69, 160
dynamic storage 146

E

edit source (STRSEU) command 94
eliminating errors in a program 241
ENBPFRCOL parameter
    CRTBNDRPG command 102, 461
    CRTRPGMOD command 116
End Debug (ENDDBG) command 247

ending a program or procedure
- abnormal end 184
- after system call 143
- normal end 183
- return overview 183
- returning without ending 184
- using bindable APIs 185

ending commitment control 364
ENDSR (end of subroutine) operation code
- specifying a return point 307

ENTMOD parameter 124
entry module 75, 124
EQUATE debug command
- definition 243
- example 285
- using 285

equating a name with a field, expression, or command 285
error indicators
- specifying 298

error subroutines
- avoiding a loop 306
- for file errors 301
- program 303
- using 300

errors
- correcting compilation 110
- correcting run-time 112
- file 293
- program 293
- runtime, Java 233
- when calling Java from RPG 223

escape messages
- definition 291
- unhandled 296

EVAL debug command
- changing values 282
- contents of a table 274
- contents of an array 273
- definition 243
- displaying data structures 274
- example 271, 272, 283
- in character format 277
- in UCS-2 format 277
- in variable-length format 278
- indicators 276
- rules for assigning values 282
- using 271

event file 453
event file for Rational Developer for i 453
examples
- compiling
  - binding multiple modules 125
  - OPM-compatible program 105
  - program for source debugging 104
  - program with static binding 105
  - sample binder listing 138
  - service program 132
- converting to RPG IV
  - all members in a file 433
  - performing a trial conversion 434
  - sample conversion 435
  - some members in a file 434
- debugging
  - adding a service program to a session 249

examples *(continued)*
- debugging *(continued)*
  - changing field values 283
  - changing the debug view of a module 252
  - displaying attributes of a field 284
  - displaying data addressed by pointers 278
  - displaying fields as hexadecimal values 277
  - displaying fields in character format 277
  - displaying fields in UCS-2 format 277
  - displaying fields in variable-length format 278
  - displaying indicators 276
  - displaying multiple-occurrence data structures 274
  - displaying the contents of a table 274
  - displaying the contents of an array 273
  - removing programs from a session 250
  - setting a conditional breakpoint 256
  - setting an unconditional breakpoint 254
  - setting debug options 248
  - source for debug examples 286
  - using %SUBSTR to display field values 281
  - viewing a different module in a debug session 251
- handling exceptions
  - *PSSR error subroutine 304
  - avoiding a loop in an error subroutine 306
  - cancel handler 313
  - file error subroutine 301
  - unhandled escape message 296
  - unhandled function check 297
  - using a cancel handler 313
  - using a condition handler 308
- I/O
  - data maintenance 397
  - inquiry by zip code and search on name 412
  - inquiry program 394
  - subfile processing 406
- interactive application 393
- managing your own heap 153
- module with multiple procedures 85
- passing parameters using the CL CALL command 139
- program/procedure call
  - checking number of passed parameters 172
  - using omitted parameters 132
- sample ILE RPG program 60
- subprocedures
  - creating a NOMAIN module 118

exception
- monitoring during run time 144
- nested, 295

exception handler
- priority of 295
- RPG-specific 293, 298

exception messages
- percolation 291
- types of 291
- unexpectedly handled by CL MONMSG 316
- unhandled 295

exception/error handling
- *PSSR error subroutine 303
- avoiding a loop 306
- cancel handler 313
- condition handler 307
- differences between ILE RPG and OPM RPG/400 294, 422
- error indicators 298

exception/error handling *(continued)*
  error/exception subroutine overview 300
  exceptions 232
  file error/exception (INFSR) subroutine 301
  general considerations 295
  MONITOR group 299
  NOOPT keyword 298
  optimization considerations 298
  overview 291
  percolation 291
  RPG-specific 293
  specifying a return point 307
  types of 290
  unhandled 295
  using 'E' extender 298
EXFMT (write/then read format) operation code 391
EXPCPY parameter 432
EXPORT keyword
  duplicate names 125
expressions
  returning values 166
extension specifications
  conversion problems 437, 444
external-references list in compiler listing 478
externally described file
  access path 343
  adding to external description 329
  advantages 325
  as program-described 328
  as WORKSTN file 382, 385
  definition 327
  file description specifications for 329
  output specifications for 332
  overriding 331
  physical and logical files 342
  record format specifications 342
  renaming field names 330
  renaming record format 329
  specifications 329

