

AS/400 Advanced Series



ILE RPG for AS/400 Programmer's Guide

Version 4



AS/400 Advanced Series



ILE RPG for AS/400 Programmer's Guide

Version 4

Note!

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

Second Edition (February 1998)

This edition applies to Version 4, Release 2, Modification 0, of IBM Application System/400 Integrated Language Environment RPG for AS/400 (Program 5769-RG1) and to all subsequent releases and modifications until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product. Consult the latest edition of the applicable IBM system bibliography for current information on this product.

Changes or additions to the text and illustrations are indicated by a vertical line to the left of the change or addition.

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

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

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

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

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

© **Copyright International Business Machines Corporation 1998. All rights reserved.**

Note to U.S. Government Users — Documentation related to restricted rights — Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.

Contents

Notices	xix
Programming Interface Information	xix
Trademarks and Service Marks	xix
About This Guide	xxi
Who Should Use This Guide	xxi
What's New This Release?	xxii
Changes to this Guide Since V3R7	xxiii

ILE RPG Introduction	1
Chapter 1. Overview of the RPG IV Programming Language	3
RPG IV Specifications	3
Cycle Programming	4
Subprocedure logic	5
Indicators	5
Operation Codes	6
Example of an ILE RPG Program	6
Using the OS/400 System	12
Interacting with the System	12
AS/400 Tools	13
Application Development ToolSet	13
Application Development ToolSet Client Server for AS/400	14
Chapter 2. RPG Programming in ILE	17
Program Creation	17
Program Management	19
Program Call	19
Source Debugging	20
Bindable APIs	20
Multi-Threaded Applications	21
Chapter 3. Program Creation Strategies	23
Strategy 1: OPM-Compatible Application	23
Method	23
Example of OPM-Compatible Program	24
Related Information	25
Strategy 2: ILE Program Using CRTBNDRPG	25
Method	25
Example of ILE Program Using CRTBNDRPG	26
Related Information	27
Strategy 3: ILE Application Using CRTRPGMOD	27
Method	28
Single-Language ILE Application Scenario	28
Mixed-Language ILE Application Scenario	29
Advanced Application Scenario	30
Related Information	31
A Strategy to Avoid	31

Chapter 4. Creating an Application Using Multiple Procedures	33
A Multiple Procedures Module — Overview	33
Main Procedures and Subprocedures	33
Prototyped Calls	34
Example of Module with Multiple Procedures	36
The Entire ARRSRPT Program	41
Coding Considerations	45
General Considerations	45
Program Creation	46
Main Procedure Considerations	46
Subprocedure Considerations	46
For Further Information	47
Main Procedures	47
Subprocedures	47
Prototyped Call	48

Creating and Running an ILE RPG Application 49

Chapter 5. Entering Source Statements	51
Creating a Library and Source Physical File	51
Using the Source Entry Utility (SEU)	52
Using SQL Statements	55

Chapter 6. Creating a Program with the CRTBNDRPG Command	57
Using the CRTBNDRPG Command	57
Creating a Program for Source Debugging	59
Creating a Program with Static Binding	60
Creating an OPM-Compatible Program Object	61
Using a Compiler Listing	63
Obtaining a Compiler Listing	63
Customizing a Compiler Listing	64
Correcting Compilation Errors	66
Correcting Run-time Errors	68
Using a Compiler Listing for Maintenance	69
Accessing the RETURNCODE Data Area	70

Chapter 7. Creating a Program with the CRTRPGMOD and CRTPGM Commands	73
Creating a Module Object	73
Using the CRTRPGMOD Command	74
Creating a Module for Source Debugging	78
Additional Examples	80
Behavior of Bound ILE RPG Modules	80
Related CL Commands	80
Binding Modules into a Program	81
Using the CRTPGM Command	82
Additional Examples	85
Related CL Commands	85
Using a Binder Listing	85
Changing a Module or Program	86
Using the UPDPGM Command	87
Changing the Optimization Level	87
Removing Observability	88

Reducing an Object's Size	88
Chapter 8. Creating a Service Program	91
Service Program Overview	91
Strategies for Creating Service Programs	92
Creating a Service Program Using CRTSRVPGM	92
Changing A Service Program	93
Related CL commands	93
Sample Service Program	94
Creating the Service Program	97
Binding to a Program	98
Updating the Service Program	100
Sample Binder Listing	100
Chapter 9. Running a Program	103
Running a Program Using the CL CALL Command	103
Passing Parameters using the CL CALL Command	103
Running a Program From a Menu-Driven Application	106
Running a Program Using a User-Created Command	108
Replying to Run-Time Inquiry Messages	108
Ending an ILE Program	109
Managing Activation Groups	109
Specifying an Activation Group	110
Running in the OPM Default Activation Group	111
Maintaining OPM RPG/400 and ILE RPG Program Compatibility	111
Deleting an Activation Group	112
Reclaim Resources Command	112
Managing Dynamically-Allocated Storage	113
Managing the Default Heap Using RPG Operations	113
Heap Storage Problems	118
Managing Your Own Heap Using ILE Bindable APIs	120
Chapter 10. Calling Programs and Procedures	129
Program/Procedure Call Overview	129
Calling Programs	130
Calling Procedures	130
The Call Stack	131
Recursive Calls	132
Parameter-Passing Considerations	133
Using a Prototyped Call	135
Using the CALLP Operation	135
Calling within an Expression	136
Examples of Free-Form Call	137
Passing Prototyped Parameters	137
Parameter Passing Styles	137
Using Operational Descriptors	140
Omitting Parameters	141
Checking for the Number of Passed Parameters	143
Passing Less Data Than Required	148
Order of Evaluation	148
Interlanguage Calls	149
Using the Fixed-Form Call Operations	150
Examples of CALL and CALLB	151
Passing Parameters Using PARM and PLIST	151

Returning from a Called Program or Procedure	153
Returning from a Main Procedure	153
Returning from a Subprocedure	156
Returning using ILE Bindable APIs	156
Using Bindable APIs	157
Examples of Using Bindable APIs	158
Calling a Graphics Routine	158
Calling Special Routines	158

Debugging and Exception Handling 161

Chapter 11. Debugging Programs	163
The ILE Source Debugger	163
Debug Commands	164
Preparing a Program for Debugging	166
Creating a Root Source View	167
Creating a COPY Source View	168
Creating a Listing View	168
Creating a Statement View	169
Starting the ILE Source Debugger	169
STRDBG Example	170
Setting Debug Options	171
Adding/Removing Programs from a Debug Session	172
Example of Adding a Service Program to a Debug Session	172
Example of Removing ILE Programs from a Debug Session	173
Viewing the Program Source	174
Viewing a Different Module	175
Changing the View of a Module	176
Setting and Removing Breakpoints	178
Setting and Removing Unconditional Job Breakpoints	179
Setting and Removing Unconditional Thread Breakpoints	180
Setting and Removing Conditional Job Breakpoints	181
National Language Sort Sequence (NLSS)	184
Setting and Removing Job Breakpoints Using Statement Numbers	185
Setting and Removing Conditional Thread Breakpoints	187
Removing All Job and Thread Breakpoints	187
Setting and Removing Watch Conditions	188
Characteristics of Watches	188
Setting Watch Conditions	189
Displaying Active Watches	191
Removing Watch Conditions	192
Example of Setting a Watch Condition	192
Stepping Through the Program Object	194
Stepping Over Call Statements	195
Stepping Into Call Statements	196
Displaying Data and Expressions	199
Changing the Value of Fields	208
Displaying Attributes of a Field	210
Equating a Name with a Field, Expression, or Command	210
Source Debug National Language Support for ILE RPG	211
Sample Source for Debug Examples	211

Chapter 12. Handling Exceptions 217

Exception Handling Overview	217
ILE RPG Exception Handling	220
Using Exception Handlers	223
Exception Handler Priority	223
Nested Exceptions	223
Unhandled Exceptions	223
Optimization Considerations	226
Using RPG-Specific Handlers	227
Specifying Error Indicators or the 'E' Operation Code Extender	227
Using an Error Subroutine	228
Specifying a Return Point in the ENDSR Operation	238
ILE Condition Handlers	239
Using a Condition Handler	239
Using Cancel Handlers	244
Problems when ILE CL Monitors for Notify and Status Messages	247
Chapter 13. Obtaining a Dump	251
Obtaining an ILE RPG Formatted Dump	251
Using the DUMP Operation Code	251
Example of a Formatted Dump	252

Working with Files and Devices 259

Chapter 14. Defining Files	261
Associating Files with Input/Output Devices	261
Naming Files	263
Types of File Descriptions	263
Using Files with External-Description as Program-Described	264
Example of Some Typical Relationships between Programs and Files	264
Defining Externally Described Files	265
Renaming Record-Format Names	266
Renaming Field Names	266
Ignoring Record Formats	266
Using Input Specifications to Modify an External Description	267
Using Output Specifications	269
Level Checking	270
Defining Program-Described Files	271
Data Management Operations and ILE RPG I/O Operations	271
Chapter 15. General File Considerations	273
Overriding and Redirecting File Input and Output	273
Example of Redirecting File Input and Output	274
File Locking	275
Record Locking	276
Sharing an Open Data Path	277
Spooling	278
Output Spooling	279
SRTSEQ/ALTSEQ in an RPG Program versus a DDS File	279
Chapter 16. Accessing Database Files	281
Database Files	281
Physical Files and Logical Files	281
Data Files and Source Files	281

Using Externally Described Disk Files	282
Record Format Specifications	282
Access Path	282
Valid Keys for a Record or File	285
Record Blocking and Unblocking	287
Using Program-Described Disk Files	288
Indexed File	288
Sequential File	290
Record Address File	290
Methods for Processing Disk Files	291
Consecutive Processing	292
Sequential-by-Key Processing	293
Random-by-Key Processing	299
Sequential-within-Limits Processing	300
Relative-Record-Number Processing	303
Valid File Operations	304
Using Commitment Control	307
Starting and Ending Commitment Control	307
Specifying Files for Commitment Control	309
Using the COMMIT Operation	309
Specifying Conditional Commitment Control	311
Commitment Control in the Program Cycle	312
DDM Files	313
Using Pre-V3R1 DDM Files	313
Chapter 17. Accessing Externally Attached Devices	315
Types of Device Files	315
Accessing Printer Devices	316
Specifying PRINTER Files	316
Handling Page Overflow	316
Using the Fetch-Overflow Routine in Program-Described Files	320
Changing Forms Control Information in a Program-Described File	323
Accessing Tape Devices	325
Accessing Display Devices	326
Using Sequential Files	326
Specifying a Sequential File	326
Using SPECIAL Files	327
Example of Using a Special File	329
Chapter 18. Using WORKSTN Files	331
Intersystem Communications Function	331
Using Externally Described WORKSTN Files	331
Specifying Function Key Indicators on Display Device Files	334
Specifying Command Keys on Display Device Files	334
Processing an Externally Described WORKSTN File	335
Using Subfiles	335
Using Program-Described WORKSTN Files	338
Using a Program-Described WORKSTN File with a Format Name	339
Using a Program-Described WORKSTN File without a Format Name	340
Valid WORKSTN File Operations	341
EXFMT Operation	342
READ Operation	342
WRITE Operation	342
Multiple-Device Files	342

Chapter 19. Example of an Interactive Application	345
Database Physical File	345
Main Menu Inquiry	346
MAINMENU: DDS for a Display Device File	346
CUSMAIN: RPG Source	347
File Maintenance	349
CUSMSTL1: DDS for a Logical File	349
MNTMENU: DDS for a Display Device File	350
CUSMNT: RPG Source	352
Search by Zip Code	358
CUSMSTL2: DDS for a Logical File	358
SZIPMENU: DDS for a Display Device File	359
SCHZIP: RPG Source	361
Search and Inquiry by Name	364
CUSMSTL3: DDS for a Logical File	364
SNAMMENU: DDS for a Display Device File	365
SCHNAM: RPG Source	366

Appendixes	371
-------------------------	-----

Appendix A. Behavioral Differences Between OPM RPG/400 and ILE RPG for AS/400	373
Compiling	373
Running	374
Debugging and Exception Handling	374
I/O	375
DBCS Data in Character Fields	377
Appendix B. Using the RPG III to RPG IV Conversion Aid	379
Conversion Overview	379
File Considerations	380
The Log File	381
Conversion Aid Tool Requirements	381
What the Conversion Aid Won't Do	382
Converting Your Source	382
The CVTRPGSRC Command	383
Converting a Member Using the Defaults	387
Converting All Members in a File	388
Converting Some Members in a File	388
Performing a Trial Conversion	388
Obtaining Conversion Reports	389
Converting Auto Report Source Members	389
Converting Source Members with Embedded SQL	390
Inserting Specification Templates	390
Converting Source from a Data File	390
Example of Source Conversion	390
Analyzing Your Conversion	393
Using the Conversion Report	394
Using the Log File	396
Resolving Conversion Problems	398
Compilation Errors in Existing RPG III Code	398
Unsupported RPG III Features	398
Use of the /COPY Compiler Directive	398

Use of Externally Described Data Structures	401
Run-time Differences	403
Appendix C. The Create Commands	405
Using CL Commands	405
How to Interpret Syntax Diagrams	405
CRTBNDRPG Command	406
Description of the CRTBNDRPG Command	408
CRTRPGMOD Command	419
Description of the CRTRPGMOD command	421
Appendix D. Compiler Listings	423
Reading a Compiler Listing	424
Prologue	424
Source Section	426
Additional Diagnostic Messages	430
Output Buffer Positions	430
/COPY Member Table	431
Compile-Time Data	431
Key Field Information	433
Cross-Reference Table	433
External References List	434
Message Summary	435
Final Summary	435
Code Generation and Binding Errors	436
Bibliography	437
Index	441

Figures

1.	RPG Program Logic Cycle	4
2.	DDS for Employee physical file	7
3.	DDS for TRANSACT physical file	7
4.	A Sample Payroll Calculation Program	10
5.	Program Creation in ILE	18
6.	OPM-Compatible Application	24
7.	ILE Program Using CRTBNDRPG	26
8.	Single-Language Application Using CRTRPGMOD and CRTPGM	29
9.	Mixed-Language Application	30
10.	Advanced Application	31
11.	Scenario to Avoid	32
12.	An ILE RPG module with Multiple Procedures	34
13.	Prototype for FmtCust Procedure	35
14.	Prototype for NumToChar Procedure	35
15.	Calling the FmtCust Procedure	35
16.	Calling the NumToChar Procedure	36
17.	Components of the ARRSRPT Module	37
18.	Source for Subprocedure InArrears	38
19.	Source for Subprocedure FmtCust	39
20.	Source for module CVTPROCS, containing subprocedure NumToChar	40
21.	The ARRSRPT Application	41
22.	ILE RPG Complete Source for ARRSRPT Module	41
23.	Output for ARRSRPT	44
24.	DDS for CUSTFILE	44
25.	DDS for CISTRPT	45
26.	ILE RPG Record Length Breakdown	51
27.	Edit Display for a New Member	53
28.	Source for EMPRPT member	54
29.	DDS for EMPRPT	55
30.	SQL Statements in an ILE RPG Program	56
31.	Display Module Source display for EMPRPT	60
32.	A Sample Payroll Calculation Program	62
33.	Sample Source Part of the Listing with Indentation	66
34.	Sample Finger In-Line Diagnostic Messages	67
35.	Sample Non-Finger In-Line Diagnostic Messages	68
36.	Source for TRANSSVC member	76
37.	Source for TRANSP /COPY member	78
38.	Source for TRANSRPT module	79
39.	DDS for TRNSDTA	80
40.	Structure of Program TRPT	81
41.	File QSYSPRT for TRPT	85
42.	Source for Service Program CvtToHex	95
43.	Source for /COPY Member with Prototype for CvtToHex	97
44.	Source for Binder Language for CvtToHex	97
45.	Source for Test Program CVTHEXPGM	99
46.	Basic Binder listing for CVTHEXPGM	101
47.	ILE RPG Program that Requires Parameters at Run Time	104
48.	DDS for EMPRPT2	105
49.	Example of an Application Menu	106
50.	Data Description Specification of an Application Menu	107

51.	Source for Menu Program	107
52.	Memory Management - Build a Linked List of Names	114
53.	Heap Storage Misuse	119
54.	/COPY file DYNARRI containing prototypes for DYNARRAY module	120
55.	Global variables and local prototypes for DYNARRAY	121
56.	DYNARRAY Subprocedures	123
57.	Sample module using procedures in DYNARRAY	127
58.	Program and Procedure Calls on the Call Stack	131
59.	Three Modules, each with subprocedures	132
60.	Recursive Call Stack To Be Avoided	133
61.	Prototype for CVTCHR	136
62.	Calling a Prototyped Procedure within an Expression	136
63.	Prototype for Procedure DO_CALC with VALUE Parameters	139
64.	Procedure Interface Definition for DO_CALC Procedure	139
65.	Prototype for ILE CEE API CEETSTA with CONST Parameter	140
66.	Requesting Operational Descriptors for a Prototyped Procedure	141
67.	Prototype for System API QCMDEXC with Optional Parameter	142
68.	Source for procedure FMTADDR	144
69.	Source for /COPY member with Prototype for Procedure FMTADDR	145
70.	Source for procedure PRTADDR	146
71.	Prototype for System API QCMDEXC with *VARSIZE Parameter	148
72.	Sample Call Syntax for ILE Bindable APIs	157
73.	Display Module Source display for program DEBUGEX	171
74.	Adding an ILE Service Program to a Debug Session	173
75.	Removing an ILE Program from a Debug Session	174
76.	Changing to a Different Module	176
77.	Source View of ILE C procedure cproc	176
78.	Changing a View of a Module	177
79.	Setting an Unconditional Job Breakpoint	180
80.	Setting a Conditional Job Breakpoint	182
81.	Setting a Conditional Job Breakpoint Using the BREAK Command	183
82.	Setting a Breakpoint Using Statement View	186
83.	Example of a Work with Watch Display	190
84.	Example of a Display Watch Window	190
85.	Example of a Display Debug Watch Display	191
86.	Example of Message Stating WATCH was Successfully Set	193
87.	Example of a Display Module Source Panel	194
88.	Display Module Source display of DBGEX Before Stepping Into RPGPGM	197
89.	Stepping into RPGPGM	198
90.	Stepping into Subprocedure Switch	199
91.	Displaying a Field using the EVAL debug command	200
92.	Sample EVAL commands based on Module DBGEX	201
93.	Sample EVAL commands for an Array	202
94.	Sample EVAL commands for a Table	203
95.	Using EVAL with Data Structures	204
96.	Sample EVAL commands for an Array	205
97.	Examples of %SUBSTR using DBGEX	207
98.	Examples of Changing the Values of Fields based on DBGEX	209
99.	Examples of Displaying the Attributes of Fields based on DBGEX	210
100.	Source for Module DBGEX	212
101.	Source for OPM Program RPGPGM	216
102.	Source for C Procedure cproc	216
103.	Call Stack and Exception Message Percolation	219

104. Scenario for Unhandled Escape Message	224
105. Scenario for Unhandled Function Check	226
106. Example of File Exception Handling	231
107. Example of *PSSR Subroutine in Main Procedure	234
108. Example of Subprocedure *PSSR Subroutine with GOTO	235
109. Example of Subprocedure *PSSR Subroutine with RETURN	236
110. Avoiding a Loop in an Error Subroutine	237
111. Source for Condition Handler for Out-of-Bounds Substring Error	240
112. Source for Registering a Condition Handler	242
113. Enabling and Coding a Cancel Handler for a Subprocedure	245
114. Output from CANHDLR program	247
115. ILE Condition Handler that Ignores CPF4906	249
116. How to Ignore Status and Notify Messages	250
117. Program Status Information section of Formatted Dump	252
118. Feedback Areas section of Formatted Dump	254
119. Information Provided for Full-Formatted Dump	255
120. Data section of Formatted Dump	256
121. Identifying a Display File in an RPG Program	262
122. Associating a file name with a display file description	262
123. Associating a file name with a display file description	263
124. Typical Relationships between an RPG Program and Files on the AS/400 System	264
125. RENAME Keyword for Record Format Names in an Externally Described File	266
126. Prefix Keyword for Record Format Names in an Externally Described File	266
127. IGNORE Keyword for Record Formats in an Externally Described File	267
128. Overriding and Adding RPG Functions to an External Description	268
129. Adding RPG Functions to an External Description	268
130. Output Specifications for an Externally Described File	270
131. Overriding File Input and Output Example	273
132. Redirecting File Input and Output Example	275
133. Output Spooling Example	279
134. Example of the Data Description Specifications for a Database File	283
135. Example of a field Reference File	284
136. DDS and corresponding File-Description Specification Detail Flow of RPG IV Exception/Error Handling	289
137. (Part 1 of 2). Using Data Description Specifications to Define the Access Path (Composite Key) for an Indexed File	289
138. (Part 2 of 2). Using Data Description Specifications to Define the Access Path (Composite Key) for an Indexed File	290
139. DDS for database file EMPMST (physical file)	294
140. DDS for database file TRWEEK (physical file)	294
141. DDS for database file EMPL1 (logical file)	294
142. Sequential-by-Key Processing, Example 1	295
143. Sequential-by-Key Processing, Example 2	296
144. Sequential-by-Key Processing, Example 5	297
145. DDS for database file CHANGE (physical file)	299
146. Random-by-Key Processing of an Externally Described File	300
147. Sequential-within-Limits Processing of an Externally Described File	302
148. DDS for record address file LIMITS (physical file)	302
149. Sequential-within-Limits Processing of a Program-Described File	303
150. Example of Using Commitment Control	311
151. Example of Using Conditional Commitment Control	312

152. Printing a Heading on Every Page	319
153. Printing a Field on Every Page	320
154. Overflow Printing: Setting of the Overflow Indicator	321
155. Use of Fetch Overflow	322
156. Example of the PRTCTL Option	325
157. SEQ Device	327
158. SPECIAL Device	329
159. User-written program USERIO	330
160. Example of the Data Description Specifications for a Display Device File	333
161. Subfile Display	336
162. Data Description Specifications for a Subfile Record Format	337
163. Data Description Specifications for a Subfile Control-Record Format	338
164. DDS for master database file CUSMST (physical file)	346
165. DDS for display device file MAINMENU	347
166. Source for module CUSMAIN	348
167. Customer Main Inquiry prompt screen	349
168. DDS for logical file CUSMSTL1	349
169. DDS for display device file MNTMENU	350
170. Source for module CUSMNT	352
171. 'Customer File Maintenance' Display Mode prompt screen	355
172. 'Customer File Maintenance' Display Mode screen	356
173. 'Customer File Maintenance' Add Mode prompt screen	356
174. 'Customer File Maintenance' Add Mode prompt screen	357
175. 'Customer File Maintenance' Delete Mode prompt screen	357
176. 'Customer File Maintenance' Update Mode prompt screen	358
177. DDS for logical file CUSMSTL2	358
178. DDS for display device file SZIPMENU	359
179. Source for module SCHZIP	361
180. 'Customer Search by Zip' prompt screen	363
181. 'Customer Search by Zip' screen	364
182. DDS for logical file CUSMSTL3	364
183. DDS for display device file SNAMMENU	365
184. Source for module SCHNAM	366
185. 'Customer Search and Inquiry by Name' prompt screen	369
186. 'Customer Search and Inquiry by Name' information screen	370
187. 'Customer Search and Inquiry by Name' detailed information screen	370
188. RPG IV Record Length Breakdown	380
189. RPG III Source for TEST1	391
190. Converted (RPG IV) Source for TEST1	392
191. Command Summary of Sample Conversion Report	394
192. Sample Source Section of Conversion Report	395
193. Sample Message Summary of Conversion Report	395
194. Sample Final Summary of Conversion Report	396
195. DDS for model log file QARNCVTLG in library QRPGL	397
196. RPG III Source for TEST2	399
197. RPG III Source for COPYDS1	399
198. RPG III /COPY file with input fields only	400
199. RPG III Source with a renamed field	401
200. RPG IV source after converting source with input fields only	401
201. DDS for external data structure	402
202. RPG III source using external data structure with array	402
203. RPG IV source with two definitions for the array	402
204. Corrected RPG IV source with a single definition for the array	402
205. RPG III source with renamed and initialized external subfield	403

206. RPG IV source with two definitions for renamed subfield	403
207. Corrected RPG IV source with a single definition	403
208. Structure of a Syntax Diagram	405
209. Sample Prologue for CRTBNDRPG	425
210. Sample Source Part of the Listing	427
211. Sample Additional Diagnostic Messages	430
212. Output Buffer Position Table	430
213. Sample /COPY Member Table	431
214. Sample Compile-Time Data Section	432
215. Sample Key Field Information	433
216. Sample Cross-Reference Table	434
217. Sample External References	435
218. Sample Message Summary	435
219. Sample Final Summary	436

Tables

1.	Commonly Used CL Commands	13
2.	Programming Languages Supported on the AS/400	17
3.	CRTBNDRPG Parameters and Their Default Values Grouped by Function	58
4.	CRTRPGMOD Parameters and Their Default Values Grouped by Function	74
5.	Parameters for CRTPGM Command and their Default Values	83
6.	Sections of the Binder Listing based on DETAIL Parameter	86
7.	Parameters for CRTSRVPGM Command and their Default Values	93
8.	Parameter Passing Options	135
9.	RPG Parameter Passing Methods	149
10.	Meaning of Factor 1 and Factor 2 Entries in PARM Operation	152
11.	Debug Views	167
12.	Non-numeric Conditional Breakpoint Expressions	185
13.	Operation Codes Allowing Extender 'E' or an Error Indicator in Positions 73-74	228
14.	Correlation of RPG Device Types with AS/400 File Types	262
15.	Data Management Operations and the Corresponding RPG I/O Operation	272
16.	System Open Options Allowed with User Open Options	278
17.	Processing Methods for DISK Files	292
18.	Valid File Operations for Keyed Processing Methods (Random by Key, Sequential by Key, Sequential within Limits)	305
19.	Valid File Operations for Non-keyed Processing Methods (Sequential, Random by Relative Record Number, and Consecutive)	306
20.	AS/400 Device Files, Related CL commands, and RPG Device Name	315
21.	Results of the Presence or Absence of an Overflow Indicator	319
22.	Layout of ILE PRTCTL Data Structure	323
23.	Layout of OPM PRTCTL Data Structure	324
24.	Valid File Operation Codes for a Sequential File	326
25.	Valid File Operations for a SPECIAL File	329
26.	Valid File Operation Codes for a WORKSTN File	341
27.	Description of Each Module in the Interactive Application Example	345
28.	Source Member Types and their Conversion Status	380
29.	CVTRPGSRC Parameters and Their Default Values Grouped by Function	383
30.	Sections of the Compiler Listing	423

Notices

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

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, 500 Columbus Avenue, Thornwood, NY 10594, USA.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independent created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact IBM Canada Ltd., Department 071, 1150 Eglinton Avenue East, North York, Ontario M3C 1H7, Canada. Such information may be available, subject to appropriate terms and conditions, including in some cases payment of a fee.

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

Programming Interface Information

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

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

Trademarks and Service Marks

The following terms are trademarks of the International Business Machines Corporation in the United States or other countries or both:

400	IBMLink
AFP	Integrated Language Environment
AS/400	Operating System/400
C/400	OS/400
COBOL/400	PROFS
DB2	RPG/400
FORTRAN/400	System/36
GDDM	System/38
IBM	VisualAge

| Windows is a registered trademark of Microsoft Corporation.

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

| Registered trademarks and unregistered trademarks are denoted by ® and ™
| respectively.

About This Guide

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

This guide shows how to:

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

For information about other AS/400 publications, see either of the following:

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

For a list of related publications, see the “Bibliography” on page 437.

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 the AS/400 system.

Though this guide shows how to use the RPG IV in an ILE framework, it does not provide detailed information on RPG IV specifications and operations. For a detailed description of the language, see the *ILE RPG for AS/400 Reference*, SC09-2508 .

Before using this guide, you should:

- Know how to use applicable AS/400 menus and displays, or Control Language (CL) commands.

- Have the appropriate authority to the CL commands and objects described here.
- Have a firm understanding of ILE as described in detail in the *ILE Concepts*, SC41-5606.

What's New This Release?

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 OS/400 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.

Changes to this Guide Since V3R7

This V4R2 guide, *ILE RPG for AS/400 Programming Guide*, SC09-2507-01, differs in many places from the V3R7 guide, *ILE RPG/400 Programming Guide*, SC09-2074-01. Most of the changes are related to the enhancements that have been made since V3R7; others reflect minor technical corrections. To assist you in using this manual, technical changes and enhancements are noted with a vertical bar (|).

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

Chapter 1. Overview of the RPG IV Programming Language

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

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

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

RPG IV Specifications

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

There are 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 NOMAIN is specified on the control specification.