F

fetch overflow
  general description 373
  logic 373
field
  changing the value while debugging 282
  displaying attributes of while debugging 284
  displaying while debugging
    as hexadecimal values 277
    in character format 277
    in UCS-2 format 277
    in variable-length format 278
    using EVAL 271
  equating with a name while debugging 285
  maintaining current values while debugging 242
field-reference file, example of 343
file
  device dependence 325
  device independence 325
  differences between ILE RPG and OPM RPG/400 423
  DISK 341
  externally described 325
  externally described disk 342
file *(continued)*
  general considerations 325
  indexed 348
  locking 337
  name
    externally described 325
    override 331
    program-described 334
  naming conventions 327
  open options 339
  override 331
  PRINTER 370
  processing charts
    sequential file 378
    SPECIAL file 380
    WORKSTN file 391
  program described 325, 334
  redirection 326
  SEQ 349, 378
  sharing 339
  valid keys 345
  WORKSTN 382
file description specifications
  commitment control 365
  for externally described files 329
  general description 57
file exception/error subroutine (INFSR)
  description 301
  example 301
  specifications for 301
file exception/errors
  definition 293
  example 301
  using an INFSR subroutine 301
file information data structure
  example 301
  using in an error subroutine 301
file locking 337
file operations
  allowed with DISK file 362, 363
  allowed with PRINTER file 370
  allowed with sequential file 378
  allowed with SPECIAL file 380
  allowed with WORKSTN file 390
file overrides
  example 337
  general discussion 336, 361
  indicated in compiler listing 469
file record length, conversion considerations 427
file sharing 339
final summary in compiler listing 479
FIND debug command 243
FIXNBR parameter
  CRTBNDRPG command 102, 458
  CRTRPGMOD command 116
flowchart
  fetch-overflow logic 373
format name 389
format of compiler listing, specifying 109
formatted dump 317
FREE (deactivate a program) operation code 440
Free Storage (CEEFRST) bindable API 69
freeing resources of ILE programs 146
FROMFILE parameter 431

FROMMBR parameter 431, 433
function check
    definition 291
    unhandled 297
function keys
    indicators 384
    with WORKSTN file 384

G

GDDM 187
GENLVL parameter
    CRTBNDRPG command 102, 452
    CRTRPGMOD command 116
Get Descriptive Information About a String Argument (CEESGI) 170
Get Heap Storage (CEEGTST) bindable API 69, 153
graphic format
    graphic CCSID
        indicated in compiler listing 471
    NLSS debug considerations 258
    rules for assigning values using EVAL 282
graphic support 187
Graphical Data Display Manager(GDDM) 187

H

halt (H1-H9) indicators
    used to end a program/procedure 183, 184
handling exceptions/errors
    *PSSR error subroutine 303
    avoiding a loop 306
    cancel handler 313
    condition handler 307
    differences between ILE RPG and OPM RPG/400 294, 422
    error indicators 298
    error/exception subroutine overview 300
    file error/exception (INFSR) subroutine 301
    general considerations 295
    NOOPT keyword 298
    optimization considerations 298
    overview 291
    percolation 291
    RPG-specific 293
    specifying a return point 307
    types of 290
    unhandled 295
    using 'E' extender 298
heap
    default heap 148
    definition 146
    example 153
help command key 384
hexadecimal values, displaying while debugging 277
home command key 384

I

I/O differences between ILE RPG and OPM RPG/400 423
ICF communications file 382
identifying an activation group 144
IGNORE keyword 330
ignoring record format 330
ILE C
    as ILE language 66
    in advanced application 77
    in mixed-language application 76
    parameter passing method 178
    source for module in debug example 290
ILE CL
    as ILE language 66
    as module in ILE program 75
    calling ILE RPG program 76
    calling RPG program 73
    in advanced application 77
    in mixed-language application 76
    parameter passing method 178
    unexpectedly handling status and notify exceptions 316
ILE COBOL
    as ILE language 66
    parameter passing method 178
ILE RPG
    behavior of bound modules 122
    behavioral differences between OPM RPG/400 421
    converting to 425
    data management operations 334
    device types supported 369
    exception handling overview 293
    logic chart 58
    overview of RPG IV language 57
    sample program 60
ILE source debugger
    debug commands 242
    description 242
    starting 247
include source view, creating 245
INDENT parameter
    CRTBNDRPG command 102, 455, 456
    CRTRPGMOD command 116
indenting structured operations in the compiler listing 109
indexed file
    access path 348
    general description 348
    valid search arguments 348
indicators
    as error indicators 298
    displaying while debugging 276
    error 298
    function key (KA-KN, KP-KY)
        with WORKSTN file 384
    halt (H1-H9)
        used to end a program/procedure 183, 184
    last record (LR)
        general description 59
        used to end a program/procedure 183, 184
    overflow
        examples 373
        fetch overflow logic 373
        general description 370
        presence or absence of 371
        relation to program cycle 373
        setting of 373
        with PRINTER file 370
    return (RT)
        used to end a program/procedure 183, 184
    using 59