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

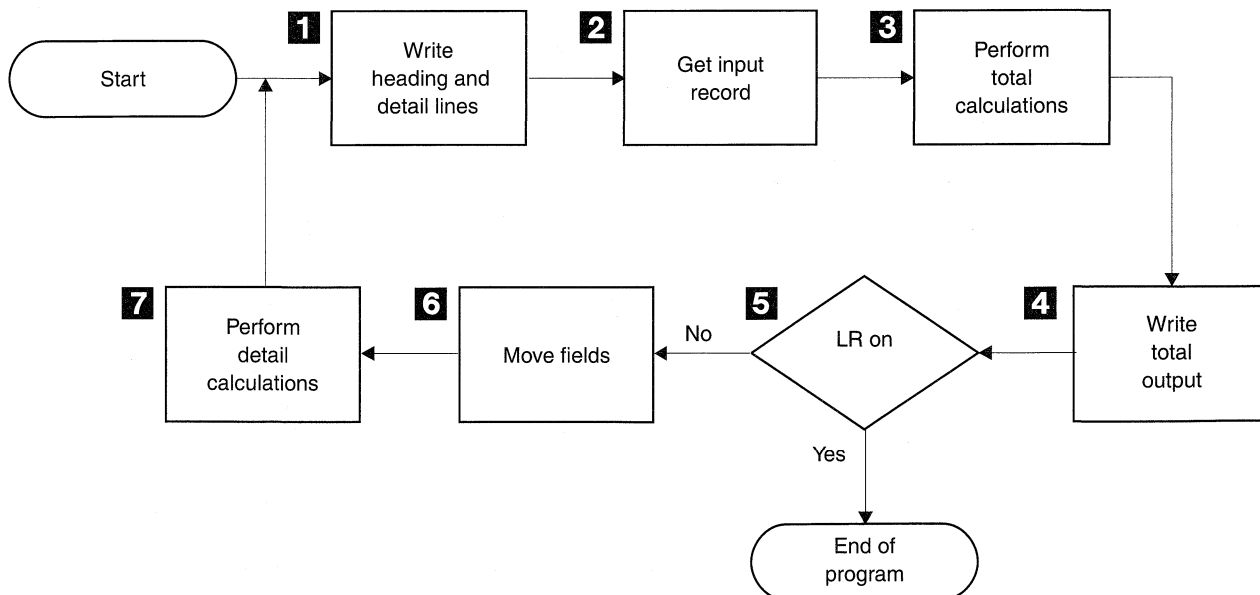


Figure 1. RPG Program Logic Cycle

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

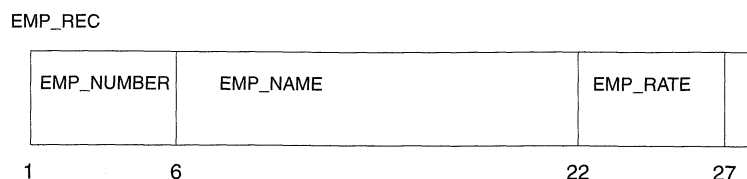
Example of an ILE RPG Program

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('EMPLOYEE NAME')
A      EMP_RATE        5 2     TEXT('EMPLOYEE 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:

TRN_REC

TRN_NUMBER	TRN_HOURS	TRN_BONUS
1	6	10

```

*.. 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 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  0 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 ...+...
CLON01Factor1+++++++Opcode(E)+Factor2+++++++Result+++++++Len++D+HiLoEq..

C   TRN_NUMBER   CHAIN   EMP_REC           99
C                   IF   NOT *IN99
C                   EVAL   PAY = CalcPay(EMP_RATE ; TRN_HOURS ;
C                               TRN_BONUS)
C                   ENDIF
```

The coding entries on the calculation specifications include:

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

Output Specifications

```

*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+...
OFile name++DF..N01N02N03Excnam++++B++A++Sb+Sa+.....
O.....N01N02N03Field+++++++YB.End++PCConstant/editword/DTformat

QSYSVRT H 1P 2 3
0 35 'PAYROLL REGISTER'
0 *DATE Y 60
0 H 1P 2
0 60 Heading1
0 H 1P 2
0 60 Heading2
0 D N1P99 2
0 TRN_NUMBER 5
0 EMP_NAME 24
0 EMP_RATE L 33
0 TRN_HOURS L 40
0 TRN_BONUS L 49
0 Pay 60 '$ 0. '
0 D N1P 99 2
0 TRN_NUMBER 5
0 35 '** NOT ON EMPLOYEE FILE **'
0 T LR
0 33 'END OF LISTING'

```

The output specifications describe what fields are to be written on the QSYSVRT 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)

* Determine any overtime hours to be paid.
C              IF          Hours > 40
C              EVAL        Overtime = (Hours - 40) * Rate * 1.5
C              EVAL        Hours = 40
C              ENDIF

* Calculate the total pay and return it to the caller
C              RETURN      Rate * Hours + Bonus + Overtime

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  0 F 80        PRINTER

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

```

Figure 4 (Part 1 of 3). A Sample Payroll Calculation Program

```

*-----*
* 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 (TRANSACTION), if the employee
* is found, compute the employee's pay and print the details.
*-----*
C   TRN_NUMBER  CHAIN      EMP_REC          99
C           IF      NOT *IN99
C           EVAL    PAY = CalcPay(EMP_RATE : TRN_HOURS :
C                                 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
*-----*
OQSYSPT  H   1P          2 3
0              35 'PAYROLL REGISTER'
0              *DATE      Y  60
0           H   1P          2
0              60 Heading1
0           H   1P          2
0              60 Heading2
0           D  N1PN99      2
0              TRN_NUMBER  5
0              EMP_NAME   24
0              EMP_RATE   L  33
0              TRN_HOURS  L  40
0              TRN_BONUS  L  49
0              Pay        60 '$  0.  '
0           D  N1P 99      2
0              TRN_NUMBER  5
0              35 '** NOT ON EMPLOYEE FILE **'
0           T   LR
0              33 'END OF LISTING'

```

Figure 4 (Part 2 of 3). A Sample Payroll Calculation Program

```

*-----*
* 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)

* Determine any overtime hours to be paid.
C              IF          Hours > 40
C              EVAL        Overtime = (Hours - 40) * Rate * 1.5
C              EVAL        Hours = 40
C              ENDIF

* Calculate the total pay and return it to the caller
C              RETURN      Rate * Hours + Bonus + Overtime

P CalcPay      E

```

Figure 4 (Part 3 of 3). A Sample Payroll Calculation Program

Using the OS/400 System

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

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

Refer to the *Publications Reference*, for a complete list of publications that discuss the OS/400 system.

Interacting with the System

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

Commonly Used Control Language Commands

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

<i>Table 1. Commonly Used CL Commands</i>		
Action	CL command	Result
Using System Menus	GO MAIN	Display main menu
	GO INFO	Display help menu
	GO CMDRPG	List commands for RPG
	GO CMDCRT	List commands for creating
	GO CMDxxx	List commands for 'xxx'
Calling	CALL program-name	Runs a program
Compiling	CRTxxxMOD	Creates xxx Module
	CRTBNDxxx	Creates Bound xxx Program
Binding	CRTPGM	Creates a program from ILE modules
	CRTSRVPGM	Creates a service program
	UPDPGM	Updates a bound program object
Debugging	STRDBG	Starts ILE source debugger
	ENDDBG	Ends ILE source debugger
Creating Files	CRTPRTF	Creates Print File
	CRTPF	Creates Physical File
	CRTSRCPF	Creates Source Physical File
	CRTLF	Creates Logical File

AS/400 Tools

The AS/400 system offers a full set of tools that you may find useful for programming.

Application Development ToolSet

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

Application Development Manager

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

An application development team using ADM can:

- Define a flexible environment where production, testing, and maintenance can be managed simultaneously

- Organize several developers working on the same application
- Build (or compile) an application quickly and easily, compiling only those components that need compiling
- Create and maintain several versions of an application

Application Dictionary Services

Application Dictionary Services is an impact analysis tool that speeds up the analysis of applications. The tool stores the information about the application objects and their relationships to other application objects in a **dictionary**. A dictionary extracts the cross-reference information about all of the objects in an application's libraries, and saves it in a set of database files stored in a library. As application objects and their relationships to other objects change, the information in the dictionary is automatically updated.

You can use this feature of ADTS to:

- Determine the impact of changing a field
- Work with fields, files, or programs that would be affected by changing a field
- Re-create all of the objects affected by changing a field
- View the structure of an application
- Determine the field reference hierarchy
- Create, modify, or delete programs or files that are documented in a dictionary
- Modify any fields or records that are documented in a dictionary
- Examine the calling hierarchy

Application Development ToolSet Client Server for AS/400

ADTS CS (for Windows®) is a workstation product that includes two server access programs:

- CoOperative Development Environment/400 (CODE/400)
- VisualAge for RPG®

CODE/400 contains features to help edit, compile, and debug: RPG, ILE RPG, COBOL, ILE COBOL, Control Language (CL), ILE C, and ILE CL host source programs; design display, printer, and database host files; and manage the components that make up your application. This enhances program development and moves the program development workload off the host. The application, when built, runs on an AS/400. For RPG and ILE RPG application development and maintenance, CODE/400 provides:

- Language sensitive editing— includes token highlighting, format lines, a full suite of prompts, and online help.
- Incremental syntax checking— provides immediate error feedback as each line of source is entered
- Program verification— performs, at the workstation, the full range of syntax and semantic checking that the compiler does, without generating object code
- Program conversion— performs, at the workstation, an OPM to ILE RPG conversion

- A windowed environment for submitting host compiles and binds
- Source-level debugging
- A DDS design utility—allows you to easily change screens, reports, and data-base files
- Access to Application Dictionary Services.

VisualAge for RPG offers a visual development environment on the workstation platform for RPG application developers to develop, maintain, and document client/server applications. Applications can be edited, compiled, and debugged on your workstation. The applications, when built, are started on a workstation and can access AS/400 host data and other AS/400 objects. Its integrated components allow application developers to preserve their current skills and easily develop AS/400 RPG applications with graphical user interfaces.

If you want to learn more about CODE/400 and VisualAge for RPG, see the most current information available on the World Wide Web at:

<http://www.software.ibm.com/ad/varpg/>

Chapter 2. 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 the AS/400 system.

ILE is a recent approach to programming on the AS/400 system. It is the result of major enhancements to the AS/400 machine architecture and the OS/400 operating system. The ILE family of compilers includes: ILE RPG, ILE C, ILE COBOL, ILE CL, and VisualAge for C++. Table 2 lists the programming languages supported by the OS/400 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 2. Programming Languages Supported on the AS/400

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 shows the two approaches to program creation.

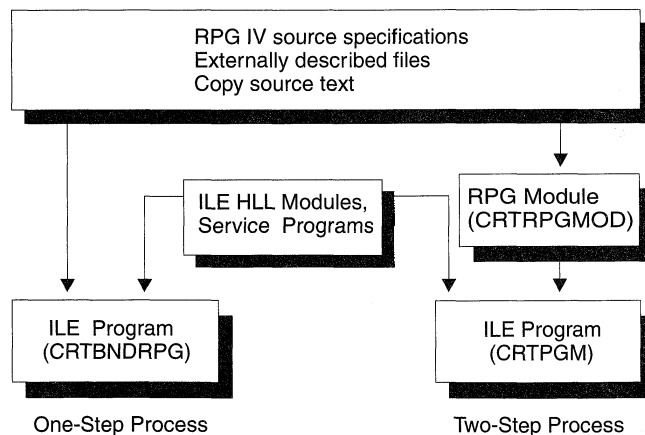


Figure 5. Program Creation in ILE

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 Chapter 6, “Creating a Program with the CRTBNDRPG Command” on page 57. For more information on the two-

step process, see Chapter 7, “Creating a Program with the CRTRPGMOD and CRTPGM Commands” on page 73. For more information on service programs, see Chapter 8, “Creating a Service Program” on page 91.

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 109.

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 113.

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 Chapter 9, “Running a Program” on page 103. For information on program/procedure call, refer to Chapter 10, “Calling Programs and Procedures” on page 129.

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 Chapter 11, “Debugging Programs” on page 163.

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 Chapter 9, “Running a Program” on page 103 and also the *System API Reference*.

Multi-Threaded Applications

The AS/400 now supports multi-threading. However, it is not supported in this release for ILE RPG procedures to run in a multi-threaded environment. If you want to call an ILE RPG procedure in a multi-threaded application, you must ensure that only one thread is calling any ILE RPG procedure at a time. If you want to try running an ILE RPG procedure in a multi-threaded environment, even though it is not supported, and even though you ensure that only one thread is calling any ILE RPG procedure at one time, you should be aware that some functions that ILE RPG uses are not thread-safe.

Chapter 3. Program Creation Strategies

There are many approaches you can take in creating programs using an ILE language. This section presents three common strategies for creating ILE programs using ILE RPG 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 DFACTGRP(*YES).

Specifying DFACTGRP(*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 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.

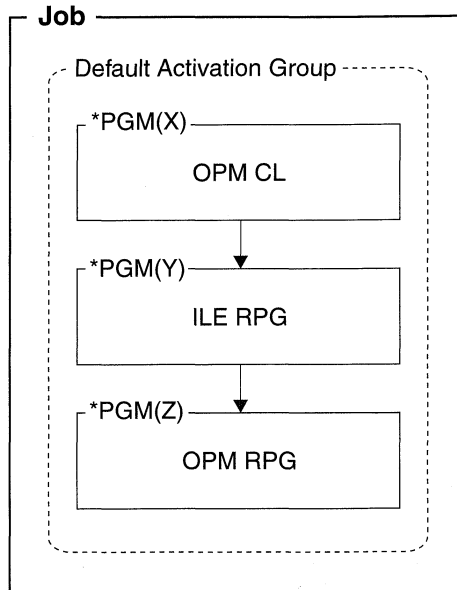


Figure 6. OPM-Compatible Application

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 382
One-step creation process	Chapter 6, “Creating a Program with the CRTBNDRPG Command” on page 57
ILE static binding	Chapter 10, “Calling Programs and Procedures” on page 129; also <i>ILE Concepts</i>
Exception handling differences	“Differences between OPM and ILE RPG Exception Handling” on page 222

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 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.

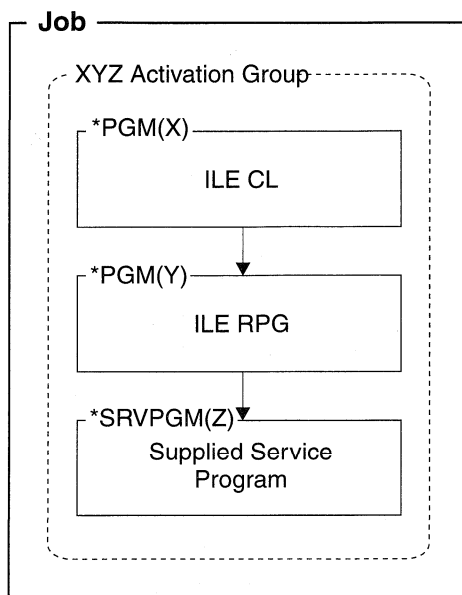


Figure 7. ILE Program Using CRTBNDRPG

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 382
One-step creation process	Chapter 6, “Creating a Program with the CRTBNDRPG Command” on page 57
Activation groups	“Managing Activation Groups” on page 109
RCLRSC	“Reclaim Resources Command” on page 112
ILE static binding	Chapter 10, “Calling Programs and Procedures” on page 129; also <i>ILE Concepts</i>
Exception handling differences	“Differences between OPM and ILE RPG Exception Handling” on page 222
Override and open scope	“Overriding and Redirecting File Input and Output” on page 273 and “Sharing an Open Data Path” on page 277; also <i>ILE Concepts</i>

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 26.

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.

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 29 shows the run-time view of this application.

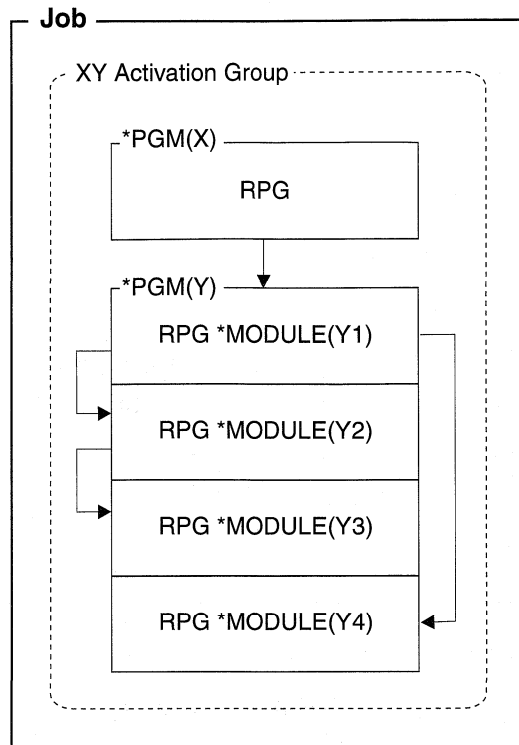


Figure 8. Single-Language Application Using CRTRPGMOD and CRTPGM

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 26 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 30 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).

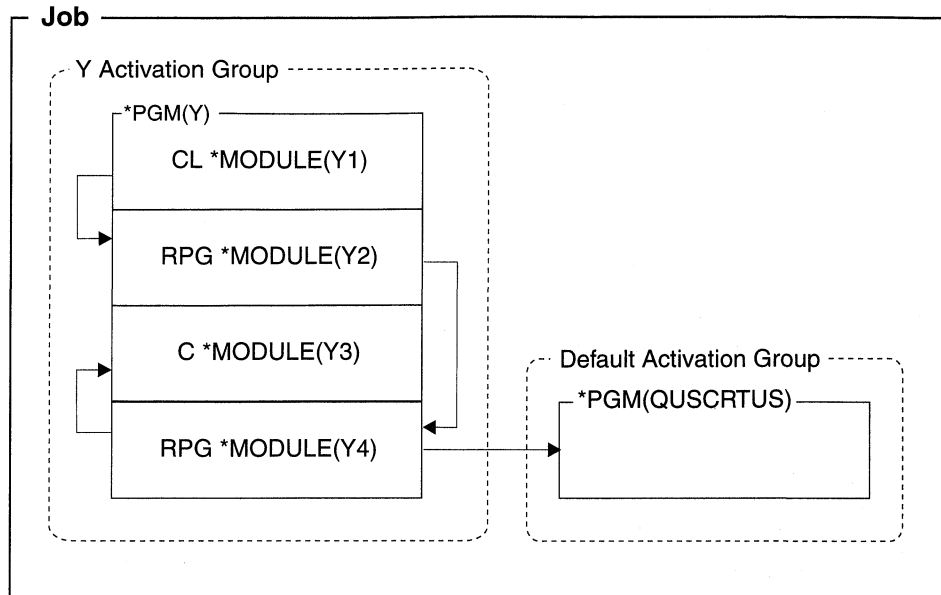


Figure 9. Mixed-Language Application

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 26 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 31 shows an example in which an ILE program is bound to two service programs.

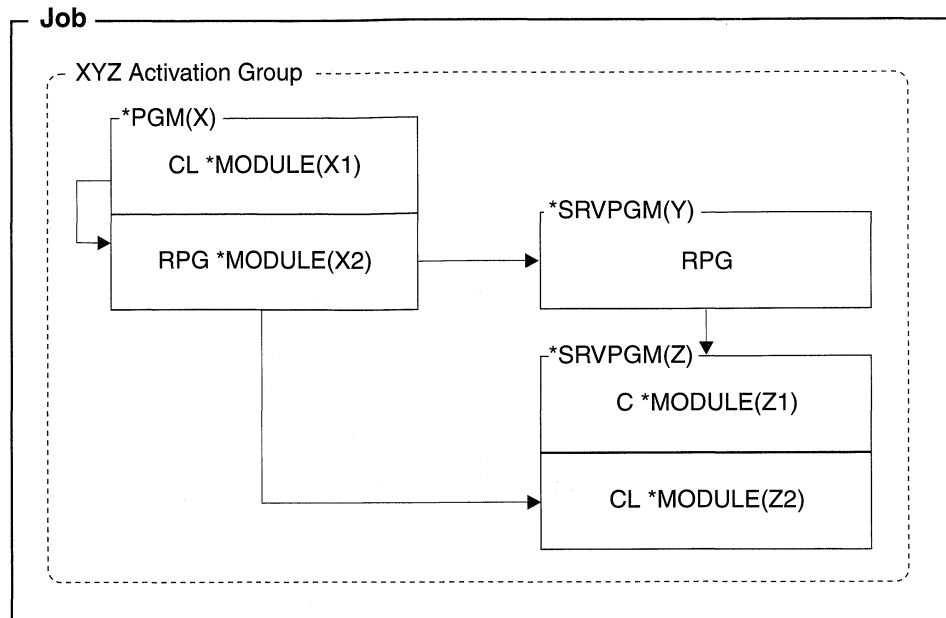


Figure 10. Advanced Application

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

See “Effect of ILE” on page 26 for details on the effects of ILE on the way your application handles calls, data, files and errors.

Related Information

Two-step creation process

Chapter 7, “Creating a Program with the CRTRPGMOD and CRTPGM Commands” on page 73

Activation groups

“Managing Activation Groups” on page 109

ILE static binding

Chapter 10, “Calling Programs and Procedures” on page 129; also *ILE Concepts*

Exception Handling

Chapter 12, “Handling Exceptions” on page 217; also *ILE Concepts*

Service programs

Chapter 8, “Creating a Service Program” on page 91; also *ILE Concepts*

Updating a Program

“Using the UPDPGM Command” on page 87

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 32.

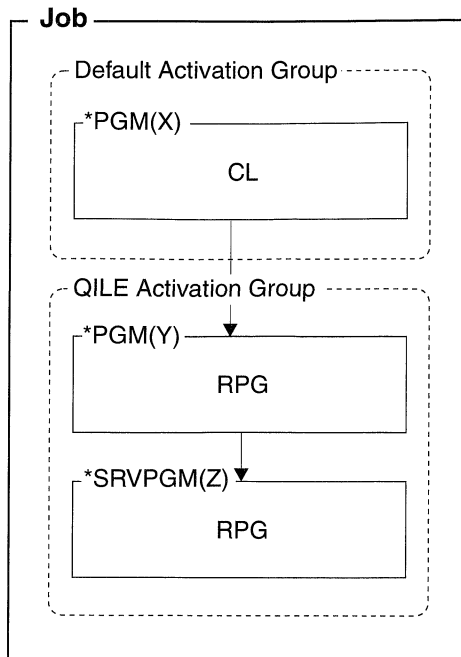


Figure 11. Scenario to Avoid. An application is split between the OPM default activation group and a named activation group.

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

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

Chapter 4. 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. A **procedure** is any piece of code that can be called with a bound call. ILE RPG has two kinds of procedures: a main procedure and a subprocedure. The way to call a subprocedure is with a prototyped 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 a main procedure and zero or more subprocedures. (If there are subprocedures, the main procedure is optional.) 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). The main procedure is defined in the **main source section**, which is the set of H, F, D, I, C, and O specifications that begin a module. In V3R1, all ILE RPG modules had a main procedure and no other procedures.

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, D, and C specifications can be used.

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 34 shows what a module might look like with multiple procedures.

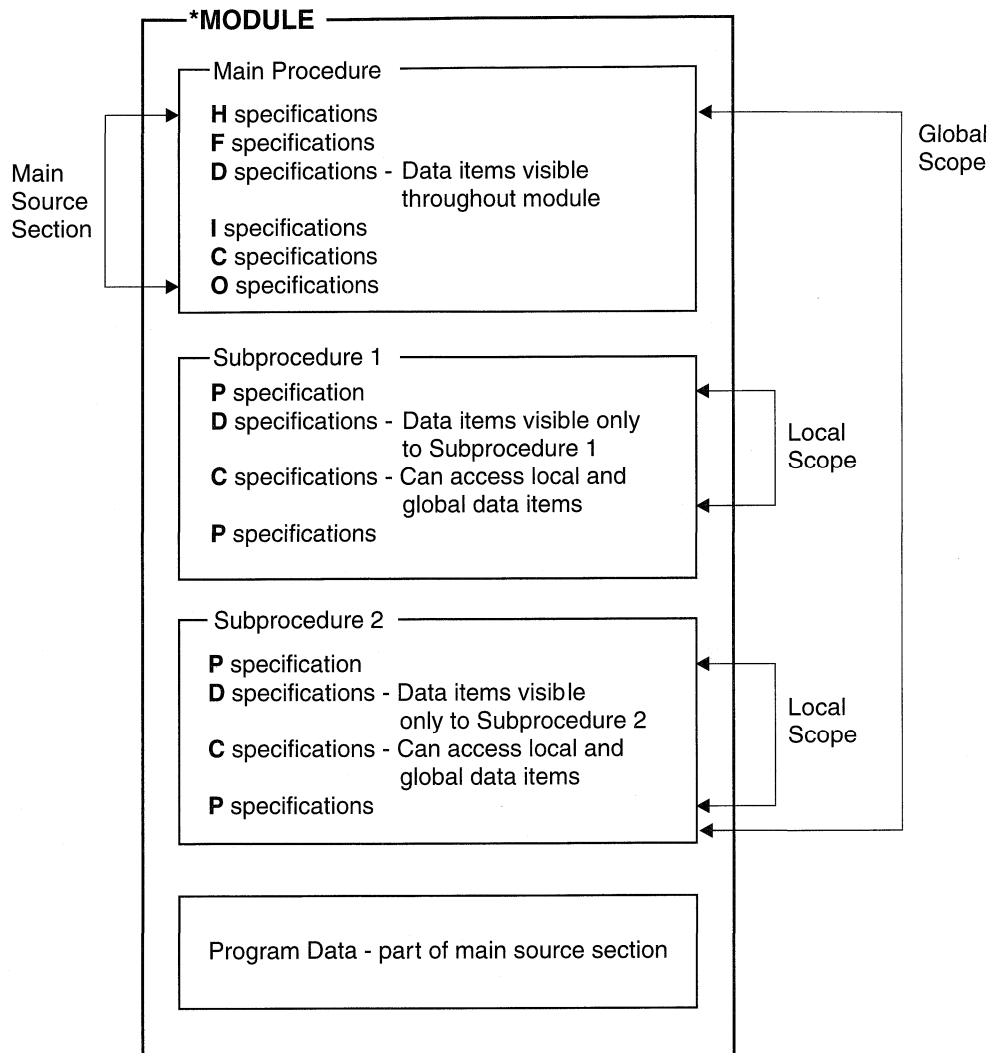


Figure 12. An ILE RPG module with Multiple Procedures

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 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 produce the formatted output fields, `FmtCust` calls a procedure `NumToChar`. `NumToChar` has a numeric input parameter that is passed by value, and returns a character field. Figure 14 shows the prototype for `NumToChar`.

```

* Prototype for procedure NumToChar
* The returned value is a character field of length 31.
D NumToChar       PR          31A
* The input parameter is packed with 30 digits and 0 decimal
* positions, passed by value.
D NUMPARM         30P 0  VALUE

```

Figure 14. Prototype for `NumToChar` 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 shows a call to `FmtCust`. Note that the names of the output parameters, shown above in Figure 13, 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

In order to format the name and address properly, FmtCust calls NumToChar to convert the customer number to a character field. Because FmtCust wants to use the return value, the call to NumToChar is made in an expression. Figure 16 on page 36 shows the call.

```

*-----
* CUSTNAME and CUSTNUM are formatted to look like this:
*   A&P Electronics      (Customer number 157)
*-----
C           EVAL      Name = CUSTNAME + ' '
C           + '(Customer number '
C           + %trimr(NumToChr(CUSTNUM)) + ' )'
```

Figure 16. Calling the NumToChar 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 opens up a number of new 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 37.

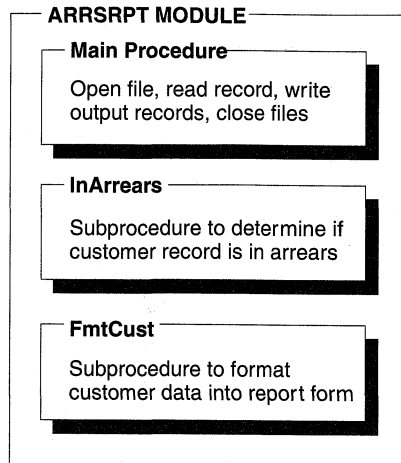


Figure 17. Components of the ARRSRPT Module

Now consider the first subprocedure, `InArrears`, which is shown in Figure 18 on page 38. `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          2
* Local declarations
D DaysLate       S          3
D DateDue        S          3

* Body of procedure
C  *ISO          MOVE      DUEDATE      DateDue
C  CurDate       SUBDUR    DateDue      DaysLate:*D

C              IF        DaysLate > 60 AND
C              AMOUNT > 100.00
C              RETURN    '1'          4
C              ELSE
C              RETURN    '0'          4 5
C              ENDIF

P InArrears      E          1

```

Figure 18. Source for Subprocedure InArrears

Figure 18 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 main procedure with prototyped entry parameters, you need to define a procedure interface. A **procedure interface definition** is a repeat of the prototype information 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). In the case of InArrears, there are no entry parameters.

Consider next the subprocedure FmtCust, which is shown in Figure 19 on page 39. 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.

One of the formatting tasks requires converting a numeric field to a character field to match the output field type. This conversion could be coded as part of the subprocedure itself. However, as this is a task that might be required by other reports, we decided to code the conversion as a separate subprocedure, NumToChar. NumToChar takes as input a numeric parameter that is passed by value. It converts the number to a character field and returns that field to the caller, in this case FmtCust.

```

*-----*
* FmtCust formats CUSTNAME, CUSTNUM, STREETNAME etc into
* readable forms
*
* Parameters:  Name      (output)
*              Address   (output)
* Globals:    CUSTNAME, CUSTNUM, STREETNUM, STREETNAME, CITY
*              STATE, ZIP
* Returns:    (none)
*-----*
P FmtCust      B
D FmtCust      PI
D Name          100A
D Address       100A

D ZipChar      S      5A

*-----*
* CUSTNAME and CUSTNUM are formatted to look like this:
*   A&P Electronics      (Customer number 157)
*-----*
C              EVAL      Name = CUSTNAME + ' '
C              + '(Customer number '
C              + %trimr(NumToChar(CUSTNUM)) + '')

*-----*
* STREETNUM, STREETNAME, CITY, STATE, and ZIP are formatted to look like:
*   27 Garbanzo Avenue, Smallville IN 51423
*-----*
C              MOVEL      ZIP      ZipChar
C              EVAL      Address = %trimr(NumToChar(STREETNUM))
C              + ' ' + %trimr(STREETNAME) + ', '
C              + %trim(CITY) + ' ' + %trim(STATE)
C              + ' ' + ZipChar

P FmtCust      E

```

Figure 19. Source for Subprocedure FmtCust

Note that NumToChar is a prototyped procedure, and so you might expect to see its prototype inside of FmtCust. You could place the prototype in FmtCust. However, we placed it in the main source section, so that it would be available to any subprocedure we might add to ARRSRPT. This is shown later on.

Finally, consider the last subprocedure of this application, NumToChar. Notice that NumToChar does not appear in the ARRSRPT module, that is shown in Figure 17 on page 37. We decided to place NumToChar inside another module called

CVTPROCS. CVTPROCS is a utility module that will contain any conversion procedures that other modules might need to use.

Figure 20 shows the source of the module CVTPROCS. Since this is a prototyped procedure, it needs the prototype to be available. So that the prototype can be shared, we have placed the prototype into a /COPY file.

```

=====
* Source for module CVTPROCS. 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 NumToChar.
-----
/COPY QRPGLSRC,CVTPROCP
-----
* NumToChar converts a numeric field to a character field
*
* Parameters:  NUMPARM  (input)
* Globals:    (none)
* Returns:    character string
*
* The subprocedure will be called by procedures outside of this
* module, and so the keyword EXPORT is required to indicate this.
-----
P NumToChar      B              EXPORT
D NumToChar      PI              31A
D  NUMPARM       30P 0  VALUE
* Local declarations
D POS           S              5P 0
D SIGN          S              1A  INZ(' ')
D ZONEDS        DS
D  NUM          30S 0
D  CHAR         30A  OVERLAY(NUM)
*
* Body of procedure:
* Handle special case of zero
C              IF      NUMPARM = 0
C              RETURN  '0'
C              ENDIF
* Handle negatives. Set sign to '-' and make positive.
C              IF      NUMPARM < 0
C              EVAL   NUMPARM = - NUMPARM
C              EVAL   SIGN = '-'
C              ENDIF
* Get the input number in character form using a data structure
C              EVAL   NUM = NUMPARM
* Left-justify
C
C  '0'          CHECK   CHAR          POS
C              EVAL   CHAR = %SUBST(CHAR:POS)
* Return the value adding the sign
C              RETURN  %TRIML(SIGN + CHAR)

P NumToChar      E

```

Figure 20. Source for module CVTPROCS, containing subprocedure NumToChar

CVTPROCS 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 46.

The Entire ARRSRPT Program

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

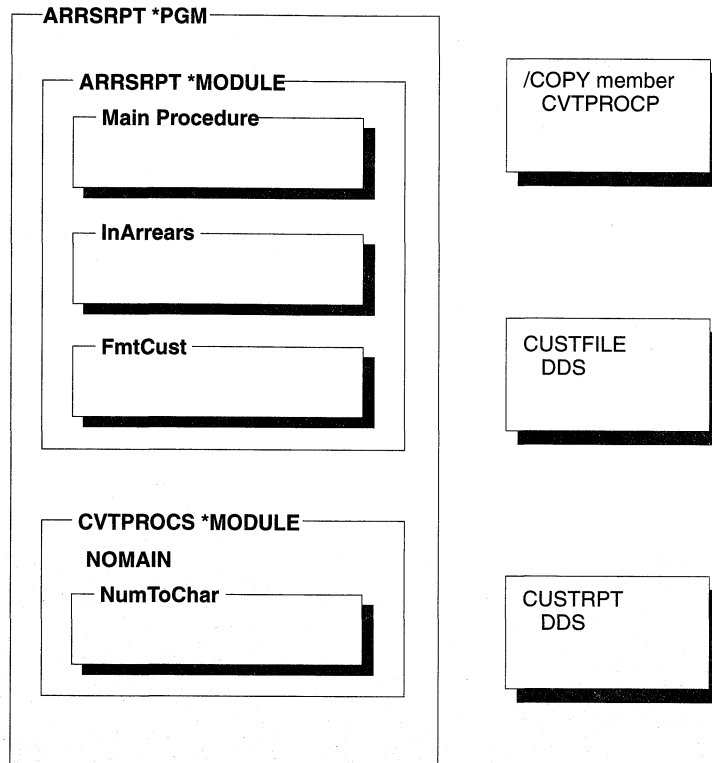


Figure 21. The ARRSRPT Application

Figure 22 shows the source for the entire ARRSRPT module.

```

=====
* Source for module ARRSRPT. Contains a main procedure and
* two subprocedures: InArrears and FmtCust.
*
* Related Module: CVTPROCS (NumToChar called by FmtCust)
=====

-----
* F I L E S
*
* CUSTFILE - contains customer information
* CUSTRPT - printer file (using format ARREARS)
-----

FCUSTFILE IP E DISK
FCUSTRPT 0 E PRINTER
  
```

Figure 22 (Part 1 of 3). ILE RPG Complete Source for ARRSRPT Module

```

*-----*
* P R O T O T Y P E S
*-----*
/COPY QRPGL, CVTPROCP

*-----*
* 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      *MDY        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

```

Figure 22 (Part 2 of 3). ILE RPG Complete Source for ARRSRPT Module

```

* Body of procedure
C *ISO MOVE DUEDATE DateDue
C CurDate SUBDUR DateDue DaysLate:*D

C IF DaysLate > 60 AND
C AMOUNT > 100.00
C RETURN '1'
C ELSE
C RETURN '0'
C ENDIF

P InArrears E

*-----
* FmtCust formats CUSTNAME, CUSTNUM, STREETNAME etc into
* readable forms
*
* Parameters: Name (output)
* Address (output)
* Globals: CUSTNAME, CUSTNUM, STREETNUM, STREETNAME, CITY
STATE, ZIP
* Returns: (none)
*-----
P FmtCust B
D FmtCust PI
D Name 100A
D Address 100A

D ZipChar S 5A

*-----
* CUSTNAME and CUSTNUM are formatted to look like this:
* A&P Electronics (Customer number 157)
*-----
C EVAL Name = CUSTNAME + ' '
C + '(Customer number '
C + %trimr(NumToChar(CUSTNUM)) + ' )'

*-----
* StreetNum, STREETNAME, CITY, STATE, and ZIP are formatted to look like:
* 27 Garbanzo Avenue, Smallville 51423
*-----
C MOVEL ZIP ZipChar
C EVAL Address = %trimr(NumToChar(STREETNUM))
C + ' ' + %trimr(STREETNAME) + ', '
C + %trim(CITY) + ' ' + %trim(STATE)
C + ' ' + ZipChar

P FmtCust E

```

Figure 22 (Part 3 of 3). ILE RPG Complete Source for ARRSRPT Module

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 NumToChar.

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.

```

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 and Figure 25 on page 45 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      AMOUNT       10 2     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
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

- Because the 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 main procedure is always exported, which means that other procedures in the program can call the main procedure by using bound calls.
- The call interface of a main procedure can be defined in one of two ways:
 1. Using a prototype and procedure interface
 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.
- You cannot define return values for a main procedure, nor can you specify that its parameters be passed by value.

Subprocedure Considerations

- Any of the calculation operations may be coded in a subprocedure. However, all files must be defined globally, so all input and output specifications must be defined in the main source section. Similarly, all data areas must be defined in the main procedure, 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 differs somewhat from that of a main procedure, because there is no cycle code for the subprocedure.
 - When a subprocedure ends, it simply returns to the caller. None of the usual termination activities, such as closing of files, occurs until the main procedure that is associated with the subprocedure itself ends. You should 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 is to code the NOMAIN 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<temp obj> by a DLT<temp obj>, and so on. This alternative applies to modules that may have a subprocedure active when the main procedure is not active.

- Exception handling within a subprocedure differs from a main procedure primarily because there is no default exception handler for subprocedures. As a result, situations where the default handler would be called for a main procedure correspond to 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 Main Procedure" on page 220
Main Procedure End	"Returning from a Main Procedure" on page 153

Subprocedures

Topic	See
Defining	Chapter on subprocedures, in the <i>ILE RPG for AS/400 Reference</i>
NOMAIN module	"Creating a NOMAIN Module" on page 75
Exception handling	"Exception Handling within Subprocedures" on page 221
Procedure Specification	Chapter on procedure specifications, in the <i>ILE RPG for AS/400 Reference</i>
Procedure Interface	Chapter on defining data and prototypes in the <i>ILE RPG for AS/400 Reference</i>
Subprocedure End	"Returning from a Subprocedure" on page 156

Prototyped Call

Topic	See
Free-form call	"Using a Prototyped Call" on page 135
General Information	<i>ILE RPG for AS/400 Reference</i> , Chapter 24
Passing parameters	"Passing Prototyped Parameters" on page 137
Prototypes	Chapter on defining data and prototypes in the <i>ILE RPG for AS/400 Reference</i>

Creating and Running an ILE RPG Application

This section provides you with the information 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 ILE terms and concepts are discussed briefly in the following pages. These terms and concepts are more fully discussed in *ILE Concepts*.

Chapter 5. Entering Source Statements

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

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

- Interactively using SEU
- Interactively using CODE/400

Initially, you may want to enter your source statements into a file called QRPGLSRC. New members of the file QRPGLSRC 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 QRPGLSRC. IBM supplies a source file QRPGLSRC 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.

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.

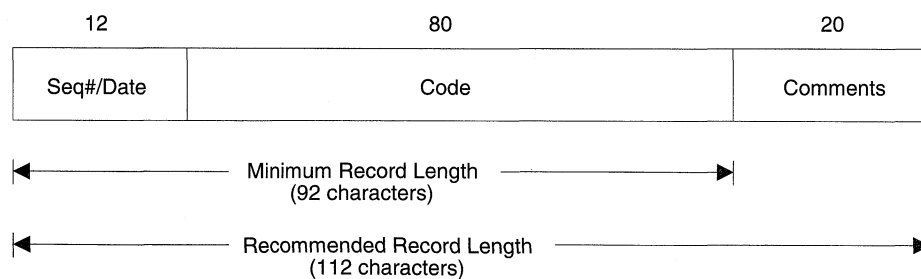


Figure 26. ILE RPG Record Length Breakdown

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/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/400: Source Entry Utility* manual.

If you name your source file QRPGLSRC, 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 QRPGLSRC type:

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

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

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

```
STRSEU SRCFILE(MYLIB/QRPGLSRC)  
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 QRPGLSRC in library MYLIB and starts an edit session.

The SEU Edit display appears as shown in Figure 27 on page 53. 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)
*                  QSYSVRT (PRINTER FILE)
* DESCRIPTION:      This program prints employee information
*                  from the file EMPMST.
*=====*
FQSYSVRT  0  F  80      PRINTER
FEMPMST   IP E          K DISK

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

IEMPREC        01

C          EVAL      TYPE = EMPTYTYPE(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 EMPTYTYPE returns a string representing the employee
* type indicated by the parameter CODE.
P EMPTYTYPE      B
D EMPTYTYPE      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 EMPTYTYPE      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 55 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 for AS/400® 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
- Start your SQL statements using the delimiter /EXEC SQL in positions 7-15 (with the / in position 7)
- You can start entering your SQL statements on the same line as the starting delimiter
- Use the continuation line delimiter (a + in position 7) to continue your statements on any subsequent lines
- Use the ending delimiter /END-EXEC in positions 7-15 (with the slash in position 7) to signal the end of your SQL statements.

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

Figure 30 on page 56 shows an example of embedded SQL statements.

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

Figure 30. SQL Statements in an ILE RPG Program

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

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

Chapter 6. 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 DFACTGRP 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 Chapter 7, “Creating a Program with the CRTRPGMOD and CRTPGM Commands” on page 73.

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

Table 3 summarizes the parameters of the CRTBNDRPG command and shows their default values.

Program Identification	
PGM(*CURLIB/*CTLSPEC)	Determines created program name and library
SRCFILE(*LIBL/QRPGLESRC)	Identifies source file and library
SRCMBR(*PGM)	Identifies file member containing source specifications
TEXT(*SRCMBRTXT)	Provides brief description of program
Program Creation	
GENLVL(10)	Conditions program creation to error severity (0-20)
OPTION(*GEN)	*GEN/*NOGEN, determines if program is created
DBGVIEW(*STMT)	Specifies type of debug view, if any, to be included in 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
Compiler Listing	
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 *NOSHOWSKP *EXPDDS *EXT)	Specifies the contents of compiler listing
Data Conversion Options	
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
Run-Time Considerations	
DFACTGRP(*YES)	Identifies whether this program always runs in the OPM default activation group
ACTGRP(QILE)	Identifies the activation group in which the program should run
SRTSEQ(*HEX)	Specifies the sort sequence table to be used
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.

See Appendix C, “The Create Commands” on page 405 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 54.

1. To create the object type:

```
CRTBNDRPG PGM(MYLIB/EMPRPT) DBGVIEW(*SOURCE) DFACTGRP(*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 31 on page 60 shows the screen which appears after entering the above command.

```

                                Display Module Source
Program:  EMPRPT           Library:  MYLIB           Module:  EMPRPT
1  *=====*
2  * MODULE NAME:      EMPRPT
3  * RELATED FILES:   EMPMST  (PHYSICAL FILE)
4  *                   QSYSPRT (PRINTER FILE)
5  * DESCRIPTION:    This program prints employee information
6  *                   from the file EMPMST.
7  *=====*
8  FQSYSPRT  O  F  80      PRINTER
9  FEMPMST   IP  E           K DISK
10
11 D TYPE           S           8A
12 D EMPTYTYPE     PR          8A
13 D CODE          1A
14
15 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 31. Display Module Source display for EMPRPT

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

For more information on debugging see Chapter 11, “Debugging Programs” on page 163.

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

For more information on service programs, see Chapter 8, “Creating a Service Program” on page 91.

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 32 on page 62.

1. To create the object, type:

```
CRTBNDRPG PGM(MYLIB/PAYROLL)
          SRCFILE(MYLIB/QRPGLESRC)
          TEXT('ILE RPG program') DFACTGRP(*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 DFACTGRP(*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
FQSYSVRT  0  F  80      PRINTER

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

*-----*
* Constant Declarations *
*-----*
D Heading1     C          'NUMBER NAME          RATE H-
D              C          OURS  BONUS  PAY      '
D Heading2     C          ' _____ , _____ -
D              C          _____ , _____ -

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

```

Figure 32 (Part 1 of 2). A Sample Payroll Calculation Program


```

*-----*
* Report Layout
* -- print the heading lines if 1P is on
* -- if the record is found (indicator 99 is off) print the payroll
* details otherwise print an exception record
* -- print 'END OF LISTING' when LR is on
*-----*
00QSYSVRT H 1P 2 3
0 35 'PAYROLL REGISTER'
0 *DATE Y 60
0 H 1P 2 60 Heading1
0 H 1P 2 60 Heading2
0 D N1PN99 2
0 TRN_NUMBER 5
0 EMP_NAME 24
0 EMP_RATE L 33
0 TRN_HOURS L 40
0 TRN_BONUS L 49
0 Pay 60 '$ 0. '
0 D N1P 99 2
0 TRN_NUMBER 5
0 35 '** NOT ON EMPLOYEE FILE **'
0 T LR
0 33 'END OF LISTING'

```

Figure 32 (Part 2 of 2). 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 423 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.

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 *ILE RPG for AS/400 Reference*.

Indenting Structured Operations

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

Line Number	Source Specifications	Comments	Src Seq
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 33. 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 34 shows an example of finger in-line diagnostic messages.

Line Number	Source Specifications	Comments	Do Num	Page Line	Change Date	Src Id	Seq Number
63	C	SETOFF		12			003100
=====>				aabb			
=====>				ccccc			
*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 34. 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 'ccccc' 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 'ccccc' 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 35 on page 68 shows an example of the non-finger in-line diagnostic messages.

Line Number	<----- Source Specifications ----->	Comments	Do	Page	Change	Src	Seq
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 35. 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 the listing statement number and SEU sequence 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/400: Source Entry Utility*.

Correcting Run-time Errors

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

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 431.

Coordinating Listing Options with Debug View Options

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

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

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 74) commands create and update a data area with the status of the last compilation. This data area is named RETURNCODE, is 400 characters long, and is placed into library QTEMP.

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

The data area RETURNCODE has the following format:

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

181-329 Not set. Always blank
330-334 Total elapsed compile time to the nearest 10th of a second
335 Not set. Always blank
336-340 Elapsed compile time to the nearest 10th of a second
341-345 Elapsed translator time to the nearest 10th of a second
346-379 Not set. Always blank
380-384 Total compile CPU time to the nearest 10th of a second
385 Not set. Always blank
386-390 CPU time used by compiler to the nearest 10th of a second
391-395 CPU time used by the translator to the nearest 10th of a second
396-400 Not set. Always blank

Chapter 7. Creating a Program with the CRTRPGMOD and CRTPGM Commands

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

This chapter shows how to:

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

Creating a Module Object

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

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 **main procedure** which consists of the set of H, F, D, I, C, and O specifications that begin the source. The 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.

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).

Table 4 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 405.

Table 4 (Page 1 of 2). CRTRPGMOD Parameters and Their Default Values Grouped by Function

Module Identification	
MODULE(*CURLIB/*CTLSPEC)	Determines created module name and library
SRCFILE(*LIBL/QRPGLESRC)	Identifies source file and library
SRCMBR(*MODULE)	Identifies file member containing source specifications
TEXT(*SRCMBRTXT)	Provides brief description of module
Module Creation	
GENLVL(10)	Conditions module creation to error severity (0-20)
OPTION(*GEN)	*GEN/*NOGEN, determines if module is created
DBGVIEW(*STMT)	Specifies type of debug view, if any, to be included in 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
ENBPFRCOL(*PEP)	Specifies whether performance collection is enabled
DEFINE(*NONE)	Specifies condition names that are defined before the compilation begins

<i>Table 4 (Page 2 of 2). CRTRPGMOD Parameters and Their Default Values Grouped by Function</i>	
PRFDTA(*NOCOL)	Specifies the program profiling data attribute
Compiler Listing	
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 *SHOWSKP *EXPDDS *EXT)	Specifies the contents of compiler listing
Data Conversion Options	
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
Run-Time Considerations	
SRTSEQ(*HEX)	Specifies the sort sequence table to be used
LANGID(*JOB RUN)	Used with SRTSEQ to specify the language identifier for sort sequence
TRUNCNBR(*YES)	Specifies action to take when numeric overflow occurs for packed-decimal, zoned-decimal, and binary fields in fixed format operations.

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 63. A sample compiler listing is provided in Appendix D, “Compiler Listings” on page 423.

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 shown in Figure 36 on page 76. The prototypes for the procedures in TRANSSVC are stored in a /COPY member, as shown in Figure 37 on page 78.

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

```

Figure 36 (Part 1 of 3). Source for TRANSSVC member

```

*-----*
* 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

```

Figure 36 (Part 2 of 3). Source for TRANSVC member

```

*-----*
* 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 36 (Part 3 of 3). Source for TRANSVC 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 37. 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 38 on page 79.

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 166.

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 transaction 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  0  F  80      PRINTER      OFLIND(*INOF)

/COPY QRPGL,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)

QSYSPRT   H  1P          1
0         OR  OF
0
0         12 'Product name'
0         40 'Quantity'
0         54 'Income'
QSYSPRT   H  1P          1
0         OR  OF
0         30 '-----+'
0         -----+
0         -----'
0         40 '-----'
0         60 '-----'
QSYSPRT   D  01          1
0         ProdName      30
0         QTY           1  40
0         Income        1  60
QSYSPRT   T  LR          1
0
0         Total         1         'Total: '

```

Figure 38. Source for TRANSRPT module

The DDS for the file TRNSDTA is shown in Figure 39 on page 80. The /COPY member is shown in Figure 37 on page 78.

```

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 39. DDS for TRNSDTA

Additional Examples

For additional examples of creating modules, see:

- “Sample Service Program” on page 94, for an example of creating a module for a service program.
- “Binding to a Program” on page 98. for an example of creating a module to be used with a service program.
- “Managing Your Own Heap Using ILE Bindable APIs” on page 120, for an example of creating a module for dynamically allocating storage for a run-time array
- “Sample Source for Debug Examples” on page 211, 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 *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 main procedure. The cycles are independent: setting on LR in one main procedure has no effect on the cycle in another.

Related CL Commands

The following CL commands can be used with modules:

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

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

Binding Modules into a Program

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

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

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 40 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.

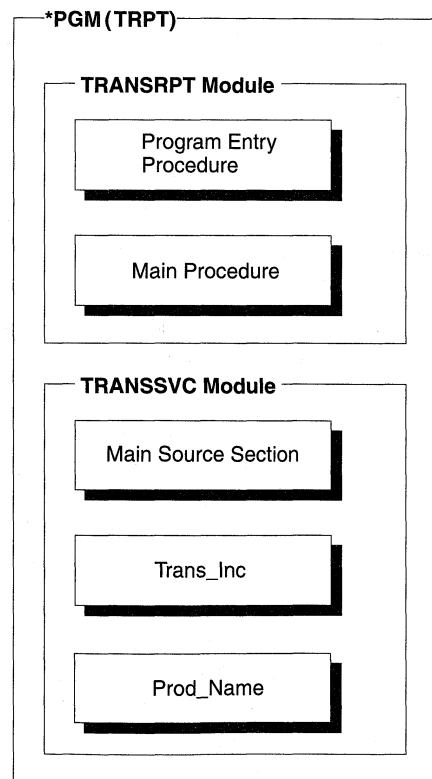


Figure 40. Structure of Program TRPT

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

In addition to binding modules together, you can also bind them to service programs (type *SRVPGM). Service programs allow you to code and maintain modules separately from the program modules. Common routines can be created as service programs and if the routine changes, the change can be incorporated by binding the service program again. The programs that use these common routines do not have to be recreated. For information on creating service programs see Chapter 8, “Creating a Service Program” on page 91.

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, CRTCLMOD, 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 110.

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

Parameter Group	Parameter(Default Value)
Identification	PGM(<i>library name/program name</i>) MODULE(*PGM)
Program access	ENTMOD(*FIRST)
Binding	BNSRVPGM(*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 38 on page 79, Figure 36 on page 76 and Figure 39 on page 80 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 41 on page 85 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 41. File QSYSPRT for TRPT

Additional Examples

For additional examples of creating programs, see:

- “Binding to a Program” on page 98, for an example of binding a module and a service program.
- “Sample Source for Debug Examples” on page 211, for an example of creating a program consisting of an RPG and C module.

Related CL Commands

The following CL commands can be used with programs:

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

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

Using a Binder Listing

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

- `*BASIC`
- `*EXTENDED`
- `*FULL`

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

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 100.

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* and the *CL Reference*.

Changing the Optimization Level

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

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

To ensure that the contents of a field reflect their most current value, especially after exception recovery, you can use the NOOPT keyword on the corresponding Definition specification. For more information, see "Optimization Considerations" on page 226.

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 of 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 Reference*.

Chapter 8. Creating a Service Program

This chapter provides:

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

Service Program Overview

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

Service programs are typically used for common functions that are frequently called within an application and across applications. For example, the ILE compilers use service programs to provide 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 94 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 7 on page 93 lists the CRTSRVPGM parameters and their defaults. For a full description of the CRTSRVPGM command and its parameters, refer to the *CL Reference*.

<i>Table 7. Parameters for CRTSRVPGM Command and their Default Values</i>	
Parameter Group	Parameter(Default Value)
Identification	SRVPGM(<i>library name/service program name</i>) 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 97 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 86.

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 42 on page 95 shows the source for the service program. Figure 43 on page 97 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 140.


```

*****
* 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
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 42 (Part 1 of 2). 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

```

Figure 42 (Part 2 of 2). Source for Service Program CvtToHex

```

=====
* 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 43. 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. 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. Figure 44

```

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

```

Figure 44. 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 42 on page 95, 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 44.

```

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 45 on page 99, 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
OQSYSPRT  H   1P
O
OQSYSPRT  E
O              ResultDS
O              Comment          +5

```

Figure 45. 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 46 on page 101 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 85 and also *ILE Concepts*.

```

5769SS1 V4R2M0 980228                                Create Program                                Page 1
MYLIB/CVTHEXPGM AS400S01 02/30/98 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
Program Library Program Library Program Library
CVTTOHEX MYLIB
Binding Binding
Directory Library Directory Library Directory Library
*NONE

5769SS1 V4R2M0 980228                                Create Program                                Page 2
MYLIB/CVTHEXPGM AS400S01 01/25/98 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

***** END OF BRIEF SUMMARY TABLE *****

5769SS1 V4R2M0 980228                                Create Program                                Page 3
MYLIB/CVTHEXPGM AS400S01 02/30/98 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

***** END OF BINDING STATISTICS *****
*CPC5D07 - Program CVTHEXPGM created in library MYLIB.
***** END OF CREATE PROGRAM LISTING *****

```

Figure 46. Basic Binder listing for CVTHEXPGM

Chapter 9. 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 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. Chapter 10, "Calling Programs and Procedures" on page 129 provides information on running programs from another HLL or calling service programs and procedures.,

Running a Program Using the CL CALL Command

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

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

```
CALL EMPRPT
```

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

```
CALL MYLIB/EMPRPT
```

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

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

Passing Parameters using the CL CALL Command

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

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

You can also type the parameters without specifying any keywords:

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

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

- A character string constant
- A numeric constant
- A logical constant

If you are passing parameters to a program where an ILE RPG 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 *CL Reference* or to 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 47 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  0  F  80      PRINTER
FEMPMST  IP  E        K DISK

IEMPREC      01
```

Figure 47 (Part 1 of 2). ILE RPG Program that Requires Parameters at Run Time

```

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

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

C   PSWORD      IFNE      'HELLO'
C   SETON
C   ENDIF

OPRINT  H  1P                2  6
0
0   H  1P                50 'EMPLOYEE INFORMATION'
0
0                12 'NAME'
0                34 'SERIAL #'
0                45 'DEPT'
0                56 'TYPE'
0   D  01N99      ENAME      20
0                ENUM      32
0                EDEPT     45
0                ETYPE     55
0   D  99                16 '***'
0                40 'Invalid Password Entered'
0                43 '***'
0

```

Figure 47 (Part 2 of 2). ILE RPG Program that Requires Parameters at Run Time

Figure 48 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 48. 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 49 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 49. Example of an Application Menu

The menu shown in Figure 49 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 50 on page 107 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          CHGINPDT
A          INDARA
A          PRINT(*LIBL/QSYSPT)
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 50. Data Description Specification of an Application Menu

Figure 51 shows the source of the application menu illustrated in Figure 49 on page 106. 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 51. 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 51 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 there is no error indicator or error subroutine (*PSSR or INFSR) to handle the exception in a main procedure. The inquiry messages require a response before the program continues running.

Note: Inquiry messages are never issued for subprocedures.

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

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

RNQ0100	RNQ0222	RNQ0431	RNQ1031	RNQ1241
RNQ0101	RNQ0231	RNQ0432	RNQ1041	RNQ1251
RNQ0102	RNQ0232	RNQ0450	RNQ1042	RNQ1255
RNQ0103	RNQ0299	RNQ0501	RNQ1051	RNQ1261
RNQ0104	RNQ0333	RNQ0502	RNQ1071	RNQ1271
RNQ0112	RNQ0401	RNQ0802	RNQ1201	RNQ1281
RNQ0113	RNQ0402	RNQ0803	RNQ1211	RNQ1282
RNQ0114	RNQ0411	RNQ0804	RNQ1215	RNQ1284
RNQ0115	RNQ0412	RNQ0805	RNQ1216	RNQ1285
RNQ0120	RNQ0413	RNQ0907	RNQ1217	RNQ1286
RNQ0121	RNQ0414	RNQ1011	RNQ1218	RNQ1287
RNQ0122	RNQ0415	RNQ1021	RNQ1221	RNQ1299
RNQ0123	RNQ0421	RNQ1022	RNQ1222	RNQ1331
RNQ0202	RNQ0425	RNQ1023	RNQ1231	RNQ9998
RNQ0211	RNQ0426	RNQ1024	RNQ1235	RNQ9999
RNQ0221				

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 *SYSRPLY 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(*SYSRPLY). The INQMSGRPY parameter occurs on the following CL commands:

- Change Job (CHGJOB)
- Change Job Description (CHGJOB)
- Create Job Description (CRTJOB)
- Submit Job (SBMJOB).

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

Ending an ILE Program

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

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

Error message-id caused program to end.

is issued and control is returned to the caller.

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

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

For more information on exception messages, see “Exception Handling Overview” on page 217.

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

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

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

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

Maintaining OPM RPG/400 and ILE RPG 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 382 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 23.

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

Reclaim Resources Command

The Reclaim Resources (RCLRSC) command is designed to free the resources for programs 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 (returning with LR off), 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 Reference*. For more information on the RCLRSC and activation groups, refer to *ILE Concepts*.

Managing Dynamically-Allocated Storage

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

You manage heaps by using the storage management operations ALLOC, REALLOC, and DEALLOC or by using 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.

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 been 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 allocates storage within the default heap.
- The DEALLOC operation frees one previous allocation of heap storage from any heap.
- The REALLOC operation changes the size of previously allocated storage from any heap.

Note: Although the ALLOC operation code works only with the default heap, the REALLOC and DEALLOC operation codes work with both the default heap and user-created heaps.

Figure 52 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 52 (Part 1 of 5). Memory Management - Build a Linked List of Names

```

*-----*
* SUBPROCEDURES *
*-----*

*-----*
* 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@

```

Figure 52 (Part 2 of 5). Memory Management - Build a Linked List of Names


```

*-----*
* 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&gm1.name_len)
C    'Name: '      dsply      dspName
C                  ENDDO

*-----*
* Restore the list pointer to its former place
*-----*
C                  EVAL      elem@ = saveElem@
P Display          E

```

Figure 52 (Part 4 of 5). Memory Management - Build a Linked List of Names

```

*-----*
* 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

```

Figure 52 (Part 5 of 5). Memory Management - Build a Linked List of Names

Heap Storage Problems

Figure 53 on page 119 shows possible problems associated with the misuse of heap storage.


```

*-----*
* Heap Storage Misuse
*-----*
D F1d1      S          25A  BASED(Ptr1)
D F1d2      S          5A   BASED(Ptr2)
D Ptr1      S          *
D Ptr2      S          *
....
C          ALLOC    25      Ptr1
C          DEALLOC
*-----*
* After this point, F1d1 should not be accessed since the
* basing pointer ptr1 no longer points to allocated storage.
*-----*
C          CALL    'SOMEPM'
*-----*
* During the previous call to 'SOMEPM', 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.
*-----*
C          EVAL    F1d1 = *ALL'a'

```

Figure 53. 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. See the *System API Reference* for specific information about the storage management bindable APIs.

Figure 54 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 54. /COPY file DYNARRI containing prototypes for DYNARRAY module

Figure 55 on page 121 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 55 (Part 1 of 2). 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 55 (Part 2 of 2). Global variables and local prototypes for DYNARRAY

Figure 56 on page 123 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      INITIALLOC      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 56 (Part 1 of 5). 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

```

Figure 56 (Part 2 of 5). DYNARRAY Subprocedures

```

=====
* 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

```

Figure 56 (Part 3 of 5). DYNARRAY Subprocedures

```

*
* 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

```

Figure 56 (Part 4 of 5). DYNARRAY Subprocedures

```

=====
* 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

```

Figure 56 (Part 5 of 5). DYNARRAY Subprocedures

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 57 on page 127 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 57. Sample module using procedures in DYNARRAY

Chapter 10. 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.

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 137 and “Examples of CALL and CALLB” on page 151. For exam-

ples of using procedure pointers, see the section on the procedure pointer data type in *ILE RPG for AS/400 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 58 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.

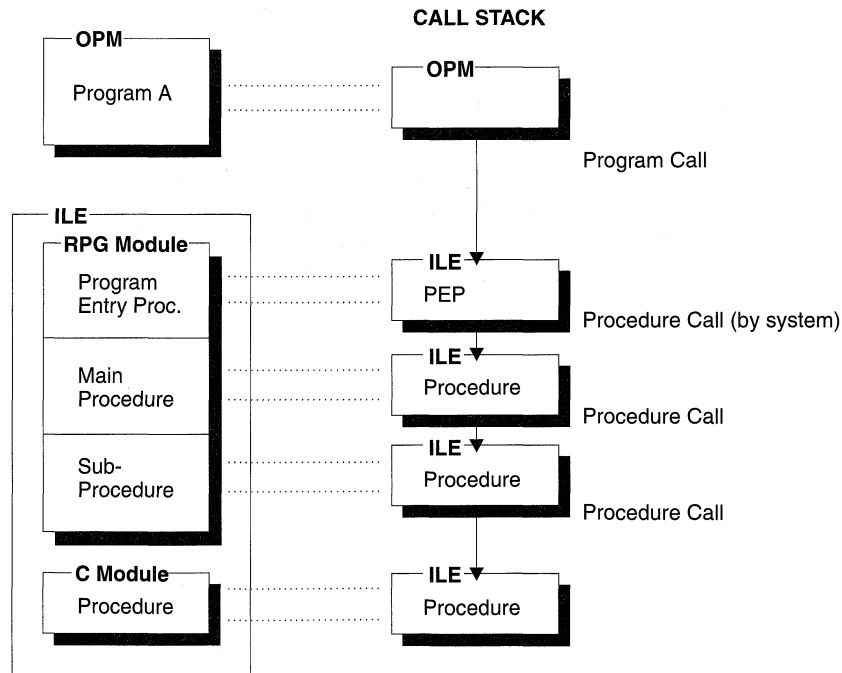


Figure 58. Program and Procedure Calls on the Call Stack

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 only 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.

A 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 main procedure.

Try to avoid situations that might inadvertently lead to recursive calls. For example, suppose there are three modules, as shown in Figure 59.

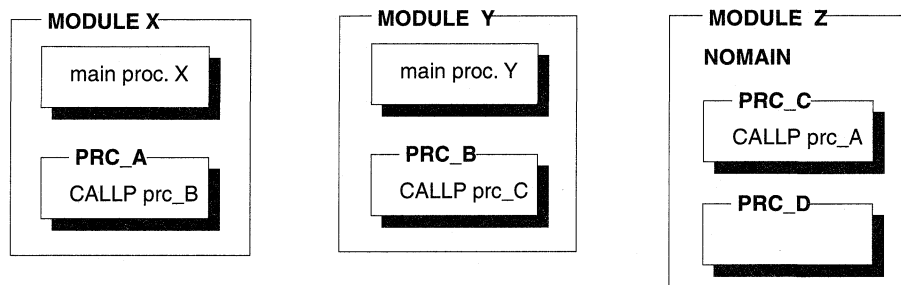
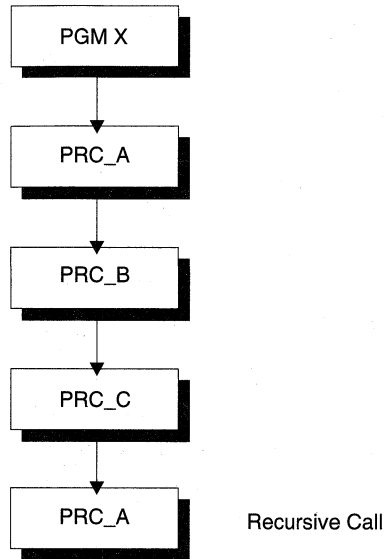


Figure 59. Three Modules, each with subprocedures

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 60 on page 133. Note that the most recent call stack entry is at the bottom.



Call Stack (bottom entry is most recent)

Figure 60. 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
- 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.

Table 8 on page 135 associates the above considerations with the two types parameters: prototyped or non-prototyped.

Table 8. Parameter Passing Options

Parameter Option	Prototyped	Not Prototyped	See Page
Compile-time parameter checking	Yes		137
Pass by reference	Yes	Yes	137
Pass by value	Yes (b)		138
Pass by read-only reference	Yes		139
Pass operational descriptors	Yes (b)	Yes (b)	140
Pass *OMIT	Yes (b)	Yes (b)	141
Control parameter omission	Yes	Yes	142
Get number of passed parameters	Yes	Yes	143
Disallow incorrect parameter length	Yes	No	148
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. It 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.

For information on passing prototyped parameters, see “Passing Prototyped Parameters” on page 137.

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 free-form syntax:

```
C          CALLP      NAME{ (PARM1 { :PARM2 ... } ) }
```

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.
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 61 shows the prototype for a procedure CVTCHR that takes a numeric input parameter and returns a character string. Figure 62 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 61. 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 62. Calling a Prototyped Procedure within an Expression

Examples of Free-Form Call

For examples of using the CALLP operation, see:

- Figure 22 on page 41
- Figure 43 on page 97
- Figure 100 on page 212
- Figure 70 on page 146
- Figure 112 on page 242

For examples of calling by using an expression, see:

- Figure 4 on page 10
- Figure 19 on page 39
- Figure 38 on page 79
- Figure 100 on page 212

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 as both are compiled using the 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.

Note: OS/400 program calls require that parameters be passed by reference. Consequently, you cannot pass a parameter by value to a program.

Passing by value allows you to:

- Pass literals and expressions as parameters.
- Pass parameters that do not match exactly the type and length that are expected.
- Pass a variable that, from the caller's perspective, will not be modified.

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

One primary use for passing by value is that 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.

To pass a parameter by value, specify the keyword **VALUE** on the parameter definition in the prototype, as shown in the figures below.