input
  file 390
input record
  unblocking 347
input specifications
  general description 57
inquiry messages
  list of 143
  replying to 143
inserting specification templates 435
INSRTPL parameter 432, 435
integer format
  TRUNCNBR parameter 458
Integrated Language Environment (ILE)
  effect on
    OPM-compatible program 72
    program using CRTBNDRPG 74
  ending an ILE program 143
  family of ILE compilers 66
  interlanguage calling considerations 180
  interlanguage calls 178
  internal structure of program 122
  overview 66
  program call 69
  program creation 67
  program creation strategies 71, 73, 75
  program management 68
interlanguage calls 178
Intersystem Communications Function (ICF) 382

J

Java
  calling Java from RPG 214
  calling Java methods 214
  calling RPG from Java 220
  calling RPG programs using PCML 239
  coding errors 223
  Java Virtual Machine (JVM) 225
  native methods 220
  prototyping 211
  runtime errors 233
JNI functions, wrappers for 225
JSON 210

K

key
  composite 346
  for a record or a file 345
  partial 346
key field information in compiler listing 476
keyed processing
  access path 343
  indexed file 348
  record-address limits file 350
  sequential-within-limits 358
  unexpected results 368
keyed-sequence access path 343
keywords
  *OMIT 171
  DDS 341
  EXPORT 125
keywords *(continued)*
  for continuation line
    CLEAR 384
    HELP 384
    HOME 384
    PRINT 384
    ROLLDOWN 384
    ROLLUP 384
  for display device file
    CLEAR 384
    HELP 384
    HOME 384
    PRINT 384
    ROLLDOWN 384
    ROLLUP 384
  NOOPT 128, 298

L

LANGID parameter
  CRTBNDRPG command 102, 457
  CRTRPGMOD command 116
languages, ILE 66
last record (LR) indicator
  used to end a program/procedure 183, 184
length of record in a file, conversion considerations 427
level checking 333
library, creating 93
LICOPT parameter
  CRTBNDRPG command 461
limitations of the XML parser 208
limits records 343
listing view, creating 246
listing, binder
  as maintenance resource 127
  basic 138
  creating 127
  determining exports in service program 130
  sections of 127
listing, compiler
  additional diagnostic messages 111
  browsing using SEU 112
  coordinating listing options with debug view options 112
  correcting compilation errors 110
  correcting run-time errors 112
  default information 108
  in-line diagnostic messages 111
  indenting structured operations 109
  obtaining 108
  reading 466
  sample listing 468
  sections of 108, 468
  specifying the format of 109
  using 107
  using as documentation 112
local variable
  in formatted dump 324
locking
  file 337
  read without locking 338
  record locking wait time 338
  retry on timeout 338
  standalone 338
  under commitment control 364

locking *(continued)*
- UNLOCK 338

log file
- about 427
- DDS for 440
- using 440

LOGFILE parameter 432
logical file
- general 342
- multi-format 342

LOGMBR parameter 433
long names
- in compiler listing 476

loop, avoiding in an error subroutine 306

M

main procedure
- coding considerations 90
- overview 79
- returning from 183
- scope of files 122

maintaining OPM compatibility 105, 145
managing activation groups 144
managing dynamically-allocated storage 146
managing programs 68
managing run-time storage 146
managing the default heap using RPG operations 148
manual code conversion 442
MCH3601 423
memory management operations
- %ALLOC built-in function 148
- %REALLOC built-in function 148
- ALLOC (allocate storage) operation code 148
- DEALLOC (free storage) operation code 148
- REALLOC (reallocate storage with new length) operation code 148