```

*-----
* 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 63. 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 64. Procedure Interface Definition for DO_CALC Procedure

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 65 on page 140 shows an example of a prototype definition for the ILE CEE API `CEETSTA` (Test for omitted argument).

```

*-----
* 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 HaveParm         S                10I 0
...
C                  CALLP  CEETSTA(HaveParm : 3)
C                  IF    HaveParm = 1
C                  do something with third parameter
C                  ENDF

```

Figure 65. 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. In the *System API Reference*, 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 fields and subfields. Operational descriptors are not available for data structures, arrays, or tables. In addition, operational descriptors are not available for data of type numeric, date, time, timestamp, basing pointer or procedure pointer.

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 66 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 66. Requesting Operational Descriptors for a Prototyped Procedure

For an example of how to use operational descriptors see “Sample Service Program” on page 94. The example consists of a service program which converts character strings which are passed to it to their hexadecimal equivalent. The service program uses operational descriptors to determine the length of the character string and the length to be converted.

Omitting Parameters

When calling a procedure, you may sometimes 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:

- Pass *OMIT
- 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. An omitted parameter cannot be referenced by the called procedure; if it is, an error 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 65 on page 140 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 94 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 QCMDXEC (Execute Command) which has an optional third parameter. To allow for this parameter, the prototype for QCMDXEC could be written as shown in Figure 67.

```
*-----  
* This prototype for QCMDXEC 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('QCMDXEC')  
D  cmd             3000A          OPTIONS(*VARSIZE)  CONST  
D  cmdlen         15P 5          CONST  
D                               3A    CONST OPTIONS(*NOPASS)
```

Figure 67. Prototype for System API QCMDXEC 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. For the 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.

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 68 on page 144 shows the source for this procedure. Figure 69 on page 145 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 68 (Part 1 of 2). 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
C                               ENDSR

```

Figure 68 (Part 2 of 2). Source for procedure FMTADDR

```

*=====
* 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 69. Source for /COPY member with Prototype for Procedure FMTADDR

Figure 70 on page 146 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
*=====
FQSYSVRT  0   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 70 (Part 1 of 2). 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
0          Address

**
00123Bumble Bee Drive
01243Hummingbird Lane
**
00003Cowslip Street      Toronto
01150Eglinton Avenue    North York
**
00012Jasper Avenue      Edmonton      Alberta
00027Avenue Road        Sudbury      Ontario

```

Figure 70 (Part 2 of 2). Source for procedure PRTADDR

To create these programs, follow these steps:

1. To create FMTADDR, using the source in Figure 68 on page 144, type:
CRTRPGMOD MODULE(MYLIB/FMTADDR)
2. To create PRTADDR, using the source in Figure 70 on page 146, 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 71 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 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 71. Prototype for System API QCMDEXC with *VARSIZE Parameter

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. Table 9 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.

<i>Table 9. RPG Parameter Passing Methods</i>	
Passing By Reference	
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)
Passing By Value	
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
Passing By Read-Only Reference	
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
Notes:	
<p>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:</p> <pre>C PARM Fld1 TEMP</pre> <p>Fld1 is protected from being changed, but TEMP is not. There is no expectation that the parameter will not be changed.</p>	

For information on the data types supported by different HLLs, consult the appropriate language manual.

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 140.

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

Examples of CALL and CALLB

For examples of using the CALL operation, see:

- "Sample Source for Debug Examples" on page 211, for example of calling an RPG program.

For examples of using the CALLB operation, see:

- Figure 45 on page 99, for an example of calling a procedure in a service program.
- Figure 56 on page 123, for an example of calling bindable APIs.
- "CUSMAIN: RPG Source" on page 347, 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 10 on page 152 shows how factor 1 and factor 2 are used.

<i>Table 10. Meaning of Factor 1 and Factor 2 Entries in PARM Operation</i>		
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 143.
- 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 141.

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 *ILE RPG for AS/400 Reference*.

For examples of the PARM operation see:

- Figure 47 on page 104
- Figure 42 on page 95

- Figure 111 on page 240

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 47 on page 104 and Figure 111 on page 240.

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 LR is on, 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 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 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 *ILE RPG for AS/400 Reference*.

Normal End

A 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 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 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 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 main procedure, with the exception of exported variables, a fresh copy is available for processing. (Exported variables 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

A 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 238).
- 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 main procedure ends abnormally, the following occurs:

- All files that are open are closed.
- Any data areas locked by the procedure are unlocked.
- If the 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 227.)

Returning without Ending

A 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 *ILE RPG for AS/400 Reference*)

If you call a 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

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. However, other than the removal of the subprocedure from the call stack no termination actions are performed until the main procedure of the program ends. In other words, all the actions listed for the normal end of a 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. Again, no further actions occur until the main procedure ends.

If 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 DFACTGRP(*YES), since procedure calls are not allowed in these procedures.

Note that if the main procedure is not active, or if there is no 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 *System API Reference*.

Using Bindable APIs

Bindable application programming interfaces (APIs) are available to all ILE languages. In some cases they provide additional function beyond that provided by a specific ILE language. They are also useful for mixed-language applications because they are HLL independent.

The bindable APIs provide a wide range of functions including:

- Activation group and control flow management
- Storage management
- Condition management
- Message services
- Source Debugger
- Math functions
- Call management
- Operational descriptor access

You access ILE bindable APIs using the same call mechanisms used by ILE RPG 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 *System API Reference*. Figure 72 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              ...
C              PARM          parmn
C              PARM          feedback
```

Figure 72. 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 *System API Reference*.

Examples of Using Bindable APIs

For examples of using bindable APIs, see:

- “Sample Service Program” on page 94, for an example of using CEEDOD
- “Managing Your Own Heap Using ILE Bindable APIs” on page 120, for an example of using CEEGTST, CEEFRST, and CEECZST.
- “Using a Condition Handler” on page 239, for an example of using CEEHDLR and CEEHDLU.
- “Using Cancel Handlers” on page 244, 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 OS/400 Graphics, which includes the Graphical Data Display Manager (GDDM®, a set of graphics primitives for drawing pictures), and Presentation Graphics Routines (a set of business charting routines). Factor 2 must contain the literal or named constant 'GDDM' (not a 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 OS/400 graphics routines.

For more information on OS/400 Graphics, graphics routines and parameters, see the *GDDM Programming Guide* manual and the *GDDM Reference*.

Note: You can call OS/400 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.

Debugging and Exception Handling

This section describes how to:

- Debug an ILE application using the ILE source debugger
- Write programs that handle exceptions
- Obtain a dump

Chapter 11. 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 debugger.

This chapter describes how to use the ILE source debugger 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 appendix on debugging in the *CL Reference* for more information on preventing unintended modification of production files.

See the chapter on debugging in *ILE Concepts* for more information on the ILE source debugger (including authority required to debug a program or service program and the effects of optimization levels).

The ILE Source Debugger

The ILE source debugger is used to detect errors in and eliminate errors from program objects and service programs. Using debug commands with any ILE program 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 debugger, you must select a debug view when you create a module object or program object using CRTRPGMOD or CRTBNDRPG. After starting the debugger you can set breakpoints and then call the program.

When a program stops because of a breakpoint or a step command, the pertinent module object's view is shown on the display at the point where the program stopped. At this point you can perform other actions such as displaying or changing field values.

Note: If your program has been optimized, you can still display fields, but their values may not be reliable. To ensure that the content of fields or data structures contain their correct (current) values, specify the NOOPT keyword on the appropriate Definition specification. To change the optimization level, see “Changing the Optimization Level” on page 87.

Debug Commands

Many debug commands are available for use with the ILE source debugger. The debug commands and their parameters are entered on the debug command line displayed on the bottom of the Display Module Source and Evaluate Expression displays. These commands can be entered in uppercase, lowercase, or mixed case.

Note: The debug commands entered on the debug command line are not CL commands.

The 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 <i>line-number</i> WHEN <i>expression</i> 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. Currently it does not apply to ILE RPG.
THREAD	Allows you to display the Work with Debugged Threads display or change the current thread. Currently it does not apply to ILE RPG.
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 debugger commands.

The online help for the ILE source debugger describes the debug commands, explains their allowed abbreviations, and provides syntax diagrams for each command. It also provides examples in each of the ILE languages of displaying and changing variables using the source debugger.

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 debugger. For more information, see “Starting the ILE Source Debugger” on page 169

The debug views are summarized in the following table:

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 produced by the ILE RPG compiler. The information contained in the listing view is dependent on whether `OPTION(*SHOWCPY)`, or `OPTION(*EXPDDS)` are specified for either create command. `OPTION(*SHOWCPY)` includes /COPY members in the listing, `OPTION(*EXPDDS)` includes externally described files.

Note that if you specify indentation in the compiler listing (via the `INDENT` parameter), the indentation will not appear in the listing view.

You create a listing view to debug a module by using the `*LIST` or `*ALL` options on the `DBGVIEW` parameter for either the `CRTRPGMOD` or `CRTBNDRPG` commands when you create a module.

The compiler creates the listing view while the module object (`*MODULE`) is being generated. The listing view is created by copying the text of the appropriate source members into the module object. There is no dependency on the source members upon which it is based, once the listing view is created.

For example, to create a listing view for a program TEST1 that contains expanded DDS type:

```

CRTBNDRPG PGM(MYLIB/TEST1) SRCFILE(MYLIB/QRPGLESRC)
          SRCMBR(TEST1) OUTPUT(*PRINT)
          TEXT('ILE RPG/400 program TEST1')
          OPTION(*EXPDDS) DBGVIEW(*LIST)

```

Specifying `DBGVIEW(*LIST)` for the `DBGVIEW` parameter and `*EXPDDS` for the `OPTION` parameter on either create command creates a listing view with expanded DDS for debugging the source for TEST1. Note that `OUTPUT(*PRINT)` and `OPTION(*EXPDDS)` are both default values.

Creating a Statement View

A **statement view** allows the module object to be debugged using statement numbers and the debug commands. Since the source will not be displayed you must make use of statement numbers which are shown on the left-most column of the source section of the compiler listing. In other words, to effectively use this view, you will need a compiler listing.

You create a statement view to debug a module by using the `*STMT` option on the `DBGVIEW` parameter for either the `CRTRPGMOD` or `CRTBNDRPG` commands when you create a module.

Use this view when:

- You have storage constraints, but do not want to recompile the module or program if you need to debug it.
- You are sending compiled objects to other users and want to be able to diagnose problems in your code using the debugger, but you do not want these users to see your actual code.

For example, to create a statement view for the program `DEBUGEX` using `CRTBNDRPG`, type:

```

CRTBNDRPG PGM(MYLIB/DEBUGEX) SRCFILE(MYLIB/QRPGLESRC)
          TEXT('ILE RPG/400 program DEBUGEX')

```

To create a statement view for a module using `CRTRPGMOD`, type:

```

CRTRPGMOD MODULE(MYLIB/DBGEX) SRCFILE(MYLIB/QRPGLESRC)
          TEXT('Entry module for program DEBUGEX')

```

By default a compiler listing and a statement view are produced. Using a compiler listing to obtain the statement numbers, you debug the program using the debug commands.

If the default values for either create command have been changed, you must explicitly specify `DBGVIEW(*STMT)` and `OUTPUT(*PRINT)`.

Starting the ILE Source Debugger

Once you have created the debug view (statement, source, `COPY`, or listing), you can begin debugging your application. To start the ILE source debugger, use the `Start Debug (STRDBG)` command. Once the debugger is started, it remains active until you enter the `End Debug (ENDDBG)` command.

Initially you can add as many as 20 program objects to a debug session by using the `Program (PGM)` parameter on the `STRDBG` command. They can be any combi-

nation 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 debugger.) 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 debugger 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 debugger.

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 73 on page 171. DEBUGEX consists of two modules, an RPG module DBGEX and a C module cproc. See “Sample Source for Debug Examples” on page 211 for the source for DBGEX, cproc, and RPGPGM.

If the entry module has a root source, COPY, or listing view, then the display will show the source of the entry module of the first program. In this case, the program was created using DBGVIEW(*ALL) and so the source for the main module, DBGEX, is shown.

```

                                Display Module Source
Program:  DEBUGEX           Library:  MYLIB           Module:  DBGEX
 1  *=====
 2  *  DEBUGEX - Program designed to illustrate use of ILE source
 3  *                debugger with ILE RPG 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 73. 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 debugger 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 debugger. (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 debugger 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 94 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 74 on page 173.
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 74. 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 75 on page 174.
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
-----
  4   CVTHEXPGM   *LIBL   *PGM
      CVTHEXPG     MYLIB   *MODULE
  4   CVTTOHEX   MYLIB   *SRVPGM
      CVTTOHEX     MYLIB   *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 75. 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 debugger maps to equivalent positions in the view you are changing to. When you change the module, the runnable statement on the displayed view is stored in memory and is viewed when the module is displayed again. Line numbers that have breakpoints set are highlighted. When a

breakpoint, step, or message causes the program to stop, and the display to be shown, the statement where the breakpoint occurred is highlighted.

Viewing a Different Module

To change the module object that is shown on the Display Module Source display, use option 5 (Display module source) on the Work with Module List display. You access the Work with Module List display from the Display Module Source display by pressing F14 (Work with Module List).

If you use this option with 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 73 on page 171 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 76 on page 176 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 77 on page 176. 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 76. 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 77. 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 78. 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 78.

```

                                     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 78. 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 its 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: This release does not support running ILE RPG procedures in a multi-threaded environment.

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 debugger.

If you are debugging a module or program created with a statement view, then you can set or remove breakpoints using statement numbers obtained from the compiler

listing. For more information on using statement numbers, see “Setting and Removing Job Breakpoints Using Statement Numbers” on page 185.

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 job breakpoint
- 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. 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.

- 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 79 on page 180. 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 79. Setting an Unconditional Job Breakpoint

- After the breakpoint is set, press F3 (Exit) to leave the Display Module Source display. The breakpoint is not removed.
- 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

Note: This section is added for your general reference. This release does not support running ILE RPG procedures in a multi-threaded environment.

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

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 80 on page 182.

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.

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 184 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 80 shows the Work with Module Breakpoints display after adding the conditional breakpoint.

```

                                Work with Module Breakpoints
Program . . . . : DEBUGEX          Library . . . . : MYLIB
Module . . . . : DBGEX            Type . . . . . : *PGM
System: TORASD80

Type options, press Enter.
1=Add  4=Clear

Opt   Line   Condition
-     127   *in02='1'
-     88
-     102

Command
====>
F3=Exit  F4=Prompt  F5=Refresh  F9=Retrieve  F12=Cancel
Breakpoint added to line 127.
                                Bottom

```

Figure 80. 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 debugger. 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 debugger. Your breakpoint is not removed.
3. Call the program. When a breakpoint is reached, the program stops, and the Display Module Source display is shown again.

Display Module Source			
Program:	DEBUGEX	Library:	MYLIB
		Module:	DBGEX
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 Array.
133			*-----
134	C	MOVE	4 Array
135			
136			*-----
			More...
Debug . . .		break 128 when BigDate='1994-09-30'	
<hr/>			
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 81. 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 12 on page 185 for the possible combinations of non-numeric conditional breakpoint expressions.

The sort sequence table used by the source debugger for expressions of type Char-8 is the sort sequence table specified on the SRTSEQ parameter for the CRTRPGMOD or CRTBNDRPG commands.

If the resolved sort sequence table is *HEX, no sort sequence table is used. Therefore, the source debugger uses the hexadecimal values of the characters to determine the sort sequence. Otherwise, the specified sort sequence table is used to assign weights to each byte before the comparison is made. Bytes between, and including, shift-out/shift-in characters are **not** assigned weights. This differs from the way ILE RPG handles comparisons; all characters, including the shift-out/shift-in characters, are assigned weights.

Notes:

1. The alternate sequence specified by ALTSEQ (*SRC) on the Control specification is not available to the ILE source debugger. Instead the source debugger uses the *HEX sort sequence table.
2. The name of the sort sequence table is saved during compilation. At debug time, the source debugger uses the name saved from the compilation to access the sort sequence table. If the sort sequence table specified at compilation time resolves to something other than *HEX or *JOB RUN, it is important the sort sequence table does *not* get altered before debugging is started. If the table cannot be accessed because it is damaged or deleted, the source debugger uses the *HEX sort sequence table.

Table 12. Non-numeric Conditional Breakpoint Expressions

Type	Possible
Char-8	<ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • 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
<p>Notes:</p> <ol style="list-style-type: none"> 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, which for ILE RPG is the module name. (ILE RPG only allows one procedure per module.) The variable *statement-number* is the statement number from the compiler listing on which you want to set a breakpoint.

For example, to set the breakpoint at statement-number 2 for the procedure TEST, type:

```
BREAK TEST/2
```

The breakpoint is set on the 'SETON LR----' line.

Note: The statement-number is labeled as the Line Number in the source listing.

```

Line  <----- Source Specifications ----->
Number .....1.....2.....3.....4.....5.....6.....7.....
          S o u r c e   L i s t i n g
1 C          MOVEL      'ABCD'      X          5
2 C          SETON
          * * * * *   E N D   O F   S O U R C E   * * * * *

Display Module Source

```

Section of the compiler listing.

```

Program:  TEST          Library:  MYLIB          Module:  TEST
(Source not available.)

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

Bottom

```

Figure 82. Setting a Breakpoint Using Statement View

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 statement number

on the debug command line.

Setting and Removing Conditional Thread Breakpoints

Note: This section is added for your general reference. This release does not support running ILE RPG procedures in a multi-threaded environment.

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 debugger 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; other-

wise, 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:  DEBUGGER

Type options, press Enter.
4=Clear 5=Display

Opt   Num   Variable           Address           Length
-     1     SALARY             080090506F027004 4

Command
===>
F3=Exit F4=Prompt F5=Refresh F9=Retrieve F12=Cancel
Bottom

```

Figure 83. 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          :  DEBUGGER
:
: 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                  :
:
:
:
.....
Command
===>
F3=Exit F4=Prompt F5=Refresh F9=Retrieve F12=Cancel
Bottom

```

Figure 84. 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 debugger 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					
-----Job-----			NUM	LENGTH	System: DEBUGGER 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 85. 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 AS/400 system 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 86 on page 193.

```

                                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 86. 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.

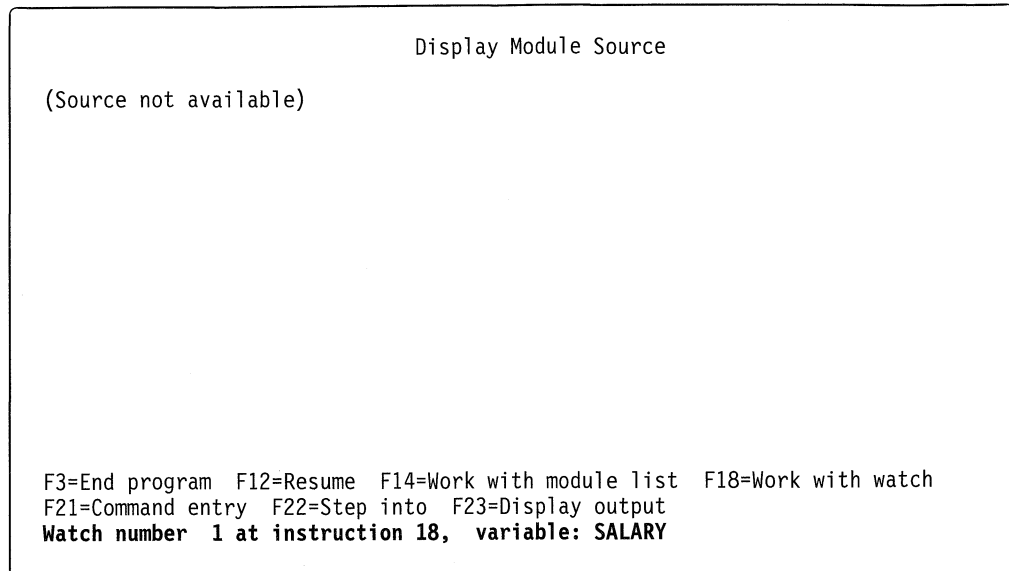


Figure 87. Example of a Display Module Source Panel

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 debugger. The program object resumes running on the next statement of the module object in which the program stopped. Typically, a breakpoint is used to stop the program object.

You can step 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 debugger 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 debugger, from the initial Display Module Source display, enter

```
STEP 1 INTO
```

This will set the step count to 1. Use the F12 key to return to the command line and then call the program. The program will stop at the first statement with debug data.

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 88.

```

                                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 88. 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 89 on page 198.

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 debugger.
4      *
5      *  This program receives a parameter InputParm from DEBUGEX,
6      *  displays it, then returns.
7      *=====
8
9      D  InputParm            S              4P 3
10
11     C    *ENTRY            PLIST
12     C    PARM              InputParm
13     C    InputParm       DSPLY
14     C    SETON
                                           Bottom
Debug . . . _____
F3=End program  F6=Add/Clear breakpoint  F10=Step  F11=Display variable
F12=Resume     F17=Watch variable  F18=Work with watch  F24=More keys
Step completed at line 13.

```

Figure 89. Stepping into RPGPGM

If the ILE source debugger 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 90 on page 199. 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 90. 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

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 91 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 91. Displaying a Field using the EVAL debug command

Figure 92 on page 201 shows the use of the EVAL command with different types of RPG fields. The fields are based on the source in Figure 100 on page 212. Additional examples are also provided in the source debugger online help.

Scalar Fields	RPG Definition
> EVAL String STRING = 'ABCDEF'	6A INZ('ABCDEF')
> EVAL Packed1D0 PACKED1D0 = -093.40	5P 2 INZ(-93.4)
> EVAL ZonedD3D2 ZONEDD3D2 = -3.21	3S 2 INZ(-3.21)
> EVAL Bin4D3 BIN4D3 = -4.321	4B 3 INZ(-4.321)
> EVAL Int5 INT5 = -2046	5I 0 INZ(-2046)
> EVAL Int10 INT10 = -31904	10I 0 INZ(-31904)
> EVAL Unsigned5 UNSIGNED5 = 2046	5U 0 INZ(2046)
> EVAL Unsigned10 UNSIGNED10 = 31904	10U 0 INZ(31904)
> EVAL DBCSString DBCSSTRING = '"BCCDD"'	3G INZ(G' BCCDD ')
> EVAL NullPtr NULLPTR = SYP:*NULL	* INZ(*NULL)
Based Fields	
> EVAL String STRING = 'ABCDEF'	6A INZ('ABCDEF')
> EVAL BasePtr BASEPTR = SPP:C01947001218	* INZ(%ADDR(String))
> EVAL BaseString BASESTRING = 'ABCDEF'	6A BASED(BasePtr)
Date, Time, Timestamp Fields	
> EVAL BigDate BIGDATE = '9999-12-31'	D INZ(D'9999-12-31')
> EVAL BigTime BIGTIME = '12.00.00'	T INZ(T'12.00.00')
> EVAL BigTstamp BIGTSTAMP = '9999-12-31-12.00.00.000000'	Z INZ(Z'9999-12-31-12.00.00.000000')

Figure 92. Sample EVAL commands based on Module DBGEX

Unexpected Results when Evaluating Variables

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 debugger 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, 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 93 shows the use of EVAL with the array in DBGEX.

```
> EVAL Array                                3S 2 DIM(2) INZ(1.23)
  ARRAY(1) = 1.23    ** Display full array **
  ARRAY(2) = 1.23

> EVAL Array(2)    ** Display second element **
  ARRAY(2) = 1.23

> EVAL Array(1..2) ** Display range of elements **
  ARRAY(1) = 1.23
  ARRAY(2) = 1.23
```

Figure 93. 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 94 on page 203 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 94. Sample EVAL commands for a Table

Displaying Data Structures

You display the contents of a data structure or its subfields as you would any standalone field. You simply use the data structure name after EVAL to see the entire contents, or the subfield name to see a subset.

When displaying a multiple-occurrence data structure, an EVAL on the data structure name will show the subfields using the current index. To specify a particular occurrence, specify the index in parentheses following the data structure name. For example, to display the contents of the second occurrence of DS1, type:

```
EVAL DS1(2)
```

Similarly, to view the contents of a particular occurrence of a subfield, use the index notation.

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 95 on page 204 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
  FLD1 OF DS1 = 'ABCDE'          OCCURS(3)
  FLD1A OF DS1(1) = 'A'          5A  INZ('ABCDE')
  FLD1A OF DS1(2) = 'B'          1A  DIM(5) OVERLAY(FLD1)
  FLD1A OF DS1(3) = 'C'          5B 2 INZ(123.45)
  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'

```

Figure 95. 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 96 on page 205 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 96. 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 = 'aaaaaaaaabbbbbbbbbb'
```

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.

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'
```

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 97 shows some examples of the use of %SUBSTR based on the source in Figure 100 on page 212.

```

> 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 97. 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 94 on page 203 and Figure 95 on page 204.

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.

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'

When assigning literals to fields, the normal RPG rules apply:

- Character literals should be in quotes.
- Graphic literals should be specified as G'oDDDDi', where o is shift-out and i is shift-in.
- Hexadecimal literals should be in quotes, preceded by an 'x'.
- Numeric literals should not be in quotes.

Note: You cannot assign a figurative constant to a field using the EVAL debug command. Figurative constants are not supported by the EVAL debug command.

Figure 98 on page 209 shows some examples of changing field values based on the source in Figure 100 on page 212. 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 98. 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 99 shows some examples of displaying field attributes based on the source in Figure 100 on page 212. 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 Int5
  TYPE = INTEGER, LENGTH = 2 BYTES
> ATTR Unsigned10
  TYPE = CARDINAL, LENGTH = 4 BYTES
> ATTR Array
  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 99. 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 100 on page 212 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 101 on page 216 and Figure 102 on page 216 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 100 on page 212, 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 102 on page 216, 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 101 on page 216, 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 Int5            S          5I 0 INZ(-2046)
D Int10           S         10I 0 INZ(-31904)
D Unsigned5       S          5U 0 INZ(2046)
D Unsigned10      S         10U 0 INZ(31904)
D DBCSString      S          3G  INZ(G'"BCCDD"')
```

Figure 100 (Part 1 of 4). Source for Module **DBGEX**

```

* 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)
*
D DS3          DS
D Title        5A  INZ('Mr. ')
D LastName     10A INZ('Jones ')
D FirstName    10A INZ('Fred  ')

*-----*
* 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

```

Figure 100 (Part 2 of 4). Source for Module DBGEX

```

=====
* 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)

```

Figure 100 (Part 3 of 4). Source for Module DBGEX


```

*-----*
* 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 Array.
*-----*
C          MOVE    4              Array

*-----*
* 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

```

Figure 100 (Part 4 of 4). Source for Module DBGEX. DBGEX is the main module of the program DEBUGEX.

```

=====
* RPGPGM - Program called by DEBUGEX to illustrate the STEP *
*          functions of the ILE source debugger.           *
*                                                       *
* This program receives a parameter InputParm from DEBUGEX, *
* displays it, then returns.                               *
=====

D InputParm      S          4P 3

C  *ENTRY        PLIST
C                   PARM          InputParm
C  InputParm     DSPLY
C                   SETON                LR

```

Figure 101. 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 102. Source for C Procedure cproc. cproc is called by DBGEX.

Chapter 12. 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, 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 220
- "Specifying Error Indicators or the 'E' Operation Code Extender" on page 227

Additionally, to learn how ILE exception handling works, read:

- "Exception Handling Overview" (for general concepts)
- "Using RPG-Specific Handlers" on page 227
- The sections on error handling in *ILE Concepts*.

For information on exception handling and the RPG cycle, see *ILE RPG for AS/400 Reference*.

Note: In this book the term 'exception handling' is used to refer to both exception handling and error handling. However, for consistency with other RPG terms, the term 'error' is used in the context of 'error indicator' and 'error subroutine'.

Exception Handling Overview

Exception handling is the process of:

- Examining an exception message which has been issued as a result of a run-time error
- Optionally modifying the exception to show that it has been received (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 131 for further discussion of a call stack.)

Figure 103 on page 219 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 103 on page 219 Procedure P1 is the control boundary.