message summary in compiler listing 479
messages
- additional diagnostic 111
- exception
  - example 296
  - types of 291
  - unhandled 295
- in-line diagnostic 111
- inquiry
  - replying to 143

migrating to ILE RPG 425
modifying a module 127
module
- about 115
- behavior of bound ILE RPG 122
- binding into a program 122
- binding multiple 125
- changing optimization level 128
- changing while debugging 251
- creating 115
- creating a NOMAIN module 118
- CRTRPGMOD command 116
- determining the entry module 124
- different debug views 252
- effect of debug data on size 244
- information in dump listing 318
- modifying and rebinding 127

module *(continued)*
- overview of multiple-procedure module 79
- preparing for debugging 244
- reducing size 129
- related CL commands 122
- relationship to program 122
- removing observability 129
- replacing in a program 128
- viewing source while debugging 250

module creation
- general discussion 115
- using CRTRPGMOD 116
- using CRTRPGMOD defaults 118

module observability 129
MODULE parameter
- CRTBNDRPG command 451
- CRTRPGMOD command 116

MONITOR group 299
MQSeries 210
multiple devices attached to application program 366
multiple-device file
- WORKSTN 391

N

named activation group 144
National Language Support (NLS) of source debugger 286
nested exceptions 295
no debug data 244
NOMAIN module
- coding considerations 89
- creating 118

nonkeyed processing 361
NOOPT keyword
- and handling exceptions 298
- maintaining current values while debugging 242
- program optimization level 128

normal program/procedure end 183
NOT
- Behavioral difference between ILE RPG and RPG/400 421

null value support
- displaying null-capable fields 280

O

observability 129
obtaining a compiler listing 108
obtaining conversion reports 434
OFL 307
omitted parameters
- *OMIT 171

ON-ERROR group 299
one-step process of program creation 101
online information
- for create commands 451
- for ILE source 243

open data path
- sharing 339

operation codes
- allowed with DISK file 362, 363
- allowed with PRINTER file 370
- allowed with sequential file 378

operation codes *(continued)*
  allowed with SPECIAL file 380
  general discussion 59
operational descriptors
  definition 170
  example 132
OPM compatibility, maintaining 105, 145
OPM default activation group
  running in 145
optimization
  definition 128
  effect on fields when debugging 242
  exception handling considerations 298
  level of
    changing an object's 128
    checking 128
OPTIMIZE parameter
  CRTBNDRPG command 102, 455
  CRTRPGMOD command 116
OPTION parameter
  coordinating listing and debug view options 246
  coordinating with debug view options 112
  CRTBNDRPG command 102, 453
  CRTRPGMOD command 116
  using 108, 113
OPTIONS keyword
  *NOPASS 171
  *OMIT 171
order of evaluation
  on prototyped call 178
output
  specifications
    program-described WORKSTN file 389
output buffer positions, in compiler listing 474
output file 390
OUTPUT parameter
  CRTBNDRPG command 102, 455
  CRTRPGMOD command 116
  using 108
output record
  blocking 347
output specifications
  example 62
  general description 57
  program-described WORKSTN file 389
  with external descriptions 332
output spooling 340
overflow
  indicators 371
  page 370
overflow indicators
  conditioning output 371
  examples 373
  fetch-overflow logic 373
  general description 371
  presence or absence of 371
  relation to program cycle 373
  setting of 373
  with PRINTER file 370
overrides, file
  example 337
  general discussion 336, 361
  indicated in compiler listing 469
overriding external description 331

P

page headings 109
page number, in PRINTER file 370
page overflow, in PRINTER file 370
parameter descriptions
  CRTBNDRPG command 451
  CRTRPGMOD command 466
  CVTRPGSRC command 430
parameter list
  created by PARM 182
  identifying 164
  rules for specifying 182
parameter table
  CRTBNDRPG command 102
  CRTRPGMOD command 116
  CVTRPGSRC command 429
parameters
  checking number passed 172
  match data type requirements 170
  omitted 171
  operational descriptors 170
  passing 164
  passing using the CL CALL command 139
  specifying 181
PARM (identify parameters) operation code
  *OMIT 171
  rules for specifying 182
  using 181
partial key 346
parts of an ILE RPG program 60
passing file parameters 178
passing parameters
  by read-only reference 168
  by reference 167, 168
  by value 168
  checking number passed 172
  example 139
  match data type requirements 170
  omitted parameters 171
  operational descriptors 170
  overview 164
  passing less data 177
  passing methods for ILE languages 178
  using PARM 181
  using PLIST 182
  using the CL CALL command 139
PCML
  see Program call Markup Language 239
percolate an exception
  using a condition handler 308
percolation of an exception 291
performance considerations
  subroutines vs. subprocedures 132
performance tips
  call for LR-on 422
  program call 184
performing a quick conversion 433
performing a trial conversion 434
PGM parameter
  CRTBNDRPG command 102
physical file 342
PLIST (identify a parameter list) operation code
  *ENTRY PLIST 182

PLIST (identify a parameter list) operation code *(continued)*
    using 182
PREFIX keyword 330
preparing a program for debugging 244
Presentation Graphics Routines (PGR) 187
preventing printing over perforation 373
PRFDTA parameter
    CRTBNDRPG command 102, 461
    CRTRPGMOD command 116
    removing observability 129
print command key 384
PRINTER file
    access current line value 375
    fetch-overflow logic 373
    file operation codes allowed 370
    maximum number of files allowed in program 370
    modify forms control 375
    overflow indicators 370
    page overflow 370
    PRTCTL (printer control) 375
procedure
    abnormal ending 184
    calling 159
    dump information 318
    normal ending 183
    passing parameters 164
    procedure pointer call 160
    returning from 183
    returning without ending 184
    static procedure call 160
    stepping over 267
procedure pointer calls 160
processing methods
    consecutive 351
    for DISK file 350
    nonkeyed 361
    random-by-key 357
    relative-record-number 360
    sequential only 352, 361
    sequential-by-key 352
    sequential-within-limits 358
    WORKSTN file 385, 390
processing XML documents
    limitations of the XML parser 208
    XML parser error codes 206
program
    abnormal ending 184
    advanced ILE 77
    binding modules 122
    calling 159, 160
    calling using expressions 166
    calling using the CALL operation 180
    calling using the CALLP operation 166
    changing 127
    changing optimization level 128
    changing while debugging 251
    different debug views 252
    effect of debug data on size 244
    ending 143
    entering source 93
    entering SQL statements 97
    example 60
    freeing resources 146
    internal structure 122
program *(continued)*
    mixed-language 76
    multiple-module
        general creation strategy 75
    normal ending 183
    OPM-compatible
        creation method 71
        effect of ILE 72
        example 72
        program creation strategy 71, 78
    passing parameters 164
    preparing for debugging 244
    program entry procedure 122
    reducing size 129
    related CL commands 126
    removing observability 129
    returning from 183
    returning without ending 184
    running 138
    running from a menu-driven application 141
    running in the OPM default activation group 145
    running using a user-created command 142
    setting watch conditions 262
    single-language
        effect of ILE 74
    stepping into 268
    stepping over 267
    stepping through 266
    updating 128
    viewing source while debugging 250
program activation 144
program creation
    coding considerations 89
    examples of 104, 105, 125
    OPM-compatible
        creating 71
        strategy to avoid 78
    strategies for
        CRTPGM command 124
        ILE application using CRTRPGMOD 75
        OPM-compatible 71
        strategy to avoid 78
        using CRTBNDRPG 73
        using CRTRPGMOD and CRTPGM 115
        using the one-step process 101
program cycle
    commitment control 367
    fetch overflow logic 373
    general description 58
    last cycle 59
program entry procedure (PEP)
    and the call stack 161
    definition 122
    determining 124
program exception/error subroutine
    description 303
    example 304
program exception/errors
    avoiding a loop 306
    definition 293
    example 304, 308
    using a *PSSR subroutine 303
program management 68
program name

program name *(continued)*
  *FROMMBR parameter 432
program status data structure
  example 181, 304
  using in an error subroutine 303
program-described file
  as DISK file 347
  as WORKSTN file 388–390
  definition 327
  physical and logical files 342
  valid search arguments 348
program/procedure call
  abnormal program/procedure end 184
  call stack 161
  calling bindable APIs 186
  calling graphics 187
  calling procedures 160
  calling programs 160
  calling special routines 187
  fixed-form call 180
  free-form call 166
  interlanguage calls 178
  normal program/procedure end 183
  overview 160
  parameter passing methods 167
  recursive calls 161
  returning from a called program or procedure 183
  returning values 166
  returning without ending 184
  static calls 160
  using the CALL operation 180
  using the CALLB operation 180
  within ILE 69
program/procedure end
  abnormal end 184
  after system call 143
  normal end 183
  return overview 183
  returning without ending 184
  using bindable APIs 185
programming tips
  creating NOMAIN module 130
  setting subprocedure breakpoints 268
prologue section of compiler listing 468
prototype
  description 80
  using 165
prototyped call
  order of evaluation of parameters 178
prototyped program or procedure
  prototyped call 80
prototyping, Java methods 211
PRTCTL (printer control)
  example 376
  general information 375