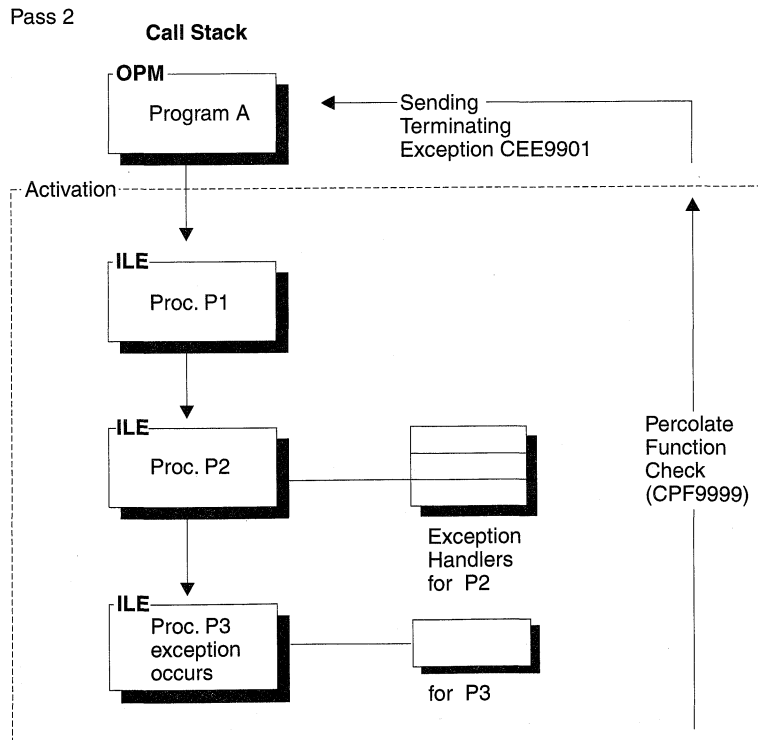
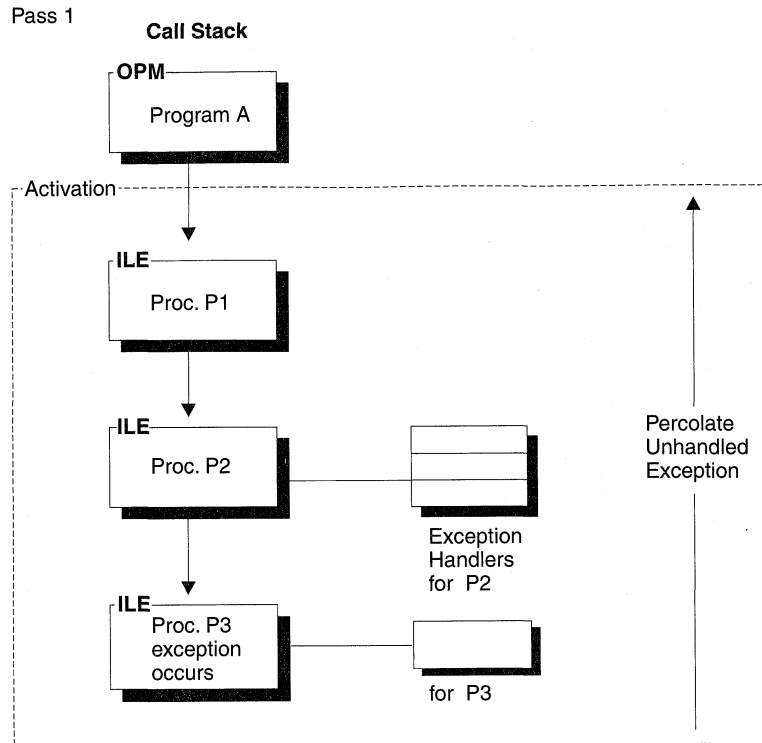


Figure 103. Call Stack and Exception Message Percolation

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 excep-

tion. 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 three types of exception handling mechanisms:

- An error indicator or an 'E' operation code extender handler
- An error subroutine handler
- A default exception handler

RPG categorizes exceptions into two classes, program and file; this determines which type of error subroutine is called. Some examples of program exceptions are division by zero, out-of-bounds array index, or SQRT of a negative number. Some examples of file exceptions are undefined record type or a device error.

There are four 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. Code a file error subroutine, which is defined by the INFSR keyword on a file description specification, for file exceptions. The file error subroutine can only be coded in the main source section. You cannot code an INFSR for a file that is used in a subprocedure.
4. 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 Main Procedure

When an exception occurs within a 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 functions %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*

- 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.

4. If no error indicator, 'E' extender, or error subroutine is coded, then the RPG default error handler is invoked.

- If the exception is *not* a function check, then the exception will be percolated.

- If the exception is a function check, then an inquiry message will be displayed. If the 'G' or 'R' option is chosen, the function check will be handled and control will resume at the appropriate point (*GETIN for 'G' or the same calculation specification that received the exception for 'R') in the procedure. Otherwise, the function check will be percolated and the procedure will be abnormally terminated.

See "Unhandled Exceptions" on page 223 for a full description of the RPG default handler.

Exception Handling within Subprocedures

Exception handling within a subprocedure differs from a main procedure in the following ways:

- Because you cannot code an *INFSR subroutine, you should handle file errors using error indicators or the 'E' operation code extender.
- There is no default handler; in other words, users will never see an inquiry message.

Exception handling within a subprocedure differs from a 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 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 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 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 373 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. I/O error subroutine handler
3. ILE condition handler
4. Program error subroutine handler
5. RPG default handler for unhandled exceptions (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 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 104 depicts this scenario and the events which occur.

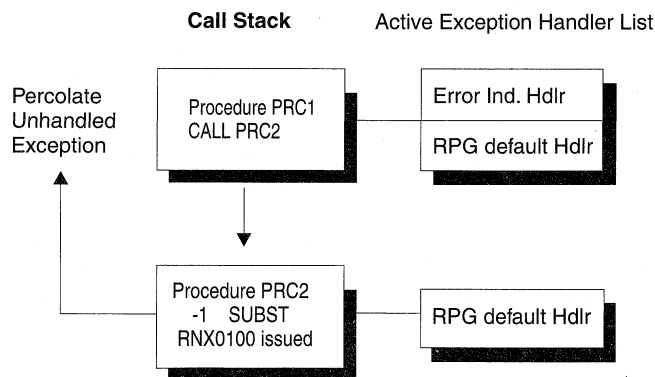


Figure 104. Scenario for Unhandled Escape Message

The following then occurs:

1. Since there is no error indicator or *PSSR error subroutine coded on the SUBST operation in PRC2, PRC2 cannot handle the program error, and so it is unhandled.
2. Since it is not a function check, it is percolated (passed up the call stack) to PRC1.
3. PRC1 receives (handles) the same exception message, and sets on the error indicator on the CALL operation with the side effect that PRC2 is terminated.

4. Processing then continues in PRC1 with the statement following the CALL operation.

Note: The same exception handling events described would apply to a procedure call (CALLB operation) as well.

Example of Unhandled Function Check

The following scenario describes the events which occur when a function check occurs in a 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 coded and there is no *PSSR error subroutine.
5. PRC2 has a pointer address error.

When PGM1 calls PGM2, a pointer error occurs because the basing pointer is defined as null. Consequently, MCH1306 is generated. A function check occurs when PRC2 tries to percolate the exception past the control boundary. Figure 105 on page 226 depicts this scenario and the events which occur.

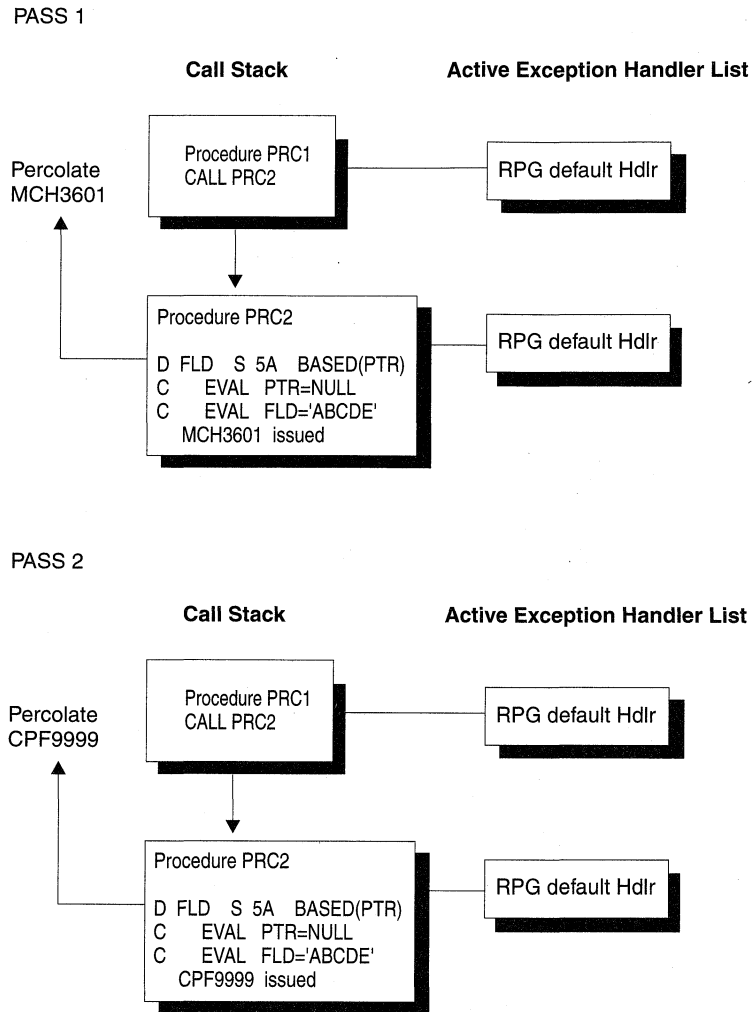


Figure 105. Scenario for Unhandled Function Check

The following then occurs:

1. Since there are no error indicators coded in PRC2, PRC2 cannot handle the function check, and so it is unhandled.
2. Since it is a function check, an inquiry message is issued describing the originating condition.
3. Depending on the response to the inquiry message, PRC2 may be terminated and the exception percolated to PRC1 (response is 'C') or processing may continue in PRC2 (response is 'G').

Optimization Considerations

While running a *FULL optimized program, the optimizer may keep frequently used values in machine registers and restore them to storage only at predefined points during normal program processing. Exception handling may break this normal processing and consequently program variables contained in registers may not be returned to their assigned storage locations.

Specifically, variables may not contain their current values if an exception occurs and you recover from it using one of:

- *PSSR error subroutine

- INFSR error subroutine
- User-defined exception handler
- The Go ('G') option from an inquiry message.
- The Retry ('R') option from an inquiry message.

ILE RPG automatically defines indicators such that they contain their current values even with full optimization. To ensure that the content of fields or data structures contain their correct (current) values, specify the NOOPT keyword on the appropriate Definition specification.

For more information on the NOOPT keyword, see *ILE RPG for AS/400 Reference*. For more information on optimization, see “Changing the Optimization Level” on page 87.

Using RPG-Specific Handlers

ILE RPG provides three ways for you to enable HLL-specific handlers and to recover from the exception:

1. error indicators or 'E' operation code extender
2. INFSR error subroutine
3. *PSSR error subroutine.

You can obtain more information about the error which occurred by coding the appropriate data structures and querying the relevant data structure fields.

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 *ILE RPG for AS/400 Reference* provides more information on the *PSSR and INFSR error subroutines, on the EXSR operation code, and on the INFDS and PSDS data structures.

Specifying Error Indicators 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 and '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 73 and 74 for the operation codes listed in Table 13 on page 228 (except for CALLP). 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 in Table 13. 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.

Table 13. Operation Codes Allowing Extender 'E' or an Error Indicator in Positions 73-74

ACQ (e)	ADDDUR (e)	ALLOC (e)	CALL (e)
CALLB(d e)	CALLP (e m/r) ¹	CHAIN (e n)	CHECK (e)
CHECKR (e)	CLOSE (e)	COMMIT (e)	DEALLOC(e/n)
DELETE (e)	DSPLY (e)	EXFMT (e)	EXTRCT (e)
FEOD (e)	IN (e)	NEXT (e)	OCCUR (e)
OPEN (e)	OUT (e)	POST (e)	READ (e n)
READC (e)	READE (e n)	READP (e n)	READPE (e n)
REALLOC (e)	REL (e)	RESET (e)	ROLBK (e)
SCAN (e)	SETGT (e)	SETLL (e)	SUBDUR (e)
SUBST (e p)	TEST (e d/t/z)	UNLOCK (e)	UPDATE (e)
WRITE (e)	XLATE (e p)		
Notes:			
1. CALLP (e m/r) is an extended Factor-2 operation code and cannot have an error indicator. However, program status and error conditions can be determined by specifying the 'e' extender with this operation code.			

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 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 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 236.

Using a File Error (INFSR) Subroutine

To handle a file error or exception in a main procedure 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 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 238.
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 106 on page 231 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 0 E DISK
FPRINT 0 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 106 (Part 1 of 2). Example of File Exception Handling

```

*-----*
* Access the product master file using the transaction product *
* number. *
*-----*
C      TRNPRDNO      CHAIN      PRDREC      10
*-----*
* If the record is found, update the quantity in the master file. *
*-----*
C              IF      NOT *IN10
C              SUB      TRNQTY      PRDQTY
C              UPDATE      PRDREC
*-----*
* If the record is not found, write to the error report *
*-----*
C              ELSE
C              EXCEPT      NOTFOUND
C              ENDIF
C              SETON      LR
*-----*
* Error handling routine. *
*-----*
C      PrdInfsr      BEGSR
*-----*
* If the master record is currently locked, write the transaction *
* record to the back log file and skip to next transaction. *
*-----*
C      PrdStatus      DSPLY
C              IF      (PrdStatus = ErrRecLock)
C              WRITE      TRNBREC
C              MOVE      '*GETIN'      ReturnPt      6
*-----*
* If unexpected error occurs, cause inquiry message to be issued. *
*-----*
C              ELSE
C              MOVE      *BLANK      ReturnPt
C              ENDIF
C              ENDSR      ReturnPt
*-----*
* Error report format. *
*-----*
OPRINT      E      NOTFOUND
O              TRNPRDNO
O              29 'NOT IN PRDMAS FILE'

```

Figure 106 (Part 2 of 2). Example of File Exception Handling

When control is passed to the error subroutine, the following occurs:

- If the error is due to a record lock, then the record is written to a backlog file and control returns to the main part with the next transaction (via *GETIN as the return point).
- If the error is due to some other reason, then blanks are moved to ReturnPt. This will result in the RPG default handler receiving control. The recovery action at that point will depend on the nature of the error.

Note that the check for a record lock error is done by matching the *STATUS sub-field 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 238.
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 107 on page 234 shows an example of a program error subroutine in a 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 107. Example of *PSSR Subroutine in 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 108 on page 235 and Figure 109 on page 236 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 108. 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 109. 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 110 on page 237 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 110. Avoiding a Loop in an Error Subroutine

To create the program and start debugging it, using the source in Figure 110, 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 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 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 111 on page 240 shows the source for the procedure RPGHDLR. The procedure defines three procedure parameters: an ILE condition token structure, a field

to contain the name of the procedure which contains the error, and a field to contain the possible actions, resume or percolate. (RPGHDLR does not promote any exceptions).

The basic logic of RPGHDLR is the following:

1. Test to see if it is an out-of-bounds error by testing the message ID
 - If it is, it writes 'Handling...' to QSYSPRT and then sets the action to 'Resume'.
 - If it is not, then it writes out 'Percolating' to QSYSPRT, and then sets the action to 'Percolate'.
2. Return.

```

*=====
* RPGHDLR: RPG exception handling procedure.
* This procedure does the following:
* Handles the exception if it is the RPG
* out of bounds error (RNX0100)
* otherwise
* percolates the exception
* It also prints out what it has done.
*
* Note: This is the exception handling procedure for the
* SHOWERR procedure.
*=====
FQSYSPRT 0 F 132 PRINTER

*-----*
* Procedure parameters
* 1. Input: Condition token structure
* 2. Input: Procedure name in which error occurred
* 3. Output: Code identifying actions to be performed on the
* exception
* 4. Output: New condition if we decide to promote the
* condition. Since this handler only resumes and
* percolates, we will ignore this parameter.
*-----*
D CondTok DS
D MsgSev 4B 0
D MsgNo 2A
D 1A
D MsgPrefix 3A
D 4A

D pPSDS S *
D PassedPSDS DS BASED(pPSDS)
D ProcName 1 10

D Action S 9B 0
*
* Action codes are:
*
D Resume C 10
D Percolate C 20

```

Figure 111 (Part 1 of 2). Source for Condition Handler for Out-of-Bounds Substring Error

```

*-----*
* Parameters                                     *
*-----*
C      *ENTRY      PLIST
C              PARM              CondTok
C              PARM              pPSDS
C              PARM              Action

*-----*
*      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'
C              EXCEPT Handling
C              EVAL    Action    = Resume
C              ELSE
C              EXCEPT Perclating
C              EVAL    Action    = Percolate
C              ENDIF
C              RETURN

*-----*
* Procedure Output                                 *
*-----*
QSYSYSPRT  E
0
0              ProcName
QSYSYSPRT  E              Handling
0              ' Handling...'
QSYSYSPRT  E              Perclating
0              ' Percolating.. .'

```

Figure 111 (Part 2 of 2). Source for Condition Handler for Out-of-Bounds Substring Error

Figure 112 on page 242 shows the source for the procedure SHOWERR, in which the condition handler RPGHDLR is registered.

The procedure parameters include a procedure pointer to RPGHDLR and a pointer to the procedure's program status data structure. In addition, it requires a definition for the error-prone array ARR1, and identification of the parameter lists used by the ILE bindable APIs CEEHDLR and CEEHDLU.

The basic logic of the program is as follows:

1. Register the handler RPGHDLR using the subroutine RegHndlr. This subroutine calls the CEEHDLR API, passing it the procedure pointer to RPGHDLR.
2. Generate an out-of-bounds substring error.

The handler RPGHDLR is automatically called. It handles the exception, and indicates that processing should resumes in the next *machine* instruction following the error. Note that the next machine instruction may not be at the beginning of the next RPG operation.

3. Generate an out-of-bounds array error.

Again, RPGHDLR is automatically called. However, this time it cannot handle the exception, and so it percolates it to the next exception handler associated with the procedure, namely, the *PSSR error subroutine.

The *PSSR cancels the procedure.

4. Unregister the condition handler RPGHDLR via a call to CEEHDLU.
5. Return

As with the RPGHDLR procedure, SHOWERR writes to QSYSVRT to show what is occurring as it is processed.

```
*=====
* SHOWERR:      Show exception handling using a user-defined  *
*               exception handler.                            *
*=====
QSYSVRT  0  F  132  PRINTER

*-----*
* The following are the parameter definitions for the CEEHDLR *
* API. The first is the procedure pointer to the             *
* procedure which will handle the exception. The second     *
* is a pointer to a communication area which will be passed  *
* to the exception handling procedure. In this example,     *
* we will pass a pointer to the name of this procedure      *
*-----*
D pConHdlr      S          *  PROCPTR
D               INZ(%paddr('RPGHDLR'))

*-----*
* PSDS
*-----*
D DSPsds        SDS          NOOPT
D ProcName      *PROC

*-----*
* Array that will be used to cause an error
*-----*
D Arr1          S          10A  DIM(5)

*-----*
* CEEHDLR Interface
*-----*
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 112 (Part 1 of 3). Source for Registering a Condition Handler

```

*-----*
* Register the handler and generate errors *
*-----*
C          EXSR      RegHndl r

*-----*
*      Generate a substring error *
*-----*
C          Z-ADD    -1      Neg1      5 0
C  Neg1    SUBST    'Hello'  Examp    10

*-----*
*      The exception was handled by the handler and control
*      resumes here. *
*-----*
C          EXCEPT  ImBack

*-----*
*      Generate an array out of bounds error *
*-----*
C          MOVE     Arr1(Neg1)  Arr1(Neg1)

*-----*
*      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      DeRegHndl r
C
C          SETON                      LR

*=====*
* RegHndl r - Call the API to register the Handler *
*=====*
C  RegHndl r  BEGSR
C                      CALLP      CEEHDLR(pConHdl r : %ADDR(DSPsds) : *OMIT)

C                      ENDSR

*=====*
* DeRegHndl r - Call the API to unregister the Handler *
*=====*
C  DeRegHndl r  BEGSR
C                      CALLP      CEEHDLU(pConHdl r : *OMIT)
C                      ENDSR

```

Figure 112 (Part 2 of 3). Source for Registering a Condition Handler

```

*=====
* *PSSR: Error Subroutine for the procedure
*=====
C      *PSSR      BEGSR

C      EXCEPT  InPssr
C      EXCEPT  Cancellng

C      ENDSR     '*CANCL'

*=====
* Procedure Output
*=====
OQSYSVRT  E      ImBack
0
0          'I'm Back'
OQSYSVRT  E      InPssr
0
0          'In PSSR'
OQSYSVRT  E      Cancellng
0
0          'Cancellng...'

```

Figure 112 (Part 3 of 3). Source for Registering a Condition Handler

If you want to try these procedures, follow these steps:

1. To create the procedure RPGHDLR, using the source shown in Figure 111 on page 240, type:

```
CRTRPGMOD MODULE(MYLIB/RPGHDLR)
```

2. To create the procedure SHOWERR, using the source shown in Figure 112 on page 242, 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(QSYSVRT) SHARE(*YES)
CALL PGM(MYLIB/ERRORTEST)
```

The output is shown below:

```

In Handler for SHOWERR
Handling...
I'm Back
In Handler for SHOWERR
Percolating...
In PSSR
Cancellng...

```

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. See the *System API Reference* for more information on these ILE bindable APIs.

Figure 113 shows an example of enabling and coding a cancel handler for a subprocedure. (Cancel handlers can also be enabled for 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

```

Figure 113 (Part 1 of 4). Enabling and Coding a Cancel Handler for a Subprocedure

```

-----
* 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 113 (Part 2 of 4). 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

```

Figure 113 (Part 3 of 4). Enabling and Coding a Cancel Handler for a Subprocedure


```

*-----
* 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

```

Figure 113 (Part 4 of 4). Enabling and Coding a Cancel Handler for a Subprocedure

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 114. 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 115 on page 249 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 115. ILE Condition Handler that Ignores CPF4906

Figure 116 on page 250 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 116. How to Ignore Status and Notify Messages

Chapter 13. 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 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.

Note: To obtain a dump of variable data, the program object must have debug data. That is, it must be created with any debug view except *NONE. If no debug data is available, then the dump will still contain the PSDS and file information.

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:

- The DUMP operation runs (is active) only if keyword DEBUG(*YES) is specified on the control specification. If the keyword is not specified, or if DEBUG(*NO) is specified, the DUMP operation is checked for errors and the statement is printed on the listing, but the DUMP is not processed.
- If the DUMP operation is conditioned, it occurs only if the condition is met.

- If a DUMP operation is bypassed by a GOTO operation, the DUMP operation does not occur.

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 211). 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	← A
Program Name	TEST	
Library	MYLIB	
Module Name	DBGEX2	← B C D E
Program Status	00202	
Previous Status	00000	
Statement in Error	00000088	
RPG Routine	RPGPGM	← F
Number of Parameters		
Message Type	MCH	
Additional Message Info	4431	
Message Data		
Program signature violation.		
Status that caused RNX9001		← G
Last File Used		
Last File Status		
Last File Operation		
Last File Routine		
Last File Statement		← H
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*	
Compile Date	123095	
Compile Time	153438	
Compiler Level	0001	
Source File	QRPGLSRC	←
Library	MYLIB	
Member	DBGEX2	

Figure 117. 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

```

```

PRINTER FEEDBACK:
Current Line Number. . . . . : 1
Current Page . . . . . : 1
Major Return Code. . . . . : 00
Minor Return Code. . . . . : 00

```

```

Output Buffer:
0000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 *
0020 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 *
0040 00000000 00000000 00000000 00000000 00000000 *

```

Figure 118. Feedback Areas section of Formatted Dump

I This is the file feedback section of the INFDS. Only fields applicable to the file type are printed. The rest of the INFDS feedback sections are not dumped, since they are only updated if they have been declared in the program.

- J** This is the file open feedback information for the file. See the *Data Management* manual for a description of the fields.
- K** This is the common I/O feedback information for the file. See the *Data Management* manual for a description of the fields.

Information with Full-Formatted Dump

Open Data Path:									
0000	64800000	00001AF0	00001B00	000000B0	00000140	000001C6	00000280	000002C0	* 0 F *
0020	00000530	00000000	00000000	00000380	00000000	06000000	00000000	00000000	* *
0040	00008000	00000000	003AC02B	A00119FF	000006C0	00003033	00000000	00000000	* *
0060	80000000	00000000	003AC005	CF001CB0	00000000	00000000	00000000	00000000	* *
0080	80000000	00000000	003AA024	D0060120	01900000	00010000	00000050	00000000	* & *
00A0	1F000000	00000000	00000000	00000000	E2D7D8E2	E8E2D7D9	E3404040	D8E2E8E2	* SPQSYSVRT QSYS*
00C0	40404040	404008F0	F4F0F7F9	D5F0F0F2					* Q04079N002QSPL & *
Open Feedback:									
0000	E2D7D8E2	E8E2D7D9	E3404040	D8E2E8E2	40404040	4040D8F0	F4F0F7F9	D5F0F0F2	*SPQSYSVRT 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 119. 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
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'

```

NAME	ATTRIBUTES	VALUE	
_QRNU_DSI_DSI1	INT(10)	1	'00000001'X
_QRNU_DSI_DSI2	INT(10)	2	'00000002'X
_QRNU_NULL_ARR	CHAR(1)	DIM(8)	
	(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
ARR	CHAR(2)	DIM(8)	
	(1-3)	'AB'	'C1C2'X
	(4-7)	' '	'4040'X
	(8)	'1'	'F1'X
ARRAY	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)	'9999-12-31-12.00.000000'	
	VALUE IN HEX	'F9F9F9F960F1F260F3F160F1F24BF0F04BF0F04BF0F0F0F0F0'X	
BIN4D3	BIN(4,3)	-4.321	'EF1F'X
BIN9D7	BIN(9,7)	98.7654321	'3ADE68B1'X
DBCSSTRING	GRAPHIC(3)	' BBCCDD '	'C2C2C3C3C4C4'X

Figure 120 (Part 1 of 2). 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)	'aaaaaaaa'		'8181818181818181'X
	(2)	'bbbbbbbbbb'		'8282828282828282'X
DS3	DS		T	
FIRSTNAME	CHAR(10)	'Fred'		'C69985844040404040'X
LASTNAME	CHAR(10)	'Jones'		'D1969585A240404040'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		
PACKED100	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		
***** END OF R P G D U M P *****				

Figure 120 (Part 2 of 2). Data section of Formatted Dump

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.

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 ILE program containing one or more procedures written in RPG IV.

Chapter 14. Defining Files

Files serve as the connecting link between a program and the device used for I/O. Each file on the system has an associated file description which describes the file characteristics and how the data associated with the file is organized into records and fields.

In order for a program to perform any I/O operations, it must identify the file description(s) the program is referencing, what type of I/O device is being used, and how the data is organized. This chapter provides general information on:

- Associating file descriptions with input/output devices
- Defining externally described files
- Defining program-described files
- Data management operations

Information on how to use externally and program-described files with different device types is found in subsequent chapters.

Associating Files with Input/Output Devices

The key element for all I/O operations on the AS/400 is the file. The system supports the following file types:

database files

allow storage of data permanently on system

device files

allow access to externally attached devices. Include display files, printer files, tape files, diskette files, and ICF files.

save files

used to store saved data on disk

DDM files

allow access to data files stored on remote systems.

Each I/O device has a corresponding file description of one of the above types which the program uses to access that device. The actual device association is made when the file is processed: the data is read from or written to the device when the file is used for processing.

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 121 on page 262 shows a file description specification for a display (WORKSTN) file FILEX.

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... *
FFilename++IPEASFR1en+LK1en+AIDevice+.Keywords+++++
FFILEX      CF  E                WORKSTN
```

Figure 121. Identifying a Display File in an RPG Program

Note that it is the file name, not the device name (specified in positions 36 through 42) which points to the OS/400 file description that contains the specifications for the actual device.

The RPG device types correspond to the above file types as follows:

RPG Device Type	AS/400 File Type
DISK	database, save, DDM files
PRINTER	printer files
WORKSTN	display, ICF files
SEQ	tape, diskette, save, printer, database
SPECIAL	N/A

Figure 122 illustrates the association of the RPG file name FILEX, as coded in Figure 121, with a system file description for a display file.

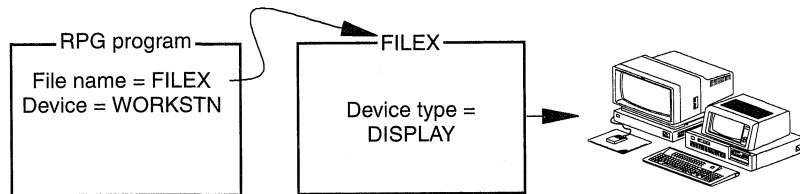


Figure 122. Associating a file name with a display file description

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 123 on page 263 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.

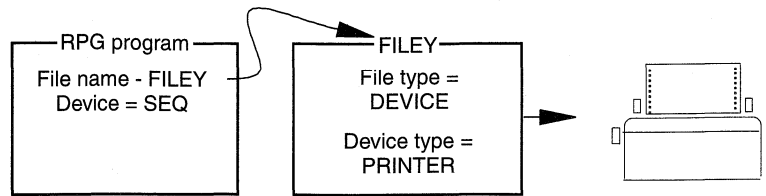


Figure 123. Associating a file name with a display file description

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 273.

Naming Files

On the AS/400 system, files are made up of members. These files are organized into libraries. The convention for naming files is library-name/file-name.

In an ILE RPG program, file names are identified in positions 7 through 16 in file description specifications. File names can be up to ten characters long and must be unique.

You do not qualify the file name with a library within a program. At run time, the system searches the library list associated with your job to find the file. If you wish to change the name, member, or specify a particular library, you can use a file override command. See “Overriding and Redirecting File Input and Output” on page 273 for more information on file overrides.

Types of File Descriptions

When identifying the file description your program will be using, you must indicate whether it is a program-described file or an externally described file.

- For a **program-described file**, the description of the fields are coded within the RPG source member on input and/or output specifications.

The description of the file to the operating system includes information about where the data comes from and the length of the records in the file.

- For an **externally described file**, the compiler retrieves the description of the fields from an external file-description which was created using DDS, IDDU, or SQL 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 270 for further information.

If an externally described file (identified by an E in position 22 of the file description specification) is specified for the devices SEQ or SPECIAL, the RPG program uses the field descriptions for the file, but the interface to the operating system is as though the file were a program-described file. Externally described files cannot specify device-dependent functions such as forms control for PRINTER files because this information is already defined in the external description.

Using Files with External-Description as Program-Described

A file created from external descriptions can be used as a program-described file in the program. To use an externally described file as a program-described file,

1. Specify the file as program-described (F in position 22) in the file description specification of the program.
2. Describe the fields in the records on the input or/and output specifications of the program.

At compile time, the compiler uses the field descriptions in the input or/and output specifications. It does not retrieve the external descriptions.