Q

QUAL debug command
  definition 243
  ILE RPG 282
querying names of called programs/procedures 181

R

random-by-key processing
  example 357
  general discussion 357
reading a record 391
reading next record
  with WORKSTN subfile 386
REALLOC (reallocate storage with new length) operation code 148
Reallocate Storage (CEECZST) bindable API 69
rebinding 127
Reclaim Activation Group (RCLACTGRP) command
  deleting activation groups 145
  named activation groups 144
Reclaim Resources (RCLRSC) command
  ILE program 74
  OPM-compatible program 72
  to free storage 146
RECNO keyword
  with relative-record-number processing 360
record
  limits 350
  locking 338
  releasing 338
  valid keys 345
record address file
  conversion problems 437, 444
  relative-record number 350
  sequential-within-limits 350
  with limits records 350
  with relative record numbers 350
record format
  for a subfile 385
  ignoring 330
  renaming 329
  specifications for externally described file 342
record length of files, conversion considerations 427
record locking 338
recursion
  calling condition handlers 308
  recursive calls 90, 161
redirection, file
  definition 326
  general description 326
reducing object size 129, 244
Register Call Stack Entry Termination User Exit Procedure (CEERTX) 313
Register ILE Condition Handler (CEEHDLR) API 308
relative-record number 350
relative-record-number processing 360
releasing a locked record 338
removing breakpoints
  about 253
  all 262
  conditional job breakpoints 255
  conditional thread breakpoints 261
  unconditional job breakpoints 253
  unconditional thread breakpoints 255
  using statement numbers 259
removing objects from a debug session 249
removing observability 129
RENAME keyword 329
renaming field names 330

renaming fields 330
renaming record-format names 329
REPLACE parameter
    CRTBNDRPG command 102, 457
    CRTRPGMOD command 116
replacing modules in a program 128
reply list of messages
    adding to 143
    changing 143
replying to run-time inquiry messages 143
requirements of Conversion Aid 427
reserved words
    *CANCL 307
    *DETC 307
    *DETL 307
    *GETIN 307
    *OFL 307
    *TOTC 307
    *TOTL 307
resulting indicators (01-99, H1-H9, OA-OG, OV, L1-L9, LR, U1-U8, KA-KN, KP-KY, RT)
    as error indicators 298
resume point 307
Retrieve Operational Descriptor Information (CEEDOD)
    example 171
    operational descriptors 170
retry on a record lock timeout 338
RETURN (return to caller) operation code
    returning without ending 184
    role in abnormal end 184
    role in normal end 183
return (RT) indicator
    used to end a program/procedure 183, 184
return points, specifying in ENDSR 307
return status parameter 379
return value
    returning using expressions 166
RETURNCODE data area 113
returning from a called main procedure 183
returning from a called procedure 183
returning from a main procedure 183
returning from a subprocedure 185
returning using ILE bindable APIs 185
returning without ending 184
rolldown command key 384
rollup command key 384
root source view, creating 245
RPG IV
    behavioral differences between RPG III 421
    converting to 71, 73, 425
    overview 57
    unsupported RPG III features 442
run-time array
    allocating storage during run-time 153
run-time errors, correcting with a compiler listing 112
run-time inquiry messages, replying to 143
run-time storage, managing 146
running a program
    differences between ILE RPG and OPM RPG/400 421
    from a menu-driven application 141
    in the OPM default activation group 145
    overview 138
    using a user-created command 142
    using the CL CALL command 139