Example of Some Typical Relationships between Programs and Files

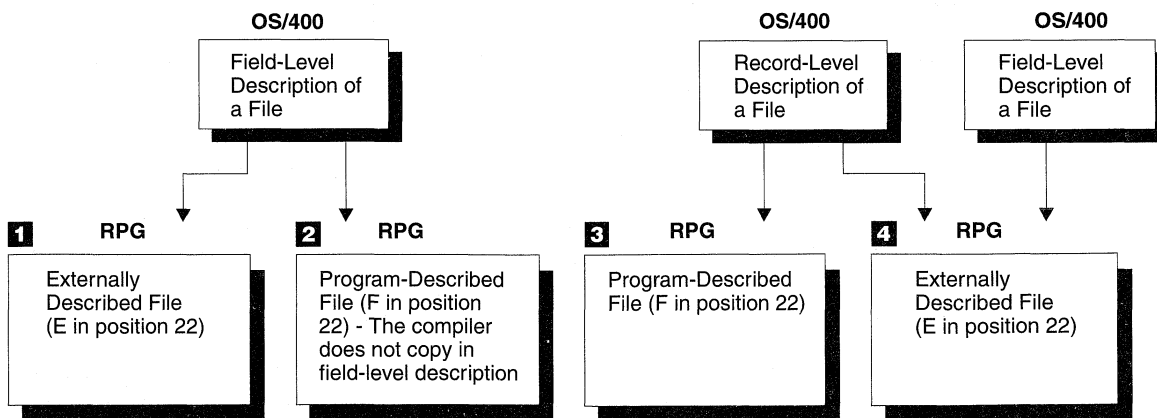


Figure 124. Typical Relationships between an RPG Program and Files on the AS/400 System

- 1** The program uses the field-level description of a file that is defined to the operating system. An externally described file is identified by an E in position 22 of the file description specifications. At compilation time, the compiler copies in the external field-level description.

- 2** An externally described file (that is, a file with field-level external description) is used as a program-described file in the program. A program-described file is identified by an F in position 22 of the file description specifications. This entry tells the compiler not to copy in the external field-level descriptions. This file does not have to exist at compilation time.
- 3** A file is described only at the record level to the operating system. The fields in the record are described within the program; therefore, position 22 of the file description specifications must contain an F. This file does not have to exist at compilation time.
- 4** A file name can be specified at compilation time (that is, coded in the RPG source member), and a different file name can be specified at run time. The E in position 22 of the file description specifications indicates that the external description of the file is to be copied in at compilation time. At run time, a file override command can be used so that a different file is accessed by the program. To override a file at run time, you must make sure that record names in both files are the same. The RPG program uses the record-format name on the input/output operations, such as a READ operation where it specifies what record type is expected. See “Overriding and Redirecting File Input and Output” on page 273 for more information.

Defining Externally Described Files

You can use DDS to describe files to the OS/400 system. Each record type in the file is identified by a unique record-format name.

An E entry in position 22 of the file description specifications identifies an externally described file. The E entry indicates to the compiler that it is to retrieve the external description of the file from the system when the program is compiled.

The information in this external description includes:

- File information, such as file type, and file attributes, such as access method (by key or relative record number)
- Record-format description, which includes the record format name and field descriptions (names, locations, and attributes).

The information the compiler retrieves from the external description is printed on the compiler listing as long as `OPTION(*EXPDDS)` is specified on either the `CRTRPGMOD` or `CRTBNDRPG` command when compiling the source member. (The default for both of these commands is `OPTION(*EXPDDS)`.)

The following section describes how to use a file description specification to rename or ignore record formats and how to use input and output specifications to modify external descriptions. Remember that input and output specifications for externally described files are optional.

Renaming Record-Format Names

Many of the functions that you can specify for externally described files (such as the CHAIN operation) operate on either a file name or a record-format name. Consequently, each file and record-format name in the program must be a unique symbolic name.

To rename a record-format name, use the RENAME keyword on the file description specifications for the externally described file as shown in Figure 125. The format is RENAME(*old name:new name*).

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... *
FFilename++IPEASFRlen+LKlen+AIDevice+.Keywords++++++++++++++++++++
FITMMSTL  IP  E          K DISK  RENAME(ITEMFORMAT:MSTITM)
*
```

Figure 125. 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 125, the record format ITEMFORMAT in the externally described file ITMMSTL is renamed MSTITM for use in this 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 126 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 126. 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 127 on page 267.

The file must have more than one record format, and not all of them can be ignored; at least one must remain. Except for that requirement, any number of record formats can be ignored for a file.

Once a record-format is ignored, it cannot be specified for any other keyword (SFILE, RENAME, or INCLUDE), or for another IGNORE.

Ignored record-format names appear on the cross-reference listing, but they are flagged as ignored.

To indicate that a record format from an externally described file, is to be ignored, enter the keyword and parameter IGNORE(*record-format name*) on the file description specification in the Keyword field.

Alternatively, the INCLUDE keyword can be used to include only those record format names that are to be used in a program. All other record formats contained in the file will be excluded.

```
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... *
FFilename++IPEASFRlen+LKlen+AIDevice+.Keywords+++++
*
* Assume the file ITMMSTL contains the following record formats:
* EMPLNO, NAME, ADDR, TEL, WAGE. To make the program run as if only the
* EMPLNO and NAME records existed, either of the following two methods
* can be used:
*
FITMMSTL  UF  E          K DISK  IGNORE(ADDR:TEL:WAGE)
*
* OR:
*
FITMMSTL  UF  E          K DISK  INCLUDE(EMPLNO:NAME)
*
```

Figure 127. IGNORE Keyword for Record Formats in an Externally Described File

Using Input Specifications to Modify an External Description

You can use the input specifications to override certain information in the external description of an input file or to add RPG functions to the external description. On the input specifications, you can:

- Assign record-identifying indicators to record formats as shown in Figure 128 on page 268.
- Rename a field as shown in Figure 128 on page 268.
- Assign control-level indicators to fields as shown in Figure 128 on page 268.
- Assign match-field values to fields for matching record processing as shown in Figure 129 on page 268.
- Assign field indicators as shown in Figure 129 on page 268.

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 128. 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 129. Adding RPG Functions to an External Description

- 1 To assign a match value to a field in an externally described record, specify the record-format name in positions 7 through 16 of the record-identification line. On the field-description line specify the name of

the field in positions 49 through 62 and assign a match-level value in positions 65 and 66.

In this example, the CUSTNO field in both records MSTREC and WKREC is assigned the match-level value M1.

- 2** To assign a field indicator to a field in an externally described record, specify the record-format name in positions 7 through 16 of the record-identification line. On the field-description line, specify the field name in positions 49 through 62, and specify an indicator in positions 69 through 74.

In this example, the field BALDUE in the record WKREC is tested for zero when it is read into the program. If the field's value is zero, indicator 98 is set on.

Using Output Specifications

Output specifications are optional for an externally described file. RPG supports file operation codes such as WRITE and UPDATE that use the external record-format description to describe the output record without requiring output specifications for the externally described file.

You can use output specification to control when the data is to be written, or to specify selective fields that are to be written. The valid entries for the field-description line for an externally described file are output indicators (positions 21 - 29), field name (positions 30 - 43), and blank after (position 45). Edit words and edit codes for fields written to an externally described file are specified in the DDS for the file. Device-dependent functions such as fetch overflow (position 18) or space/skip (positions 40 - 51) are not valid in an RPG program for externally described files. The overflow indicator is not valid for externally described files either. For a description of how to specify editing in the DDS, see the *DDS Reference*.

If output specifications are used for an externally described file, the record-format name is specified in positions 7 - 16 instead of the file name.

If all the fields in an externally described file are to be placed in the output record, enter *ALL in positions 30 through 43 of the field-description line. If *ALL is specified, you cannot specify other field description lines for that record.

If you want to place only certain fields in the output record, enter the field name in positions 30 through 43. The field names you specify in these positions must be the field names defined in the external record description, unless the field was renamed on the input specifications. See Figure 130 on page 270.

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 ...+... *
OFile name++DF..N01N02N03Excnam++++B++A++Sb+Sa+.....*
OITMREC   D   20
0.....N01N02N03Field++++++YB.End++PConstant/editword/DTformat++
0
*
*ALL 1
*
OSLSREC   D   30
0
0          SLSNAM 2
0          COMRAT
0          15     BONUS
*

```

Figure 130. 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 Chapter 12, "Handling Exceptions" on page 217.

The RPG program does not provide level checking for program-described files or for files using the devices SEQ or SPECIAL.

For more information on how to specify level checking, see the *Data Management* manual.

Defining Program-Described Files

Program-described files are files whose records and fields are described on input/output specifications in the program that uses the file. To use a program-described file in an RPG program you must:

1. Identify the file(s) in the file description specifications.
2. If it is an input file, describe the record and fields in the input specifications. The file name in positions 7 through 16 in the input specifications must be the same as the corresponding name entered in the file specifications.

On the record-identification entries you indicate whether you want to perform sequence checking of records within the file.

3. Enter the same file name as in step 1 in the FACTOR 2 field of those calculation specifications which require it. For example, WRITE operations to a program-described file require a data structure name in the result field.
4. If it is an output file, describe the record and fields in the output specifications. In addition, you specify how the output is to be printed. The file name in positions 7 through 16 in the output specifications must be the same as the corresponding name entered in the file specifications.

A program-described file must exist on the system, and be in your library list, before the program can run. To create a file, use one of the Create File commands, which can be found in the *CL Reference*.

Data Management Operations and ILE RPG 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 15 on page 272 shows the data management operations provided by the AS/400 and their corresponding ILE RPG operation. It also shows which operations are allowed for which ILE RPG device type.

Table 15. 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
WRITE	
By relative record number	WRITE
By key	WRITE, EXCEPT, primary and secondary file
Sequential	WRITE, EXCEPT
FEOD	FEOD
UPDATE	
By relative record number	UPDATE, primary and secondary file
By key	UPDATE, primary and secondary file
DELETE	
By relative record number	DELETE, primary and secondary file
By key	DELETE, primary and secondary file
ACQUIRE	ACQ
RELEASE	REL
COMMIT	COMMIT
ROLLBACK	ROLBK
CLOSE	CLOSE, LR RETURN

Chapter 15. General File Considerations

This chapter provides information on the following aspects of file processing on the AS/400 using RPG:

- overriding and redirecting file input and output
- file locking by an RPG program
- record locking by an RPG program
- sharing an open data path
- AS/400 spooling functions
- using SRTSEQ/ALTSEQ in an RPG program versus a DDS file

Overriding and Redirecting File Input and Output

OS/400 commands can be used to override a parameter in the specified file description or to redirect a file at compilation time or run time. File redirection allows you to specify a file at run time to replace the file specified in the program (at compilation time):

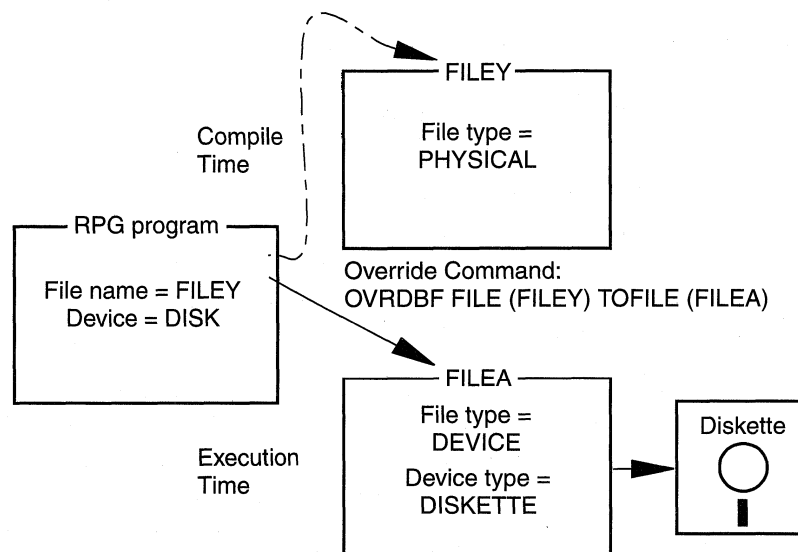


Figure 131. Overriding File Input and Output Example

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.

Not all file redirections or overrides are valid. At run time, checking ensures that the specifications within the RPG program are valid for the file being processed. The OS/400 system allows some file redirections even if device specifics are contained

in the program. For example, if the RPG device name is PRINTER, and the actual file the program connects to is not a printer, the OS/400 system ignores the RPG print spacing and skipping specifications. There are other file redirections that the OS/400 system does not allow and that cause the program to end. For example, if the RPG device name is WORKSTN and the EXFMT operation is specified in the program, the program is stopped if the actual file the program connects to is not a display or ICF file. In ILE, overrides are scoped to the activation group level, job level, or call level. Overrides that are scoped to the activation group level remain in effect until they are deleted, replaced, or until the activation group in which they are specified ends. Overrides that are scoped to the job level remain in effect until they are deleted, replaced, or until the job in which they are specified ends. This is true regardless of the activation group in which the overrides were specified. Overrides that are scoped to the call level remain in effect until they are deleted, replaced, or until the program or procedure in which they are specified ends.

The default scope for overrides is the activation group. For job-level scope, specify `OVRSCOPE(*JOB)` on the override command. For call-level scope, specify `OVRSCOPE(*CALLLVL)` on the override command.

See the *Data Management* manual for more detailed information on valid file redirections and file overrides. *ILE Concepts* also contains information about overrides and activation group vs. job level scope.

Example of Redirecting File Input and Output

The following example shows the use of a file override at compilation time. Assume that you want to use an externally described file for a TAPE device which does not have field-level description. You must:

1. Define a physical file named FMT1 with one record format that contains the description of each field in the record format. The record format is defined on the data description specifications (DDS). For a tape device, the externally described file should contain only one record format.
2. Create the file named FMT1 with a Create Physical File CL command.
3. Specify the file name of QTAPE (which is the IBM-supplied device file name for magnetic tape devices) in the RPG program. This identifies the file as externally described (indicated by an E in position 22 of the file description specifications), and specifies the device name SEQ in positions 36 through 42.
4. Use an override command—`OVRDBF FILE(QTAPE) TOFILE(FMT1)`—at compilation time to override the QTAPE file name and use the FMT1 file name. This command causes the compiler to copy in the external description of the FMT1 file, which describes the record format to the RPG compiler.
5. Create the RPG program using the CRTBNDRPG command or the CRTPGM command.
6. Call the program at run time. The override to file FMT1 should not be in effect while the program is running. If the override is in effect, use the CL command `DLTOVR` (Delete Override) before calling the program.

Note: You may need to use the CL command `OVRTAPF` before you call the program to provide information necessary for opening the tape file.

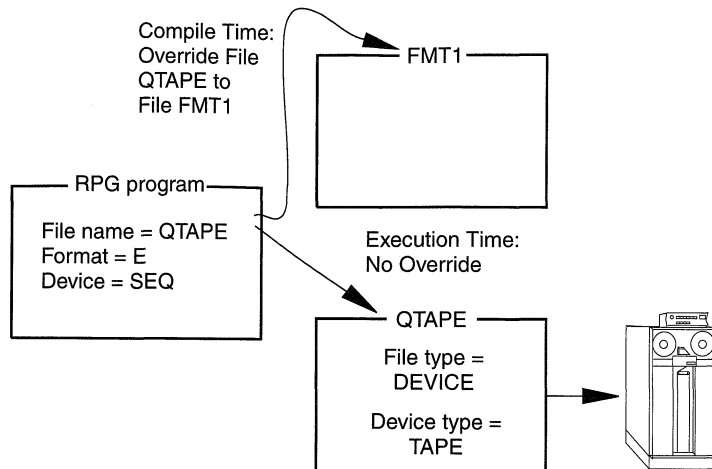


Figure 132. Redirecting File Input and Output Example

File Locking

The OS/400 system allows a lock state (exclusive, exclusive allow read, shared for update, shared no update, or shared for read) to be placed on a file used during the execution of a job. Programs within a job are not affected by file lock states. A file lock state applies only when a program in another job tries to use the file concurrently. The file lock state can be allocated with the CL command ALCOBJ (Allocate Object). For more information on allocating resources and lock states, see the *Data Management* manual.

The OS/400 system places the following lock states on database files when it opens the files:

File Type	Lock State
Input	Shared for read
Update	Shared for update
Add	Shared for update
Output	Shared for update

The shared-for-read lock state allows another user to open the file with a lock state of shared for read, shared for update, shared no update, or exclusive allow read, but the user cannot specify the exclusive use of the file. The shared-for-update lock state allows another user to open the file with shared-for-read or shared-for-update lock state.

The RPG program places an exclusive-allow-read lock state on device files. Another user can open the file with a shared-for-read lock state.

The lock state placed on the file by the RPG program can be changed if you use the Allocate Object command.

Record Locking

When a record is read by a program, it is read in one of two modes: input or update. If a program reads a record for update, a lock is placed on that record. Another program cannot read the same record for update until the first program releases that lock. If a program reads a record for input, no lock is placed on the record. A record that is locked by one program can be read for input by another program.

In RPG IV programs, you use an update file to read records for update. A record read from a file with a type other than update can be read for inquiry only. By default, any record that is read from an update file will be read for update. For update files, you can specify that a record be read for input by using one of the input operations CHAIN, READ, READE, READP, or READPE and specifying an operation code extender (N) in the operation code field following the operation code name.

When a record is locked by an RPG IV program, that lock remains until one of the following occurs:

- the record is updated.
- the record is deleted.
- another record is read from the file (either for inquiry or update).
- a SETLL or SETGT operation is performed against the file
- an UNLOCK operation is performed against the file.
- an output operation defined by an output specification with no field names included is performed against the file.

Note: An output operation that adds a record to a file does not result in a record lock being released.

If your program reads a record for update and that record is locked through another program in your job or through another job, your read operation will wait until the record is unlocked (except in the case of shared files, see “Sharing an Open Data Path” on page 277). 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 (RNQ1218) 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 307 for more information.)

Sharing an Open Data Path

An open data path is the path through which all input and output operations for a file are performed. Usually a separate open data path is defined each time a file is opened. If you specify SHARE(*YES) for the file creation or on an override, the first program's open data path for the file is shared by subsequent programs that open the file concurrently.

The position of the current record is kept in the open data path for all programs using the file. If you read a record in one program and then read a record in a called program, the record retrieved by the second read depends on whether the open data path is shared. If the open data path is shared, the position of the current record in the called program is determined by the current position in the calling program. If the open data path is not shared, each program has an independent position for the current record.

If your program holds a record lock in a shared file and then calls a second program that reads the shared file for update, you can release the first program's lock by :

- performing a READ operation on the update file by the second program, or
- using the UNLOCK or the read-no-lock operations.

In ILE, shared files are scoped to either the job level or the activation group level. Shared files that are scoped to the job level can be shared by any programs running in *any* activation group within the job. Shared files that are scoped to the activation group level can be shared *only* by the programs running in the same activation group.

The default scope for shared files is the activation group. For job-level scope, specify OVRSCOPE(*JOB) on the override command.

ILE RPG offers several enhancements in the area of shared ODPs. If a program or procedure performs a read operation, another program or procedure can update the record as long as SHARE(*YES) is specified for the file in question. In addition, when using multiple-device files, if one program acquires a device, any other program sharing the ODP can also use the acquired device. It is up to the programmer to ensure that all data required to perform the update is available to the called program.

Sharing an open data path improves performance because the OS/400 system does not have to create a new open data path. However, sharing an open data path can cause problems. For example, an error is signaled in the following cases:

- If a program sharing an open data path attempts file operations other than those specified by the first open (for example, attempting input operations although the first open specified only output operations)

- If a program sharing an open data path for an externally described file tries to use a record format that the first program ignored
- If a program sharing an open data path for a program described file specifies a record length that exceeds the length established by the first open.

When several files in one program are overridden to one shared file at run time, the file opening order is important. In order to control the file opening order, you should use a programmer-controlled open or use a CL program to open the files before calling the program.

If a program shares the open data path for a primary or secondary file, the program must process the detail calculations for the record being processed before calling another program that shares that open data path. Otherwise, if lookahead is used or if the call is at total time, sharing the open data path for a primary or secondary file may cause the called program to read data from the wrong record in the file.

You must make sure that when the shared file is opened for the first time, all of the open options that are required for subsequent opens of the file are specified. If the open options specified for subsequent opens of a shared file are not included in those specified for the first open of a shared file, an error message is sent to the program.

Table 16 details the system open options allowed for each of the open options you can specify.

<i>Table 16. System Open Options Allowed with User Open Options</i>	
RPG User Open Options	System Open Options
INPUT	INPUT
OUTPUT	OUTPUT (program created file)
UPDATE	INPUT, UPDATE, DELETE
ADD	OUTPUT (existing file)

For additional information about sharing an open data path, see the *DB2 for AS/400 Database Programming* manual. *ILE Concepts* also contains information about sharing open data paths and activation group versus job level scope.

Spooling

Spooling is a system function that puts data into a storage area to wait for processing. The AS/400 system provides for the use of input and output spooling functions. Each AS/400 file description contains a spool attribute that determines whether spooling is used for the file at run time. The RPG program is not aware that spooling is being used. The actual physical device from which a file is read or to which a file is written is determined by the spool reader or the spool writer. For more detailed information on spooling, see the *Data Management* manual.

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:

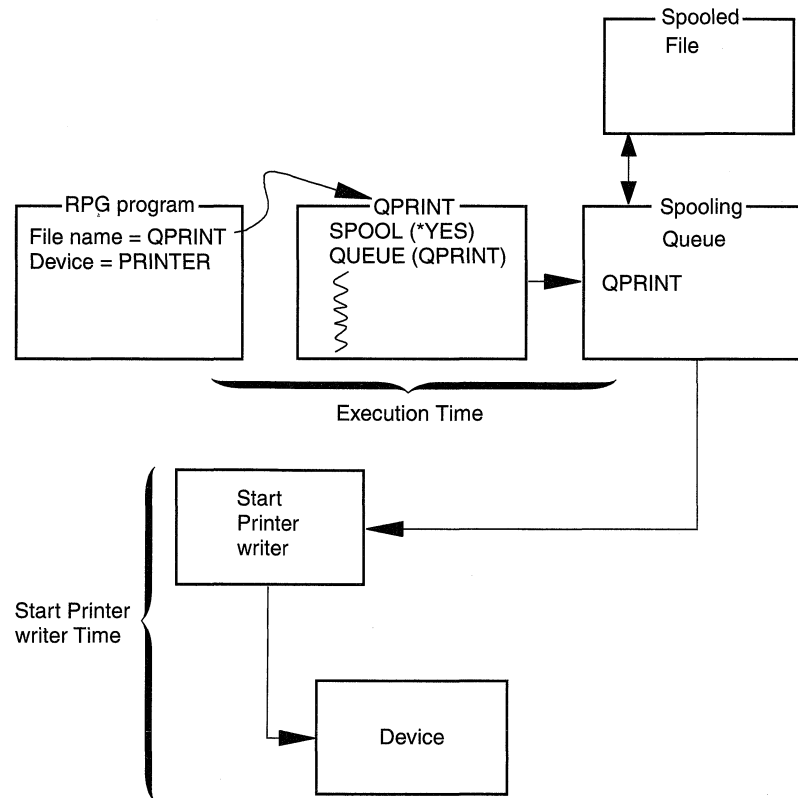


Figure 133. Output Spooling Example

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, AS/400 spooling support allows you to redirect a file after the program has run. You can direct the same printed output to a different device such as a diskette.

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.

Chapter 16. 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 AS/400. They can be either physical or logical files and either externally described or program-described. You access database files by associating the file name with the device DISK in positions 36 through 42 of the file description specifications.

Database files can be created by OS/400 Create File commands. For more information on describing and creating database files, refer to the *DB2 for AS/400 Database Programming* manual and the *DDS Reference*.

Physical Files and Logical Files

Physical files contain the actual data that is stored on the system, and a description of how data is to be presented to or received from a program. They contain only one record format, and one or more members. Records in database files can be externally or program-described.

A physical file can have a keyed sequence access path. This means that data is presented to a program in a sequence based on one or more key fields in the file.

Logical files do not contain data. They contain a description of records found in one or more physical files. A logical file is a view or representation of one or more physical files. Logical files that contain more than one format are referred to as **multi-format** logical files.

If your program processes a logical file which contains more than one record format, you can use a read by record format to set the format you wish to use.

Data Files and Source Files

A **data file** contains actual data, or a view of the data. Records in data files are grouped into members. All the records in a file can be in one member or they can be grouped into different members. Most database commands and operations by default assume that database files which contain data have *only one* member. This means that when your program accesses database files containing data, you do not need to specify the member name for the file unless your file contains more than one member. If your file contains more than one member and a particular member is not specified, the first member is used.

Usually, database files that contain source programs are made up of more than one member. Organizing source programs into members within database files allows you to better manage your programs. The **source member** contains source statements that the system uses to create program objects.

Using Externally Described Disk Files

Externally described DISK files are identified by an E in position 22 of the file description specifications. The E indicates that the compiler is to retrieve the external description of the file from the system when the program is compiled. Therefore, you must create the file before the program is compiled.

The external description for a DISK file includes:

- The record-format specifications that contain a description of the fields in a record
- Access path specifications that describe how the records are to be retrieved.

These specifications result from the DDS for the file and the OS/400 create file command that is used for the file.

Record Format Specifications

The record-format specifications allow you to describe the fields in a record and the location of the fields in a record. The fields are located in the record in the order specified in the DDS. The field description generally includes the field name, the field type, and the field length (including the number of decimal positions in a numeric field). Instead of specifying the field attributes in the record format for a physical or logical file, you can define them in a field-reference file.

In addition, the DDS keywords can be used to:

- Specify that duplicate key values are not allowed for the file (UNIQUE)
- Specify a text description for a record format or a field (TEXT).

For a complete list of the DDS keywords that are valid for a database file, see the *DB2 for AS/400 Database Programming*.

Figure 134 on page 283 shows an example of the DDS for a database file, and Figure 135 on page 284 for a field-reference file that defines the attributes for the fields used in the database file. See the *DDS Reference* for more information on a field-reference file.

Access Path

The description of an externally described file contains the access path that describes how records are to be retrieved from the file. Records can be retrieved based on an arrival sequence (non-keyed) access path or on a keyed-sequence access path.

The arrival sequence access path is based on the order in which the records are stored in the file. Records are added to the file one after another.

For the keyed-sequence access path, the sequence of records in the file is based on the contents of the key field that is defined in the DDS for the file. For example, in the DDS shown in Figure 134 on page 283, CUST is defined as the key field. The keyed-sequence access path is updated whenever records are added, deleted, or when the contents of a key field change.

For a complete description of the access paths for an externally described database file, see the *DB2 for AS/400 Database Programming* manual.

```

*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ..*
A.....T.Name+++++.Len++TDpB.....Functions+++++*****
A** LOGICAL CUSMSTL  CUSTOMER MASTER FILE
A
A          R CUSREC          UNIQUE
A          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 134. 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 135 on page 284.

```

*.. 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=0th')
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 135. 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 134 on page 283. 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. (However, variable-length fields cannot be used as key fields in an RPG program.) All record types in a file do not have to have the same key fields. For example, an order header record can have the ORDER field defined as the key field, and the order detail records can have the ORDER and LINE fields defined as the key fields.

The key for a file is determined by the valid keys for the record types in that file. The file's key is determined in the following manner:

- If all record types in a file have the same number of key fields defined in the DDS that are identical in attributes, the *key for the file* consists of all fields in the key for the record types. (The corresponding fields do not have to have the same name.) For example, if the file has three record types and the key for each record type consists of fields A, B, and C, the file's key consists of fields A, B, and C. That is, the file's key is the same as the records' key.
- If all record types in the file do not have the same key fields, the key for the file consists of the key fields *common* to all record types. For example, a file has three record types and the key fields are defined as follows:
 - REC1 contains key field A.
 - REC2 contains key fields A and B.
 - REC3 contains key fields A, B, and C.

The file's key is field A—the key field common to all record types.

- If no key field is common to all record types, there is no key for the file.

In an ILE RPG 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. The search argument cannot contain a floating point, variable length, or null-capable field.

For an operation to a record name, the maximum number of fields that you can specify in a search argument is equal to the total number of key fields valid for that record type.

If the search argument consists of one field, you can specify a literal, a field name, or a KLIST name with one KFLD. If the search argument is composed of more than one field (a composite key), you must specify a KLIST with multiple KFLDs. To process null-valued keys a KLIST must be used.

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 433.

In all these file operations (CHAIN, DELETE, READE, READPE, SETGT, and SETLL), you can also specify a search argument that contains fewer than the total number of fields valid for the file or record. Such a search argument refers to a partial key.

Referring to a Partial Key

The rules for the specification of a search argument that refers to a partial key are as follows:

- The search argument is composed of fields that correspond to the leftmost (high-order) fields of the key for the file or record.
- Only the rightmost fields can be omitted from the key list (KLIST) for a search argument that refers to a partial key. For example, if the total key for a file or record is composed of key fields A, B, and C, the valid search arguments that refer to a partial key are field A, and fields A and B.
- Each field in the search argument must be identical in attributes to the corresponding key field in the file or record. The attributes include the length, data type, the number of decimal positions, and format (for example, packed or zoned).

- A search argument cannot refer to a portion of a key field.

If a search argument refers to a partial key, the file is positioned at the first record that satisfies the search argument or the record retrieved is the first record that satisfies the search argument. For example, the SETGT and SETLL operations position the file at the first record on the access path that satisfies the operation and the search argument. The CHAIN operation retrieves the first record on the access path that satisfies the search argument. The DELETE operation deletes the first record on the access path that satisfies the search argument. The READE operation retrieves the next record if the portion of the key of that record (or the record of the specified type) on the access path matches the search argument. The READPE operation retrieves the prior record if the portion of the key of that record (or the record of the specified type) on the access path matches the search argument. For more information on the above operation codes, see the *ILE RPG for AS/400 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 OS/400 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 3c 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 RRR and Key information on block reads) for files in which the records are blocked and unblocked by the RPG compiler. The feedback area is updated each time a block of records is transferred. (For further details on the file information data structure see the *ILE RPG for AS/400 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 136 on page 289 and Figure 137 on page 289 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 136. 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 137. (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 138. (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 AS/400 environment, specify the record-address file length as 4, since each field is 4 bytes in length. When using a record-address file created in the System/36 environment™, specify the record-address file length as 3, since each field is 3 bytes in length. If you specify the record-address file length as blank, the compiler will check the primary record length at run time and determine whether to treat the record-address file as 3 byte or as 4 byte. A minus 1 (-1 or hexadecimal FFFFFFFF) relative-record-number value stops the use of a relative-record-address file record. End of file occurs when all records from the record-address file have been processed.

Methods for Processing Disk Files

The methods of disk file processing include:

- Consecutive processing
- Sequential-by-key processing
- Random-by-key processing
- Sequential-within-limits processing.
- Relative-record-number processing

Table 17 on page 292 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.

<i>Table 17. Processing Methods for DISK Files</i>			
Processing Method	Limits Processing (Pos. 28)	Record Address Type (Pos. 34)	File Organization (Pos. 35)
Externally Described Files			
<i>With Keys</i>			
Sequentially	Blank	K	Blank
Randomly	Blank	K	Blank
Sequential within limits (by record-address file)	L	K	Blank
<i>Without Keys</i>			
Randomly/consecutively	Blank	Blank	Blank
Program Described Files			
<i>With Keys (indexed file)</i>			
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
<i>Without Keys</i>			
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, see the *DB2 for AS/400 Database Programming*.

Sequential-by-Key Processing

For the sequential-by-key method of processing, records are read from the file in key sequence.

The sequential-by-key method of processing is valid for keyed files used as primary, secondary, or full procedural files.

For output files and for input files that do not use random functions (such as SETLL, SETGT, CHAIN, or ADD) and that have only one record format, the ILE RPG 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, see the *DB2 for AS/400 Database Programming* manual.

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 139 on page 294 and Figure 140 on page 294 shows the data description specifications (DDS) for the physical files used by the examples. Figure 141 on page 294 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 139. 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 140. 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 141. 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: YDRPT1
* 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 0 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 142. 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: YDRPT2
* 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 0 F 80 PRINTER
FEMPL1 IF E K DISK

* The two records (EMPREC and RCWEEK) are contained in the same
* file, and a record-identifying indicator is assigned to each
* record. The record-identifying indicators are used to control
* processing for the different record types. No control levels
* or match fields can be specified for a full-procedural file.