S

scope
    of files 122
screen design aid (SDA) 141
search argument
    externally described file
        description 345
        referencing a partial key 346
        valid 345
    program-described file 348
SECLVL parameter 432
SEQ file
    example 378
    file operation codes allowed 378
    general description 378
    processing chart 378
    restrictions 378
    variable-length 378
sequence checking
    on input specifications 334
sequential file 349
sequential-by-key processing
    examples 352
    general discussion 352
sequential-only processing 351, 352
sequential-within-limits processing
    examples 358
    general discussion 358
serialized procedure 191
service program
    adding to a debug session 249
    binder language 134
    binding with CRTBNDRPG 105
    changing 131
    creating 129
    example 132
    in advanced application 77
    reasons for using 129
    reclaiming resources 146
    related CL commands 131
    sample binder listing 138
    strategies for creating 130
    updating 137
service program creation
    about 129
    strategies 130
SET debug command
    definition 243
SETLL
    exception MCH3601 423
setting breakpoints
    about 253
    conditional job breakpoints 255
    conditional thread breakpoints 261
    example 254, 256
    unconditional job breakpoints 253
    unconditional thread breakpoints 255
    using statement numbers 259
setting debug options 248
sharing an open data path for a file 339
sort sequence
    affect of SRTSEQ parameter 341
    ALTSEQ table in compiler listing 475

sort sequence *(continued)*
  debug considerations 258
source debugging
  adding an object to a session 249
  built-in functions
    %ADDR 280
    %INDEX 280
    %SUBSTR 280
    %VARS 280
    changing values using %SUBSTR 282
    examples 281
    general discussion 280
  changing field values 282
  changing modules while debugging 251
  coordinating with listing options 112
  creating a program for debugging 104
  differences between ILE RPG and OPM RPG/400 422
  displaying attributes of 284
  displaying data addressed by pointers 278
  displaying data and expressions 271
  displaying fields as hexadecimal values 277
  displaying fields in character format 277
  displaying fields in UCS-2 format 277
  displaying fields in variable-length format 278
  displaying indicators 276
  displaying multiple-occurrence data structures 274
  displaying the contents of a table 274
  displaying the contents of an array 273
  general discussion 241
  National Language Support 286
  NLSS considerations 258
  obtaining a formatted dump 317
  OPM program limit in debug session 249
  optimization effects 128, 242
  overview 69
  preparing a program 244
  removing an object from a session 249, 250
  rules for assigning values using EVAL 282
  setting and removing breakpoints 253
  setting debug options 248
  setting watch conditions 262
  starting the source 247
  stepping through 266
  unexpected results 273
  updating production files 248
  viewing shorthand names 286
  viewing source 250
source entry utility (SEU)
  browsing a compiler listing 112
  entering source 94
source from a data file, converting 435
source member types, conversion of 426
source physical file, creating 93
source program
  converting all members 433
  converting auto report source members 435
  converting some members 434
  converting to ILE RPG 428
  entering into system 93
  entering SQL statements 97
  file and member names when converting 427
  record length of when converting 427
  source member types when converting 426
source section of compiler listing 470
special command keys 384
SPECIAL file
  deleting records from 380
  general discussion 379, 380
  valid file operations 380
special routines, calling 187
specification templates, inserting 435
specifications
  description of 57
  externally described file 329
  file description 329
  order 57
  record format 342
  types 57
specifying a return point 307
specifying an activation group 144
specifying error indicators 298
specifying the format of compiler listing 109
spooling 340
SRCFILE parameter
  CRTBNDRPG command 102, 451
  CRTRPGMOD command 116
SRCMBR parameter
  CRTBNDRPG command 102, 452
  CRTRPGMOD command 116
SRTSEQ parameter
  affect on key comparisons 341
  CRTBNDRPG command 102, 456
  CRTRPGMOD command 116
  debug considerations 258
stack, call 161, 291
Start Debug (STRDBG) command
  Update Production files (UPDPROD) parameter 248
starting commitment control 364
starting the ILE source 247
statement view
  creating 246
  using for debug 259
static calls 69, 160
static procedure call 160
status codes
  data management errors 424
STEP debug command
  definition 243
  into 268
  over 267
stepping while debugging
  into a program or procedure 268
  over a program or procedure 267
  through a program 266
STGMDL parameter
  CRTBNDRPG command 460
  CRTRPGMOD command 466
storage management
  allocating during run-time 153
  dynamic storage 146
  managing run-time 146
storage model
  inherit storage model 188
  recommendations 188
  single-level storage model 188
  teraspace storage model 188
strategies for creating ILE programs 71
STRSEU (edit source) command 94