IEMPREC 01
I*
IRCWEEK 02
I*

* The READ operation code reads a record from the EMPL1 file. An
* end-of-file indicator is specified in positions 58 and 59. If
* the end-of-file indicator 99 is set on by the READ operation,
* the program branches to the EOFEND tag and processes the end-of-
* file routine.

C SETOFF 12
C READ EMPL1 99
C 99 GOTO EOFEND
C*
C 01 MOVE ENUM EMPNO 5 0
C*
C IF (*IN02='1') AND (ENUM=EMPNO)
C SETON 12
C ENDIF

* Since EMPL1 is defined as a full-procedural file, indicator
* *INLR has to be seton to terminate the program after processing
* the last record.

C EOFEND TAG
C 99 SETON LR

```

Figure 143 (Part 1 of 2). Sequential-by-Key Processing, Example 2

OPRINT	H	1P		2	6	
0						40 'EMPLOYEE WEEKLY WORKING '
0						52 'HOURS REPORT'
0	H	01		1		12 'EMPLOYEE: '
0			ENAME			32
0	H	01		1		12 'SERIAL #: '
0			ENUM			17
0			EDEPT			27 'DEPT: '
0			EDEPT			30
0			ETYPE			40 'TYPE: '
0			ETYPE			41
0	H	01		1		20 'WEEK #'
0						50 'HOURS WORKED'
0	D	12		1		
0			WEEKNO			18
0			EHWRK	3		45

Figure 143 (Part 2 of 2). Sequential-by-Key Processing, Example 2

EXAMPLE PROGRAM 3 (Matching-Record Technique): In this example, the TRWEEK file is defined as a secondary input file. The EMPREC and RCWEEK records are processed as matching records, with the ENUM field in both records assigned the match level value of M1. Record-identifying indicators 01 and 02 are assigned to the records to control the processing for the different record types.

```

*****
* PROGRAM NAME: YDRPT5 *
* 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 0 F 80 PRINTER
FEMPMST IP E K DISK
FTRWEEK IS E K DISK

IEMPREC 01
I ENUM M1
IRCWEEK 02
I ENUM M1

```

Figure 144 (Part 1 of 2). Sequential-by-Key Processing, Example 5

```

C 01          Z-ADD 0          TOTHR5 1
C 01          Z-ADD 0          TOTOV5 1
C 01          SETOFF          12
C*
C MR          IF      (*IN02='1')
C            ADD    EHWRK    TOTHR5
C            SUB    ENHRS    OVTHRS    4 111
C 11         ADD    OVTHRS    TOTOV5
C            SETON          12
C            ENDIF

OPRINT      H  1P          2 6
0          50 'YTD PAYROLL SUMMARY'
0          D  01          1
0          12 'EMPLOYEE: '
0          32
0          D  01          1
0          12 'SERIAL #: '
0          17
0          27 'DEPT: '
0          30
0          40 'TYPE: '
0          41
0          D  02 MR      1
0          8 'WEEK #'
0          10
0          32 'HOURS WORKED = '
0          38
0          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
* (TOTHR5 and TOTOV5) are equal to zeros in this case.

0          D  01NMR      1
0          70 'YTD HOURS WORKED = '
0          78
0          D  01NMR      1
0          70 'YTD OVERTIME HOURS = '
0          78

* These 2 total output lines are processed before performing
* detail calculations. Therefore, the total fields
* (TOTHR5 and TOTOV5) for the employee in the last retrieved
* record will be printed out if the specified indicators are on.

0          T  01 12      1
0          OR  LR 12
0          70 'YTD HOURS WORKED = '
0          78
0          T  01 12      1
0          OR  LR 12
0          70 'YTD OVERTIME HOURS = '
0          78
0          TOTOV5      3 78

```

Figure 144 (Part 2 of 2). Sequential-by-Key Processing, Example 5

Random-by-Key Processing

For the random-by-key method of processing, a search argument that identifies the key of the record to be read is specified in factor 1 of the calculation specifications for the CHAIN operation. Figure 146 on page 300 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 288).

Example of Random-by-Key Processing

The following is an example of how to use the random-by-key method of processing data. Figure 139 on page 294 and Figure 145 show the data description specifications (DDS) for the physical files used by EMSTUPD (Figure 146 on page 300).

```
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 145. 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 146. 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 147 on page 302 shows an example of an indexed file being processed sequentially within limits. Figure 149 on page 303 shows the same example with externally described files instead of program-described files.

Figure 139 on page 294 shows the data description specifications (DDS) for the physical file used by the program ESWLIM1 (Figure 147 on page 302) and ESWLIM2 (Figure 149 on page 303).

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   0   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
0                                               12 'SERIAL #'
0                                               22 'NAME'
0                                               45 'DEPT'
0                                               56 'TYPE'
0           D   01                               1
0                               ENUM              10
0                               ENAME            35
0                               EDEPT           45
0                               ETYPE           55

```

Figure 147. Sequential-within-Limits Processing of an Externally Described File

EXAMPLE PROGRAM 2 (Sequential-within-Limits Processing): Figure 148 shows the data description specifications (DDS) for the record-address limits file used by the program ESWLIM2 (Figure 149 on page 303).

```

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 148. 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
0                12 'SERIAL #'
0                22 'NAME'
0                45 'DEPT'
0                56 'TYPE'
0          D   01                1
0                ENUM             10
0                ENAME            35
0                EDEPT            45
0                ETYPE            55
0*

```

Figure 149. 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 18 on page 305 shows the valid file operation codes allowed for DISK files processed by keys and Table 19 on page 306 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 18. 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
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

Notes:

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 OS/400 CREATE FILE command.

Table 19. 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
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
Notes:					
<ol style="list-style-type: none"> 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 OS/400 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 AS/400:
 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 AS/400 when to start and end commitment control: Use the CL commands STRCMTCTL (Start Commitment Control) and ENDCMTCTL (End Commitment Control). See the *CL Reference* for information on these commands.
- In the RPG program:
 1. Specify commitment control (COMMIT) on the file-description specifications of the files you want under commitment control.
 2. Use the COMMIT (commit) operation code to apply a group of changes to files under commitment control, or use the ROLBK (Roll Back) operation code to eliminate the pending group of changes to files under commitment control. For information on how the rollback function is performed by the system, refer to *Backup and Recovery – Advanced* 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 308 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 311.

When you complete a group of changes with a COMMIT operation, you can specify a label to identify the end of the group. In the event of an abnormal job end, this identification label is written to a file, message queue, or data area so that you know which group of changes is the last group to be completed successfully. You specify this file, message queue, or data area on the STRCMTCTL command.

Before you call any program that processes files specified for commitment control, issue the STRCMTCTL command. If you call a program that opens a file specified

for commitment control before you issue the STRCMTCTL command, the opening of the file will fail.

The CL command ENDCMTCTL notifies the system that your activation group or job has finished processing files under commitment control. See the *CL Reference* for further information on the STRCMTCTL and ENDCMTCTL commands.

Commitment Control Locks

On the STRCMTCTL command, you specify a level of locking, either LCKLVL(*ALL), LCKLVL(*CHG), or LCKLVL(*CS). When your program is operating under commitment control and has processed an input or output operation on a record in a file under commitment control, the record is locked by commitment control as follows:

- Your program can access the record.
- Another program in your activation group or job, with this file under commitment control, can read the record. If the file is a shared file, the second program can also update the record.
- Another program in your activation group or job that does not have this file under commitment control cannot read or update the record.
- Another program in a separate activation group or job, with this file under commitment control, can read the record if you specified LCKLVL(*CHG), but it cannot read the record if you specified LCKLVL(*ALL). With either lock level, the next program cannot update the record.
- Another program that does not have this file under commitment control and that is not in your activation group or job can read but not update the record.
- Commitment control locks are different than normal locks, depend on the LCKLVL specified, and can only be released by the COMMIT and ROLBK operations.

The COMMIT and ROLBK operations release the locks on the records. The UNLOCK operation will not release records locked using commitment control. See the *CL Reference* for details on lock levels.

The number of entries that can be locked under commitment control before the COMMIT or ROLBK operations are required may be limited. For more information, see the *Backup and Recovery – Advanced* 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*, and also the *Data Management* manual.

Specifying Files for Commitment Control

To indicate that a DISK file is to run under commitment control, enter the keyword COMMIT in the keyword field of the file description specification.

When a program specifies commitment control for a file, the specification applies only to the input and output operations made by this program for this file. Commitment control does not apply to operations other than input and output operations. It does not apply to files that do not have commitment control specified in the program doing the input or output operation.

When more than one program accesses a file as a shared file, all or none of the programs must specify the file to be under commitment control.

Using the COMMIT Operation

The COMMIT operation tells the system that you have completed a group of changes to the files under commitment control. The ROLBK operation eliminates the current group of changes to the files under commitment control. For information on how to specify these operation codes and what each operation does, see the *ILE RPG for AS/400 Reference*.

If the system fails, it implicitly issues a ROLBK operation. You can check the identity of the last successfully completed group of changes using the label you specify in factor 1 of the COMMIT operation code, and the notify-object you specify on the STRCMTCTL command.

At the end of an activation group or job, or when you issue the ENDCMTCTL command, the OS/400 system issues an implicit ROLBK, which eliminates any changes since the last ROLBK or COMMIT operation that you issued. To ensure that all your file operations have effect, issue a COMMIT operation before ending an activation group or job operating under commitment control.

The OPEN operation permits input and output operations to be made to a file and the CLOSE operation stops input and output operations from being made to a file. However, the OPEN and CLOSE operations do not affect the COMMIT and ROLBK operations. A COMMIT or ROLBK operation affects a file, even after the file has been closed. For example, your program may include the following steps:

1. Issue COMMIT (for files already opened under commitment control).
2. Open a file specified for commitment control.
3. Perform some input and output operations to this file.
4. Close the file.

5. Issue ROLBK.

The changes made at step 3 are rolled back by the ROLBK operation at step 5, even though the file has been closed at step 4. The ROLBK operation could be issued from another program in the same activation group or job.

A program does not have to operate all its files under commitment control, and to do so may adversely affect performance. The COMMIT and ROLBK operations have no effect on files that are not under commitment control.

Note: When multiple devices are attached to an application program, and commitment control is in effect for the files this program uses, the COMMIT or ROLBK operations continue to work on a file basis and not by device. The database may be updated with partially completed COMMIT blocks or changes that other users have completed may be eliminated. It is your responsibility to ensure this does not happen.

Example of Using Commitment Control

This example illustrates the specifications and CL commands required for a program to operate under commitment control.

To prepare for using commitment control, you issue the following CL commands:

1. CRTJRNRCV JRNRCV (RECEIVER)

This command creates a journal receiver RECEIVER.

2. CRTJRN JRN(JOURNAL) JRNRCV(RECEIVER)

This command creates a journal JOURNAL and attaches the journal receiver RECEIVER.

3. STRJRNPF FILE(MASTER TRANS) JRN(JOURNAL)

This command directs journal entries for the file MASTER and the file TRANS to the journal JOURNAL.

In your program, you specify COMMIT for the file MASTER and the file TRANS:


```

*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... *
FFilename++IPEASFRlen+LKlen+AIDevice+.Keywords+++++
FMASTER  UF  E   K   DISK  COMMIT
FTRANS   UF  E   K   DISK  COMMIT
F*
*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... *
CLON01Factor1+++++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 150. 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 151 on page 312 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 ...+... *
CLON01Factor1+++++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 151. Example of Using Conditional Commitment Control

Commitment Control in the Program Cycle

Commitment control is intended for full procedural files, where the input and output is under your control. Do not use commitment control with primary and secondary files, where input and output is under the control of the RPG program cycle. The following are some of the reasons for this recommendation:

- You cannot issue a COMMIT operation for the last total output in your program.
- It is difficult to program within the cycle for recovery from a locked-record condition.
- Level indicators are not reset by the ROLBK operation.
- After a ROLBK operation, processing matching records may produce a sequence error.

DDM Files

ILE RPG 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 *Distributed Data Management* manual.

Using Pre-V3R1 DDM Files

If you are using a pre-Version 3 Release 1.0 DDM file, the key comparison is not done at the Data Management level during a READE, READPE, or SETLL operation, or during sequential-within-limits processing by a record address file. The READE, READPE, or SETLL operation or sequential-within-limits processing by a record address file, will instead compare the keys using the *HEX collating sequence.

This may give different results than expected when 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, READPE, or SETLL operation or during sequential-within-limits processing by a record address file, an informational message (RNI2002) will be issued.

Note: The performance of I/O operations that have the possibility of not finding a record (SETLL, CHAIN, SETGT, READE, READPE), will be slower than the pre-Version 3 Release 1.0 equivalent.

Chapter 17. Accessing Externally Attached Devices

You can access externally attached devices from RPG by using device files.

Device files are files that provide access to externally attached hardware such as printers, tape units, diskette units, display stations, and other systems that are attached by a communications line.

This chapter describes how to access externally attached devices using RPG device names PRINTER, SEQ, and SPECIAL. For information on display stations and ICF devices see Chapter 18, "Using WORKSTN Files" on page 331

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 20 to access the associated externally attached devices:

Device File	Associated Externally Attached Device	CL commands	RPG Device Name
Printer Files	Provide access to printer devices and describe the format of printed output.	CRTPRTF CHGPRTF OVRPRTF	PRINTER
Tape Files	Provide access to data files which are stored on tape devices.	CRTTAPF CHGTAPF OVRTAPF	SEQ
Diskette Files	Provide access to data files which are stored on diskette devices.	CRTDKTF CHGDKTF OVRDKTF	DISK
Display Files	Provide access to display devices.	CRTDSPF CHGDSPF OVRDSPF	WORKSTN
ICF Files	Allow a program on one system to communicate with a program on the same system or another system.	CRTICFF CHGICFF OVRICFF	WORKSTN

The device file contains the file description, which identifies the device to be used; it does not contain data.

Accessing Printer Devices

PRINTER files of ILE RPG programs associate with the printer files on the AS/400 system:

Printer files allow you to print output files. This chapter provides information on how to specify and use printer files in ILE RPG programs.

Specifying PRINTER Files

To indicate that you want your program to access printer files, specify PRINTER as the device name for the file in a File Description specification. Each file must have a unique file name. A maximum of eight printer files is allowed per program.

PRINTER files can be either externally-described or program-described. Overflow indicators OA-OG and OV, fetch overflow, space/skip entries, and the PRTCTL keyword are not allowed for an externally-described PRINTER file. See the *ILE RPG for AS/400 Reference* for the valid output specification entries for an externally-described file. See the *DDS Reference* for information about the DDS for externally-described printer files.

For an externally-described PRINTER file, you can specify the DDS keyword INDARA. If you try to use this keyword for a program-described PRINTER file, you get a run-time error.

You can use the CL command CRTPRTF (Create Print File) to create a printer file (see the *CL Reference* for further information on the CRTPRTF command); or you can also use the IBM-supplied file names. See the *Data Management* manual for more information on these file names.

The file operation codes that are valid for a PRINTER file are WRITE, OPEN, CLOSE, and FEOD. For a complete description of these operation codes, see the *ILE RPG for AS/400 Reference*.

Handling Page Overflow

An important consideration when you use a PRINTER file is page overflow. For an externally-described PRINTER file, you are responsible for handling page overflow. Do one of the following:

- Specify an indicator, *IN01 through *IN99, as the overflow indicator using the keyword OFLIND(*overflow indicator*) in the Keywords field of the file description specifications.
- Check the printer device feedback section of the INFDS for line number and page overflow. Refer to the *ILE RPG for AS/400 Reference* for more information.
- Count the number of output lines per page.
- Check for a file exception/error by specifying an indicator in positions 73 and 74 of the calculation specifications that specify the output operation, or by specifying an INFSR that can handle the error. The INFDS has detailed information on the file exception/error. See Chapter 12, "Handling Exceptions" on page 217 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 21 on page 319 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 153 on page 320.
- 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.

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 (<i>indicator</i>)	No entry	Continuous printing; no overflow recognized.
OFLIND (<i>indicator</i>)	Entry	Processes normal overflow.

Example of Printing Headings on Every Page

Figure 152 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
0.....N01N02N03FIELD++++++YB.END++PCONSTANT/EDITWORD/DTFORMAT++
0          OR      L2
0
0                      8 'DATE'
0                      18 'ACCOUNT'
0                      28 'N A M E'
0                      46 'BALANCE'
0*
```

Figure 152. Printing a Heading on Every Page

Example of Printing a Field on Every Page

Figure 153 on page 320 shows the necessary coding for the printing of certain fields on every page; a skip to 06 is done either on an overflow condition or on a change in control level (L2). The NL2 indicator prevents the line from printing and skipping twice in the same cycle.

```

*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... *
OFilename++DF..N01N02N03Excnam++++B++A++Sb+Sa+.....
OPRINT      D      OANL2                      3 6
0           OR      L2
0.....N01N02N03Field+++++++YB.End++PConstant/editword/DTformat++
0           ACCT                      8
0*

```

Figure 153. 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 154 on page 321 shows the normal processing of overflow printing when fetch overflow is set on and when it is set off.

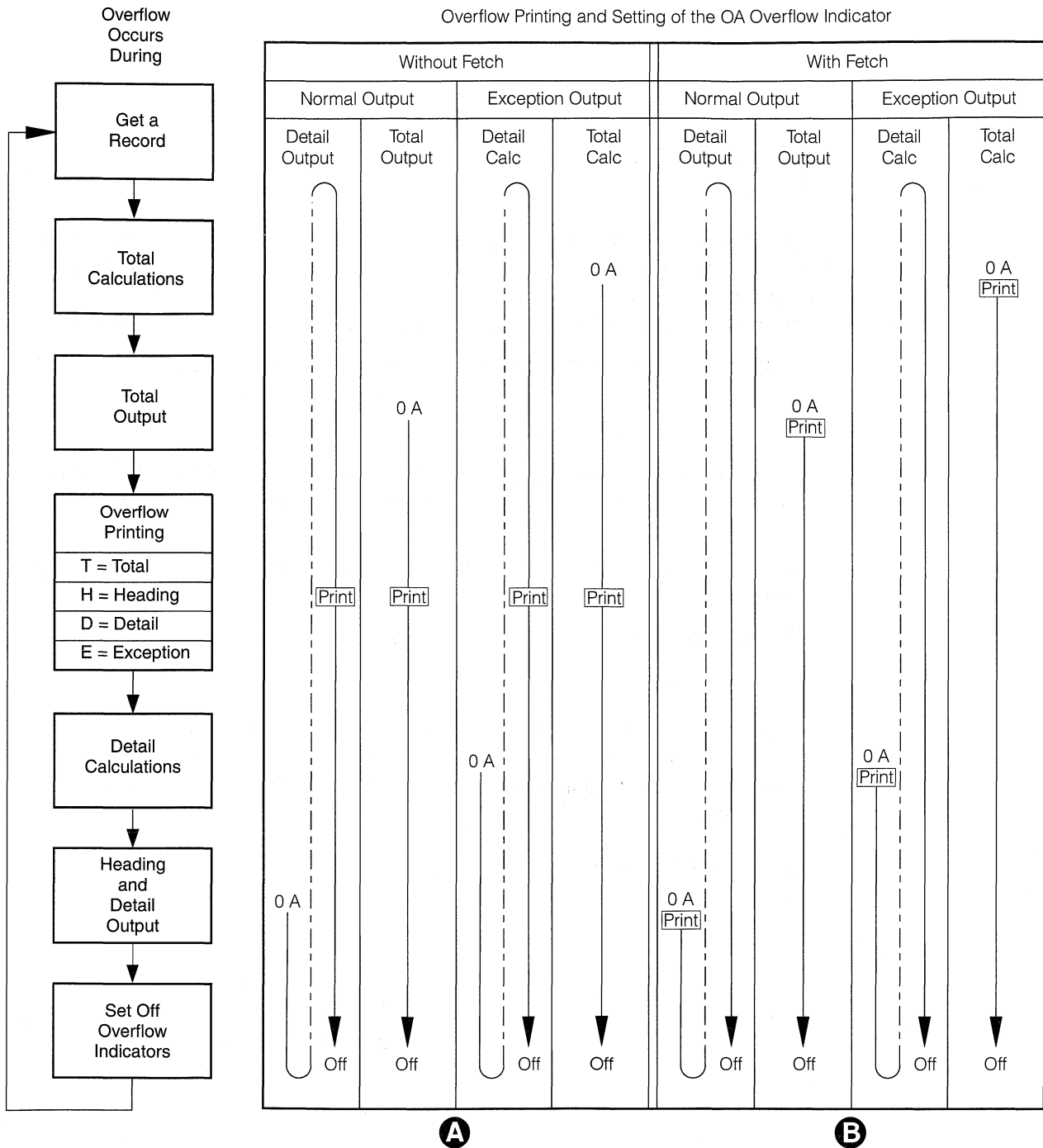


Figure 154. Overflow Printing: Setting of the Overflow Indicator

- 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 155 shows the use of fetch overflow.

```

*.. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... *
OFile name++DF..N01N02N03Excnam++++B++A++Sb+Sa+.....
OPRINTER  H   OA                3 05
0.....N01N02N03Field+++++++YB.End++PConstant/editword/DTformat++
0
0          TF  L1                1
0          T   L1          EMPLTOT      25
0          T   L1          EMPLTOT      35
0          T   L1          EMPLTOT      45
0          TF  L1                1
0          T   L1          EMPLTOT      55
0          T   L1          EMPLTOT      65
0          T   L1          EMPLTOT      75
0          T   L1                1
0*

```

Figure 155. 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 22. To use the OPM-defined data structure layout, specify PRTCTL(*data-structure name*:*COMPAT). The OPM PRTCTL data structure layout is shown in Table 23 on page 324.

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:

<i>Table 22. Layout of ILE PRTCTL Data Structure</i>	
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:

<i>Table 23. Layout of OPM PRTCTL Data Structure</i>	
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 156 on page 325 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      0      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
0.....N01N02N03Field+++++YB.End++PConstant/editword/DTformat++
0              DATA              25

```

Figure 156. Example of the PRTCTL Option

On the file description specifications, the PRTCTL keyword is specified for the PRINT file. The name of the associated data structure is LINE.

The LINE data structure is defined on the input specifications as having only those subfields that are predefined for the PRTCTL data structure. The first four subfields in positions 1 through 12 are used to supply space and skip information that is generally specified in positions 40 through 51 of the output specifications. The PRTCTL keyword allows you to change these specifications within the program.

In this example, the value in the SpAfter subfield is changed to 3 when the value in the CurLine (current line count value) subfield is equal to 10. (Assume that indicator 01 was set on as a record identifying indicator.)

Accessing Tape Devices

Use the SEQ device specifications whenever you write to a tape file. To write variable-length records to a tape file, use the RCDBLKFMT parameter of the CL command CRTTAPF or OVRTAPF. When you use the RCDBLKFMT parameter, the length of each record to be written to tape is determined by:

- the highest end position specified in the output specifications for the record or,
- if you do not specify an end position, the compiler calculates the record length from the length of the fields.

Read variable-length records from tape just like you would read records from any sequentially organized file. Ensure the record length specified on the file description specification accommodates the longest record in the file.

Accessing Display Devices

You use display files to exchange information between your program and a display device such as a workstation. A display file is used to define the format of the information that is to be presented on a display, and to define how the information is to be processed by the system on its way to and from the display.

See Chapter 18, "Using WORKSTN Files" on page 331 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 e.g. tape, printer and diskette.

You can also use the CL override commands, for example OVRDBF, OVRDKTF and OVRTAPF, to specify the actual I/O device when the program is run.

Specifying a Sequential File

A sequential (SEQ) device specification, entered in positions 36 through 42 in the file description specification, indicates that the input or output is associated with a sequentially-organized file. Refer to Figure 157 on page 327. The actual device to be associated with the file while running the program can be specified by a OS/400 override command or by the file description that is pointed to by the file name. If SEQ is specified in a program, no device-dependent functions such as space/skip, or CHAIN can be specified.

The following figure shows the operation codes allowed for a SEQ file.

File Description Specifications Positions		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 157 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 0  F 132      SEQ
*
```

Figure 157. SEQ Device

A SEQ device is specified for the PAYOTIME file. When the program is run, you can use a OS/400 override command to specify the actual device (such as printer, tape, or diskette) to be associated with the file while the program is running. For example, diskette can be specified for some program runs while printer can be specified for others. The file description, pointed to by the file name, can specify the actual device, in which case an override command need not be used.

Using SPECIAL Files

The RPG device name SPECIAL (positions 36 - 42 of the file description specifications) allows you to specify an input and/or output device that is not directly supported by the ILE RPG 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 158 on page 329. 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 25 on page 329 shows the file operation codes that are valid for a SPECIAL file.

Table 25. 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 158 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  0  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      MOVE         'HELLO'    FLD
C      MOVE         '1'        FLD1          1
C      WRITE        EXCPTN     OUTBUF
C      MOVE         '2'        FLD1          1
C      WRITE        EXCPTN     OUTBUF
C      SETON
LR

```

Figure 158. SPECIAL Device

Figure 159 on page 330 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 ...+... *
CLON01Factor1+++++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 159. 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.

Chapter 18. Using WORKSTN Files

Interactive applications on the AS/400 generally involve communication with:

- One or more work station users via display files
- One or more programs on a remote system via ICF files
- One or more devices on a remote system via ICF files.

Display files are objects of type *FILE with attribute of DSPF on the AS/400 system. You use display files to communicate interactively with users at display terminals. Like database files, display files can be either externally-described or program-described.

ICF files are objects of type *FILE with attribute of ICFF on the AS/400 system. You use ICF files to communicate with (send data to and receive data from) other application programs on remote systems (AS/400 or non-AS/400). An ICF file contains the communication formats required for sending and receiving data between systems. You can write programs that use ICF files which allow you to communicate with (send data to and receive data from) other application programs on remote systems.

When a file in an RPG program is identified with the WORKSTN device name then that program can communicate interactively with a work-station user or use the Intersystem Communications Function (ICF) to communicate with other programs. This chapter describes how to use:

- Intersystem Communications Function (ICF)
- Externally-described WORKSTN files
- Program-described WORKSTN files
- Multiple-device files.

Intersystem Communications Function

To use the ICF, define a WORKSTN file in your program that refers to an ICF device file. Use either the system supplied file QICDMF or a file created using the OS/400 command CRTICFF.

You code for ICF by using the ICF as a file in your program. The ICF is similar to a display file and it contains the communications formats required for the sending and receiving of data between systems.

For further information on the ICF, refer to *ICF Programming* 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 *DDS Reference*.)

In addition to the field descriptions (such as field names and attributes), the DDS for a display-device file are used to:

- Format the placement of the record on the screen by specifying the line-number and position-number entries for each field and constant.
- Specify attention functions such as underlining and highlighting fields, reverse image, or a blinking cursor.
- Specify validity checking for data entered at the display work station. Validity-checking functions include detecting fields where data is required, detecting mandatory fill fields, detecting incorrect data types, detecting data for a specific range, checking data for a valid entry, and processing modules 10 or 11 check-digit verification.
- Control screen management functions, such as determining if fields are to be erased, overlaid, or kept when new data is displayed.
- Associate indicators 01 through 99 with command attention keys or command function keys. If a function key is described as a command function key (CF), both the response indicator and the data record (with any modifications entered on the screen) are returned to the program. If a function key is described as a command attention key (CA), the response indicator is returned to the program but the data record remains unmodified. Therefore, input-only character fields are blank and input-only numeric field are filled with zeros, unless these fields have been initialized otherwise.
- Assign an edit code (EDTCDE) or edit word (EDTWRD) keyword to a field to specify how the field's values are to be displayed.
- Specify subfiles.

A display-device-record format contains three types of fields:

- *Input fields.* Input fields are passed from the device to the program when the program reads a record. Input fields can be initialized with a default value. If the default value is not changed, the default value is passed to the program. Input fields that are not initialized are displayed as blanks into which the workstation user can enter data.
- *Output fields.* Output fields are passed from the program to the device when the program writes a record to a display. Output fields can be provided by the program or by the record format in the device file.
- *Output/input (both) fields.* An output/input field is an output field that can be changed. It becomes an input field if it is changed. Output/input fields are passed from the program when the program writes a record to a display and passed to the program when the program reads a record from the display. Output/input fields are used when the user is to change or update the data that is written to the display from the program.

If you specify the keyword INDARA in the DDS for a WORKSTN file, the RPG program passes indicators to the WORKSTN file in a separate indicator area, and not in the input/output buffer.

For a detailed description of an externally-described display-device file and for a list of valid DDS keywords, see the *DDS Reference*.

Figure 160 on page 333 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
A          R PROMPT          REF(DSTREF) 1
A          73N61             TEXT('Item Prompt Format')
A                               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          PRICE      R      5 37'Price'
A          WHSLOC      R      7 2'Warehouse Location' 7
A          ONHAND      R      7 22
A          ONHAND      R      9 2'On Hand'
A          ALLOC      R      9 10
A          AVAIL      R      9 19'Allocated' 8
A          ALLOC      R      9 30
A          AVAIL      R      9 40'Available'
A*

```

Figure 160. Example of the Data Description Specifications for a Display Device File

This display device file contains two record formats: PROMPT and RESPONSE.

- 1 The attributes for the fields in this file are defined in the DSTREF field reference file.
- 2 The OVERLAY keyword is used so that both record formats can be used on the same display.
- 3 Function key 3 is associated with indicator 98, which is used by the programmer to end the program.
- 4 The PUTRETAIN keyword allows the value that is entered in the ITEM field to be kept in the display. In addition, the ITEM field is defined as an input field by the I in position 38. ITEM is the only input field in these record formats. All of the other fields in the record are output fields since position 38 is blank for each of them.
- 5 The ERRMSG keyword identifies the error message that is displayed if indicator 61 is set on in the program that uses this record format.
- 6 The LOCK keyword prevents the work-station user from using the keyboard when the RESPONSE record format is initially-displayed.
- 7 The constants such as 'Description', 'Price', and 'Warehouse Location' describe the fields that are written out by the program.
- 8 The line and position entries identify where the fields or constants are written on the display.

Specifying Function Key Indicators on Display Device Files

The function key indicators, KA through KN and KP through KY are valid for a program that contains a display device WORKSTN file if the associated function key is specified in the DDS.

The function key indicators relate to the function keys as follows: function key indicator KA corresponds to function key 1, KB to function key 2 ... KX to function key 23, and KY to function key 24.

Function keys are specified in the DDS with the CFxx (command function) or CAxx (command attention) keyword. For example, the keyword CF01 allows function key 1 to be used. When you press function key 1, function key indicator KA is set on in the RPG program. If you specify the function key as CF01 (99), both function key indicator KA and indicator 99 are set on in the RPG program. If the work-station user presses a function key that is not specified in the DDS, the OS/400 system informs the user that an incorrect key was pressed.

If the work-station user presses a specified function key, the associated function key indicator in the RPG program is set on when fields are extracted from the record (move fields logic) and all other function key indicators are set off. If a function key is not pressed, all function key indicators are set off at move fields time. The function key indicators are set off if the user presses the Enter key.

Specifying Command Keys on Display Device Files

You can specify the command keys Help, Roll Up, Roll Down, Print, Clear, and Home in the DDS for a display device file with the keywords HELP, ROLLUP, ROLLDOWN, PRINT, CLEAR, and HOME.

Command keys are processed by an RPG program whenever the compiler processes a READ or an EXFMT operation on a record format for which the appropriate keywords are specified in the DDS. When the command keys are in effect and a command key is pressed, the OS/400 system returns control to the RPG program. If a response indicator is specified in the DDS for the command selected, that indicator is set on and all other response indicators that are in effect for the record format and the file are set off.

If a response indicator is not specified in the DDS for a command key, the following happens:

- For the Print key without *PGM specified, the print function is processed.
- For the Roll Up and Roll Down keys used with subfiles, the displayed subfile rolls up or down, within the subfile. If you try to roll beyond the start or end of a subfile, you get a run-time error.
- For the Print Key specified with *PGM, Roll Up and Roll Down keys used without subfiles, and for the Clear, Help, and Home keys, one of the *STATUS values 1121-1126 is set, respectively, and processing continues.

Processing an Externally Described WORKSTN File

When an externally-described WORKSTN file is processed, the OS/400 system transforms data from the program to the format specified for the file and displays the data. When data is passed to the program, the data is transformed to the format used by the program.

The OS/400 system provides device-control information for processing input/output operations for the device. When an input record is requested from the device, the OS/400 system issues the request, and then removes device-control information from the data before passing the data to the program. In addition, the OS/400 system can pass indicators to the program indicating which fields, or if any fields, in the record have been changed.

When the program requests an output operation, it passes the output record to the OS/400 system. The OS/400 system provides the necessary device-control information to display the record. It also adds any constant information specified for the record format when the record is displayed.

When a record is passed to a program, the fields are arranged in the order in which they are specified in the DDS. The order in which the fields are displayed is based on the display positions (line numbers and position) assigned to the fields in the DDS. The order in which the fields are specified in the DDS and the order in which they appear on the screen need not be the same.

For more information on processing WORKSTN files, see “Valid WORKSTN File Operations” on page 341.

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 161 on page 336.) 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 162 on page 337 shows an example of the DDS for a subfile-record format, and Figure 163 on page 338 shows an example of the DDS for a subfile control-record format.

For a description of how to use subfile keywords, see the *DDS Reference*.

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
A          R SUBFIL                      REF(DSTREF) 1
A          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 162. 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 358.