structured operations
    indenting 109
subfields
    for file information data structure 319, 321
    for program status data structure 318
    for PRTCTL 376
subfiles
    control-record format 385
    descriptions 385
    examples 387
    file operation codes allowed with 386
    general description 385, 386
    record format 385
    uses of 387
subprocedures
    coding considerations 90
    debugging 270
    example 62
    information in compiler listing 478
    local data in dump listing 324
    logic flow 59
    overview 79
    returning from 185
    scope of files 122
    stepping into 268
    stepping over 267
SUBR23R3 (message retrieval) 187
SUBR40R3 (manipulating Double Byte Characters variables) 187
SUBR41R3 (manipulating Double Byte Characters variables) 187
subroutines
    avoiding a loop 306
    calling SUBR routines 187
    error 300
    file error (INFSR) 301
    program error (*PSSR) 303
substring of character or graphic literal
    ILE debug built-in %SUBSTR 281
summary tables
    file operation codes allowed with
        DISK 361
        PRINTER 370
        sequential 378
        SPECIAL 380
        WORKSTN 390
    sequential file processing 378
    SPECIAL file processing 380
syntax diagrams
    CRTBNDRPG command 449
    CRTRPGMOD command 464
    CVTRPGSRC command 429
    interpreting 448
system functions
    spooling 340
system reply list
    adding to 143
    changing 143

T

table
    displaying while debugging 274
table of parameters
table of parameters *(continued)*
    CRTBNDRPG command 102
    CRTRPGMOD command 116
    CVTRPGSRC command 429
tape file 349
TBREAK debug command
    definition 243
    using 255, 261
templates, inserting specification 435
teraspace 188
test library, using 248
testing breakpoints 253
TEXT parameter
    CRTBNDRPG command 102, 452
    CRTRPGMOD command 116
TGTCCSID parameter
    CRTBNDRPG command 463
TGTRLS parameter
    CRTBNDRPG command 102, 459
    CRTRPGMOD command 116
THREAD debug command
    definition 243
    using 255
threaded applications
    coding considerations 188
    debugging 253
    locking and unlocking procedures 190
    overview 70
TOFILE parameter 431, 434
TOMBR parameter 432, 433
TOTC 307
TOTL 307
trial conversion, performing 434
TRUNCNBR parameter
    CRTBNDRPG command 102, 458
    CRTRPGMOD command 116
two-step process of program creation 115
types of exception handlers 290

U

UCS-2 format
    displaying while debugging 277
    UCS-2 CCSID
        indicated in compiler listing 471
unblocking/blocking records 347
unconditional breakpoint
    definition 253
    setting 254
    setting and removing for job 253
    setting and removing for thread 255
    using statement numbers 259
unhandled escape message 296
unhandled exceptions 295
unhandled function check 297
Unregister ILE Condition Handler (CEEHDLU) API 308
unsigned integer format
    TRUNCNBR parameter 458
Update Program (UPDPGM) command
    using 128
updating a service program 137
user entry procedure (UEP)
    and the call stack 161
    role in program 122

user-created command, running an RPG program 142
USRPRF parameter on CRTBNDRPG
  CRTBNDRPG command 102, 457

V

valid file operations
  SPECIAL file 380
valid keys
  for file 345
  for records 345
variable-length format
  displaying while debugging 278
variable-length records 378
view, debug
  changing while debugging 252
  COPY source 245
  default 246
  definition 244
  listing 246
  root source 245
  statement 246
viewing source while debugging 250

W

WATCH debug command
  definition 243
  example 265
  setting conditions 262
Work with Reply List Entry (WRKRPLYE) command
  changing a system reply list 143
WORKSTN file
  definition 382
  examples 393
  externally described
    processing 385
  externally-described 382
  file operation codes allowed with 390
  function key indicators with 384
  multiple-device 391
  processing 390
  program-described
    calculation specifications 389
    combined file 390
    considerations 389
    general 388
    input file 390
    input specifications 389
    output file 390
    output specifications 389
    with format name 389
    without format name 390
  sample data maintenance program 397
  sample inquiry and search program 412
  sample inquiry program 394
  sample subfile processing program 406
  subfiles
    control-record format 385
    examples 387
    for display-device file 385
    record format 385
    uses of 387

WORKSTN file *(continued)*
  using 382

X

XML
  processing XML documents
    limitations of the XML parser 208
    XML parser error codes 206
XML parser error codes 206

SC09-2507-09