```

*.. 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 163. 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 26 on page 341. The file name must be specified in factor 2 for this operation. A format must exist at the device before any input operations can take place. This requirement can be satisfied on a display device by conditioning an output record with 1P or by writing the first format to the device in another program (for example, in the CL program). The EXFMT operation is not valid for a program-described WORKSTN file. You can also use the EXCEPT operation to write to a WORKSTN file.

Additional Considerations

When using a format name with a program-described WORKSTN file, you must also consider the following:

- The name specified in positions 53 through 80 of the output specifications is assumed to be the name of a record format in the DDS that was used to create the file.
- If a Kn specification is present for an output record, it must also be used for any other output records for that file. If a Kn specification is not used for all output records to a file, a run-time error will occur.

Using a Program-Described WORKSTN File without a Format Name

When a record-format name is not used, a program-described display-device file describes a file containing one record-format description with one field. The fields in the record must be described within the program that uses the file.

When you create the display file by using the Create Display File command, the file has the following attributes:

- A variable record length can be specified; therefore, the actual record length must be specified in the using program. (The maximum record length allowed is the screen size minus one.)

- No indicators are passed to or from the program.
- No function key indicators are defined.
- The record is written to the display beginning in position 2 of the first available line.

Input File

For an input file, the input record, which is treated by the OS/400 device support as a single input field, is initialized to blanks when the file is opened. The cursor is positioned at the beginning of the field, which is position 2 on the display.

Output File

For an output file, the OS/400 device support treats the output record as a string of characters to be sent to the display. Each output record is written as the next sequential record in the file; that is, each record displayed overlays the previous record displayed.

Combined File

For a combined file, the record, which is treated by the OS/400 device support as a single field, appears on the screen and is both the output record and the input record. Device support initializes the input record to blanks, and the cursor is placed in position 2.

For more information on program-described-display-device files, see *Data Management* manual.

Valid WORKSTN File Operations

Table 26 shows the valid file operation codes for a WORKSTN file.

Table 26. Valid File Operation Codes for a WORKSTN File		
File Description Specifications Positions		Calculation Specifications Positions
17	18	26-35
I	P/S	CLOSE, ACQ, REL, NEXT, POST, FORCE
I	P/S	WRITE ¹ , CLOSE, ACQ, REL, NEXT, POST, FORCE
I	F	READ, OPEN, CLOSE, ACQ, REL, NEXT, POST
C	F	READ, WRITE ¹ , EXFMT ² , OPEN, CLOSE, ACQ, REL, NEXT, POST, UPDATE ³ , CHAIN ³ , READC ³
O	Blank	WRITE ¹ , OPEN, CLOSE, ACQ, REL, POST
Notes: <ol style="list-style-type: none"> 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 *ILE RPG for AS/400 Reference* for a complete description of each of these operation codes.

Multiple-Device Files

Any RPG WORKSTN file with at least one of the keywords DEVID, SAVEIND, MAXDEV(*FILE) or SAVEDS specified on the file description specification is a multiple-device file. Through a multiple-device file, your program may access more than one device.

The RPG program accesses devices through program devices, which are symbolic mechanisms for directing operations to an actual device. When you create a file (using the DDS and commands such as the create file commands), you consider such things as which device is associated with a program device, whether or not a file has a requesting program device, which record formats will be used to invite devices to respond to a READ-by-file-name operation, and how long this READ operation will wait for a response. For detailed information on the options and requirements for creating a multiple-device file, see the chapter on display files in *Data Management* manual, and information on ICF files in *ICF Programming* manual, and the manuals you are referred to in these two publications.

With multiple-device files, you make particular use of the following operation codes:

- In addition to opening a file, the OPEN operation implicitly acquires the device you specify when you create the file.
- The ACQ (acquire) operation acquires any other devices for a multiple-device file.
- The REL (release) operation releases a device from the file.
- The WRITE operation, when used with the DDS keyword INVITE, invites a program device to respond to subsequent read-from-invited- program-devices

operations. See the sections on inviting a program device in *ICF Programming* manual and *Data Management* 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 and *Data Management* 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 *ILE RPG for AS/400 Reference* for details of the RPG operation codes.

On the file description specification you can specify several keywords to control the processing of multiple-device files.

- The MAXDEV keyword indicates whether it is a single or multiple device file. Specify MAXDEV(*FILE) to process a multiple device file with the maximum number of devices taken from the definition of the file being processed. Specify MAXDEV(*ONLY) to process only one device.
- The DEVID keyword allows you to specify the name of a program device to which input and output operations are directed.

When a read-from-one-program-device or WRITE operation is issued, the device used for the operation is the device specified as the parameter to the DEVID keyword. This field is initialized to blanks and is updated with the name of the device from which the last successful input operation occurred. It can also be set explicitly by moving a value to it. The ACQ operation code does not affect the value of this field. If the DEVID keyword is not specified, the input operation is performed against the device from which the last successful input operation occurred. A blank device name is used if a read operation has not yet been performed successfully from a device.

When a read-from-one-program device or WRITE operation is issued with a blank device name, the RPG compiler implicitly uses the device name of the requestor device for the program. If you call an RPG program interactively and acquire an ICF device against which you want to perform one of these operations, you must explicitly move the device name of the ICF device into the field name specified with the DEVID keyword prior to performing the operation. If this is not done, the device name used will either be blank (in which case the interactive requestor device name is used), or the device name used is the one from the last successful input operation. Once you have performed an I/O oper-

ation to the ICF device, you do not need to modify the value again unless an input operation completes successfully with a different device.

- The **SAVEDS** keyword indicates a data structure that is saved and restored for each device acquired to a file. The **SAVEIND** keyword indicates a set of indicators to be saved and restored for each device acquired to a file. Before an input operation, the current set of indicators and data structure are saved. After the input operation, the RPG compiler restores the indicators and data structure for the device associated with the operation. This may be a different set of indicators or data structure than was available before the input operation.
- The **INFDS** keyword specifies the file information data structure for the **WORKSTN** file. The RPG ***STATUS** field and the major/minor return code for the I/O operation can be accessed through this data structure. Particularly when ICF is being used, both fields are useful for detecting errors that occurred during I/O operations to multiple-device files.

Note: When specifying these control options, you must code the **MAXDEV** option before the **DEVID**, **SAVEIND** or **SAVEDS** options.

Chapter 19. 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 27. Description of Each Module in the Interactive Application Example

Module	Description
"Main Menu Inquiry" on page 346	An example of a basic menu inquiry program that uses a WORKSTN file to display menu choices and accept input.
"File Maintenance" on page 349	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 358	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 364	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 164 on page 346 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 164. 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(QSYSVRT)
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 165. DDS for display device file MAINMENU

In addition to describing the constants, fields, line numbers, and horizontal positions for the screen, the record formats also define the display attributes for these entries.

Note: Normally, the field attributes are defined in a field-reference file rather than in the DDS for a file. The attributes are shown on the DDS so you can see what they are.

CUSMAIN: RPG Source

```

*****
* PROGRAM NAME: CUSMAIN *
* RELATED FILES: MAINMENU (DSPF) *
* RELATED PGMS: CUSMNT (ILE RPG PGM) *
* SCHZIP (ILE RPG PGM) *
* SCHNAM (ILE RPG PGM) *
* DESCRIPTION: THIS IS A CUSTOMER MAIN INQUIRY PROGRAM. *
* IT PROMPTS THE USER TO CHOOSE FROM ONE OF THE *
* FOLLOWING ACTIONS: *
* 1.MAINTAINS (ADD, UPDATE, DELETE AND DISPLAY) *
* CUSTOMER RECORD. *
* 2.SEARCH CUSTOMER RECORD BY ZIP CODE. *
* 3.SEARCH CUSTOMER RECORD BY NAME. *
*****

FMAINMENU CF E WORKSTN

C EXFMT HDRSCN
C*
C DOW NOT *IN03
C SELECT
C WHEN *IN05
C CALLB 'CUSMNT'
C WHEN *IN06
C CALLB 'SCHZIP'
C WHEN *IN07
C CALLB 'SCHNAM'
C ENDSL
C EXFMT HDRSCN
C ENDDO
C*
C SETON LR

```

Figure 166. 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 167 on page 349.

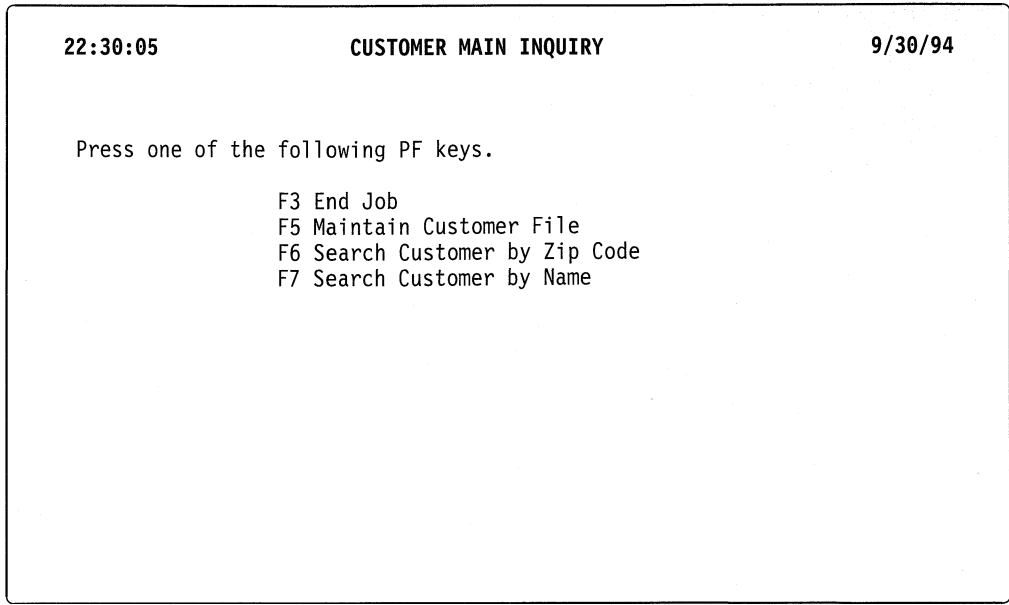


Figure 167. Customer Main Inquiry prompt screen

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 168. 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                                CA05(05 'ADD MODE')
A                                CA06(06 'UPDATE MODE')
A                                CA07(07 'DELETE MODE')
A                                CA08(08 'DISPLAY MODE')
A      MODE          8A 0 1 4DSPATR(HI)
A                                1 13'MODE'
A                                DSPATR(HI)
A                                2 4TIME
A                                DSPATR(HI)
A                                2 28'CUSTOMER FILE MAINTENANCE'
A                                DSPATR(HI RI)
A                                2 70DATE
A                                EDTCDE(Y)
A                                DSPATR(HI)
A      CUST          R    Y I 10 25DSPATR(CS)
A                                CHECK(RZ)
A 51                                ERRMSG('CUSTOMER ALREADY ON +
A                                FILE' 51)
A 52                                ERRMSG('CUSTOMER NOT ON FILE' +
A                                52)
A                                10 33'<--Enter Customer Number'
A                                DSPATR(HI)
A                                23 4'F3 End Job'
A                                23 21'F5 Add'
A                                23 34'F6 Update'
A                                23 50'F7 Delete'
A                                23 66'F8 Display'
A      R CSTINQ
A                                TEXT('DISPLAY CUST INFO')
A                                CA12(12 'PREVIOUS SCREEN')

```

Figure 169 (Part 1 of 2). DDS for display device file MNTMENU


```

A          MODE          8A 0 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      0 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 0 23 13
A          R CSTBLD TEXT('ADD CUST RECORD')
A          CA12(12 'PREVIOUS SCREEN')
A          MODE      8 0 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      0 4 25DSPATR(HI)
A          NAME      R      I 6 25
A          7 17'Address' DSPATR(HI)
A          ADDR1     R      I 7 25
A          8 17'Address' DSPATR(HI)
A          ADDR2     R      I 8 25
A          9 20'City'   DSPATR(HI)
A          CITY      R      I 9 25
A          10 19'State' DSPATR(HI)
A          STATE     R      I 10 25
A          10 36'Zip'   DSPATR(HI)
A          ZIP       R      Y I 10 40
A          23 2'F12 Cancel Addition'

```

Figure 169 (Part 2 of 2). DDS for display device file MNTMENU

The DDS for the MNTMENU display device file contains three record formats: HDRSCN, CSTINQ, and CSTBLD. The HDRSCN record prompts for the customer number and the mode of processing. The CSTINQ record is used for the Update,

Delete, and Display modes. The fields are defined as output/input (B in position 38). The fields are protected when Display or Delete mode is selected (DSPATR(PR)). The CSTBLD record provides only input fields (I in position 38) for a new record.

The HDRSCN record format contains the constant 'Customer File Maintenance'. The ERRMSG keyword defines the messages to be displayed if an error occurs. The CA keywords define the function keys that can be used and associate the function keys with indicators in the RPG program.

CUSMNT: RPG Source

```

*****
* PROGRAM NAME: CUSMNT *
* RELATED FILES: CUSMSTL1 (LF) *
* MNTMENU (DSPF) *
* DESCRIPTION: THIS PROGRAM SHOWS A CUSTOMER MASTER *
* MAINTENANCE PROGRAM USING A WORKSTN FILE. *
* THIS PROGRAM ALLOWS THE USER TO ADD, UPDATE, *
* DELETE AND DISPLAY CUSTOMER RECORDS. *
* PF3 IS USED TO QUIT THE PROGRAM. *
*****

FCUSMSTL1 UF A E K DISK
FMNTMENU CF E WORKSTN

C CSTKEY KLIST
C KFLD CUST

*****
* MAINLINE *
*****

C MOVE 'DISPLAY ' MODE
C EXFMT HDRSCN
C*
C DOW NOT *IN03
C EXSR SETMOD
C*
C CUST IFNE *ZERO
C MODE CASEQ 'ADD' ADDSUB
C MODE CASEQ 'UPDATE' UPDSUB
C MODE CASEQ 'DELETE' DELSUB
C MODE CASEQ 'DISPLAY' INQSUB
C
C ENDCS
C ENDIF
C*
C EXFMT HDRSCN
C ENDDO
C MOVE *ON *INLR

```

Figure 170 (Part 1 of 3). Source for module CUSMNT

```

*****
*   SUBROUTINE - ADDSUB                               *
*   PURPOSE   - ADD NEW CUSTOMER TO FILE             *
*****

```

```

C   ADDSUB      BEGSR
C   CSTKEY      CHAIN      CMLREC1                50
C               IF        NOT *IN50
C               MOVE      *ON          *IN51
C               ELSE
C               MOVE      *OFF         *IN51
C               MOVE      *BLANK      NAME
C               MOVE      *BLANK      ADDR1
C               MOVE      *BLANK      ADDR2
C               MOVE      *BLANK      CITY
C               MOVE      *BLANK      STATE
C               Z-ADD     *ZERO        ZIP
C               EXFMT     CSTBLD
C               IF        NOT *IN12
C               WRITE     CMLREC1
C               ENDIF
C               ENDIF
C               ENDSR

```

```

*****
*   SUBROUTINE - UPDSUB                               *
*   PURPOSE   - UPDATE CUSTOMER MASTER RECORD       *
*****

```

```

C   UPDSUB      BEGSR
C               MOVE      *OFF          *IN04
C   CSTKEY      CHAIN      CMLREC1                52
C               IF        NOT *IN52
C               EXFMT     CSTINQ
C               IF        NOT *IN12
C               UPDATE    CUSMSTL1
C               ELSE
C               UNLOCK    CUSMSTL1
C               ENDIF
C               ENDIF
C               ENDSR

```

Figure 170 (Part 2 of 3). Source for module CUSMNT

```

*****
*   SUBROUTINE - DELSUB                               *
*   PURPOSE    - DELETE CUSTOMER MASTER RECORD      *
*****

C   DELSUB      BEGSR
C               MOVE          *ON          *IN04
C   CSTKEY      CHAIN      CMLREC1          52
C               IF          NOT *IN52
C               EXFMT      CSTINQ
C               IF          NOT *IN12
C               DELETE     CMLREC1
C               ELSE
C               UNLOCK     CUSMSTL1
C               ENDIF
C               ENDIF
C               ENDSR

*****
*   SUBROUTINE - INQSUB                               *
*   PURPOSE    - DISPLAY CUSTOMER MASTER RECORD      *
*****

C   INQSUB      BEGSR
C               MOVE          *ON          *IN04
C   CSTKEY      CHAIN      CMLREC1          52
C               IF          NOT *IN52
C               EXFMT      CSTINQ
C               UNLOCK     CUSMSTL1
C               ENDIF
C               ENDSR

*****
*   SUBROUTINE - SETMOD                               *
*   PURPOSE    - SET MAINTENANCE MODE                *
*****

C   SETMOD      BEGSR
C               SELECT
C               WHEN          *IN05
C               MOVE          'ADD      '   MODE
C               WHEN          *IN06
C               MOVE          'UPDATE  '   MODE
C               WHEN          *IN07
C               MOVE          'DELETE  '   MODE
C               WHEN          *IN08
C               MOVE          'DISPLAY '   MODE
C               ENDSL
C*
C               MOVE          MODE          MODE1
C               ENDSR

```

Figure 170 (Part 3 of 3). Source for module CUSMNT

This program maintains a customer master file for additions, changes, and deletions. The program can also be used for inquiry.

The program first sets the default (display) mode of processing and displays the customer maintenance prompt screen. The workstation user can press F3, which turns on indicator 03, to request end of job. Otherwise, to work with customer information, the user enters a customer number and presses Enter. The user can

change the mode of processing by pressing F5 (ADD), F6 (UPDATE), F7 (DELETE), or F8 (DISPLAY).

To add a new record to the file, the program uses the customer number as the search argument to chain to the master file. If the record does not exist in the file, the program displays the CSTBLD screen to allow the user to enter a new customer record. If the record is already in the file, an error message is displayed. The user can press F12, which sets on indicator 12, to cancel the add operation and release the record. Otherwise, to proceed with the add operation, the user enters information for the new customer record in the input fields and writes the new record to the master file.

To update, delete, or display an existing record, the program uses the customer number as the search argument to chain to the master file. If a record for that customer exists in the file, the program displays the customer file inquiry screen CSTINQ. If the record is not in the file, an error message is displayed. If the mode of processing is display or delete, the input fields are protected from modification. Otherwise, to proceed with the customer record, the user can enter new information in the customer record input fields. The user can press F12, which sets on indicator 12, to cancel the update or delete operation, and release the record. Display mode automatically releases the record when Enter is pressed.

In Figure 171, 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 171. '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 172 on page 356.

```

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 172. 'Customer File Maintenance' Display Mode screen

The workstation user responds to the add prompt by entering a new customer number as shown in Figure 173.

```

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 173. 'Customer File Maintenance' Add Mode prompt screen

In Figure 174 on page 357 a new customer is added to the Customer Master File.

```

ADD MODE                                CUSTOMER FILE MAINTENANCE                                9/30/94
22:32:04

Customer: 00012

Name JUDAH GOULD
Address 2074 BATHURST AVENUE
Address
City YORKTOWN
State NY Zip 70068

F12 Cancel Addition

```

Figure 174. 'Customer File Maintenance' Add Mode prompt screen

The workstation user responds to the delete prompt by entering a customer number as shown in Figure 175.

```

DELETE MODE                                CUSTOMER FILE MAINTENANCE                                9/30/94
22:32:55

00011  <--Enter Customer Number

F3 End Job    F5 Add    F6 Update    F7 Delete    F8 Display

```

Figure 175. 'Customer File Maintenance' Delete Mode prompt screen

The workstation user responds to the update prompt by entering a customer number as shown in Figure 176 on page 358.

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 176. '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 177. DDS for logical file CUSMSTL2

The DDS for the database file used by this program describe one record format: CMLREC2. The logical file CUSMSTL2 keyed by zip code is based on the physical file CUSMST, as indicated by the PFILE keyword. The record format created by the logical file will include only those fields specified in the logical file DDS. All other fields will be excluded.

SZIPMENU: DDS for a Display Device File

```

A*****
A*   FILE NAME:  SZIPMENU                               *
A*  RELATED PGMS: SCHZIP                               *
A*  RELATED FILES: CUSMSTL2 (LOGICAL FILE)            *
A*  DESCRIPTION: THIS IS THE DISPLAY FILE SZIPMENU. IT HAS 6 *
A*                RECORD FORMATS.                    *
A*****
A                                     REF(CUSMSTL2)
A                                     CHGINPDTF(CS)
A                                     PRINT(QSYSVRT)
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
A                                     SFL
A      NAME      R      9 4
A      ARBAL      R      9 27EDTCDE(J)
A      R SUBCTL
A                                     SFLCTL(SUBFILE)

```

Figure 178 (Part 1 of 2). DDS for display device file SZIPMENU

```

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 0 4 14DSPATR(HI)
A 7 4'Customer Name'
A DSPATR(HI UL)
A 7 27'A/R Balance'
A DSPATR(HI UL)

```

Figure 178 (Part 2 of 2). DDS for display device file SZIPMENU

The DDS for the SZIPMENU display device file contains six record formats: HEAD, FOOT1, FOOT2, PROMPT, SUBFILE, and SUBCTL.

The PROMPT record format requests the user to enter a zip code. If the zip code is not found in the file, an error message is displayed. The user can press F3, which sets on indicator 03, to end the program.

The SUBFILE record format must be defined immediately preceding the subfile-control record format SUBCTL. The subfile record format, which is defined with the keyword SFL, describes each field in the record, and specifies the location where the first record is to appear on the display (here, on line 9).

The subfile-control record format contains the following unique keywords:

- SFLCTL identifies this format as the control record format and names the associated subfile record format.
- SFLCLR describes when the subfile is to be cleared of existing records (when indicator 55 is on). This keyword is needed for additional displays.
- SFLDSPCTL indicates when to display the subfile-control record format (when indicator 55 is off).
- SFLDSP indicates when to display the subfile (when indicator 55 is off).
- SFLSIZ specifies the total size of the subfile. In this example, the subfile size is 13 records that are displayed on lines 9 through 21.
- SFLPAG defines the number of records on a page. In this example, the page size is the same as the subfile size.
- ROLLUP indicates that indicator 95 is set on in the program when the roll up function is used.

The OVERLAY keyword defines this subfile-control record format as an overlay format. This record format can be written without the OS/400 system erasing the screen first. F4 is valid for repeating the search with the same zip code. (This use of F4 allows a form of roll down.)

SCHZIP: RPG Source

```

*****
* PROGRAM NAME: SCHZIP *
* RELATED FILES: CUSMSTL2 (LOGICAL FILE) *
* SZIPMENU (WORKSTN FILE) *
* DESCRIPTION: THIS PROGRAM SHOWS A CUSTOMER MASTER SEARCH *
* PROGRAM USING WORKSTN SUBFILE PROCESSING. *
* THIS PROGRAM PROMPTS THE USER FOR THE ZIP CODE*
* AND DISPLAYS THE CUSTOMER MASTER RECORD BY *
* ZIP CODE. *
* ROLL UP KEY CAN BE USED TO LOOK AT ANOTHER *
* PAGE. PF3 IS USED TO QUIT THE PROGRAM. *
*****

FCUSMSTL2 IF E K DISK
FSZIPMENU CF E WORKSTN SFILE(SUBFILE:RECNUM)

C CSTKEY KLIST
C KFLD ZIP

*****
* MAINLINE *
*****

C WRITE FOOT1
C WRITE HEAD
C EXFMT PROMPT
C*
C DOW NOT *IN03
C MOVE *OFF *IN61
C CSTKEY SETLL CMLREC2 20
C IF NOT *IN20
C MOVE *ON *IN61
C ELSE
C EXSR SFLPRC
C END
C IF (NOT *IN03) AND (NOT *IN04)
C IF NOT *IN61
C WRITE FOOT1
C WRITE HEAD
C ENDIF
C EXFMT PROMPT
C ENDIF
C ENDDO
C*
C SETON LR

```

Figure 179 (Part 1 of 2). Source for module SCHZIP

```

*****
*   SUBROUTINE - SFLPRC                                     *
*   PURPOSE   - PROCESS SUBFILE AND DISPLAY                *
*****

C   SFLPRC      BEGSR
C   NXPAG      TAG
C              EXSR      SFLCLR
C              EXSR      SFLFIL
C   SAMPAG     TAG
C              WRITE     FOOT2
C              WRITE     HEAD
C              EXFMT     SUBCTL
C              IF        *IN95
C              IF        NOT *IN71
C              GOTO      NXPAG
C              ELSE
C              GOTO      SAMPAG
C              ENDIF
C              ENDIF
C              ENDSR

*****
*   SUBROUTINE - SFLFIL                                     *
*   PURPOSE   - FILL SUBFILE                               *
*****

C   SFLFIL      BEGSR
C              DOW        NOT *IN21
C   ZIP         READE     CMLREC2      71
C              IF        *IN71
C              MOVE      *ON           *IN21
C              ELSE
C              ADD        1            RECNUM
C              WRITE     SUBFILE      21
C              ENDIF
C              ENDDO
C              ENDSR

*****
*   SUBROUTINE - SFLCLR                                     *
*   PURPOSE   - CLEAR SUBFILE RECORDS                     *
*****

C   SFLCLR      BEGSR
C              MOVE      *ON           *IN55
C              WRITE     SUBCTL
C              MOVE      *OFF          *IN55
C              MOVE      *OFF          *IN21
C              Z-ADD     *ZERO         RECNUM      5 0
C              ENDSR

```

Figure 179 (Part 2 of 2). Source for module SCHZIP

The file description specifications identify the disk file to be searched and the display device file to be used (SZIPMENU). The SFILE keyword for the WORKSTN file identifies the record format (SUBFILE) that is to be used as a subfile. The relative-record-number field (RECNUM) specified controls which record within the subfile is being accessed.

The program displays the PROMPT record format and waits for the workstation user's response. F3 sets on indicator 03, which controls the end of the program. The zip code (ZIP) is used to position the CUSMSTL2 file by the SETLL operation. Notice that the record format name CMLREC2 is used in the SETLL operation instead of the file name CUSMSTL2. If no record is found, an error message is displayed.

The SFLPRC subroutine handles the processing for the subfile: clearing, filling, and displaying. The subfile is prepared for additional requests in subroutine SFLCLR. If indicator 55 is on, no action occurs on the display, but the main storage area for the subfile records is cleared. The SFLFIL routine fills the subfile with records. A record is read from the CUSMSTL2 file. If the zip code is the same, the record count (RECNUM) is incremented and the record is written to the subfile. This subroutine is repeated until either the subfile is full (indicator 21 on the WRITE operation) or end of file occurs on the CUSMSTL2 file (indicator 71 on the READE operation). When the subfile is full or end of file occurs, the subfile is written to the display by the EXFMT operation by the subfile-control record control format. The user reviews the display and decides whether:

- To end the program by pressing F3.
- To restart the zip code by pressing F4. The PROMPT record format is not displayed, and the subfile is displayed starting over with the same zip code.
- To fill another page by pressing ROLL UP. If end of file has occurred on the CUSMSTL2 file, the current page is re-displayed; otherwise, the subfile is cleared and the next page is displayed.
- To continue with another zip code by pressing ENTER. The PROMPT record format is displayed. The user can enter a zip code or end the program.

In Figure 180, 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 180. 'Customer Search by Zip' prompt screen

The subfile is written to the screen as shown in Figure 181 on page 364.

22:34:45	CUSTOMER SEARCH BY ZIP	9/30/94
Zip Code 11201		
<u>Customer Name</u>	<u>A/R Balance</u>	
Rick Coupland	300.00	
Mikhail Yuri	150.00	
Karyn Sanders	5.00	
ENTER - Continue F3 - End Job F4 - RESTART ZIP CODE		

Figure 181. '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 182. DDS for logical file CUSMSTL3

The DDS for the database file used in this program defines one record format named CUSREC and identifies the NAME field as the key fields.

SNAMMENU: DDS for a Display Device File

```

A 55 SFLCLR
A N55 SFLDSPCTL
A N55 SFLDSP
A ROLLUP(95 'ROLL UP')
A OVERLAY
A CA04(04 'RESTART SEARCH NAME')
A 5 4'Search Name'
A SRCNAM R 0 5 17REFFLD(NAME CUSMSTL3)
A DSPATR(HI)
A 7 6'Select'
A DSPATR(HI)
A 8 6' "X" Customer Name '
A DSPATR(HI)
A DSPATR(UL)
A 8 42' Number Zip Code '
A DSPATR(HI)
A DSPATR(UL)
A R CUSDSP
A OVERLAY
A CA04(04 'RESTART ZIP CDE')
A 6 25'Customer'
A CUST 5S 00 6 35DSPATR(HI)
A 8 25'Name'
A NAME 20A 0 8 35DSPATR(HI)
A 10 25'Address'
A ADDR1 20A 0 10 35DSPATR(HI)
A ADDR2 20A 0 11 35DSPATR(HI)
A 13 25'City'
A CITY 20A 0 13 35DSPATR(HI)
A 15 25'State'
A STATE 2A 0 15 35DSPATR(HI)
A 15 41'Zip Code'
A ZIP 5S 00 15 50DSPATR(HI)
A 17 25'A/R Balance'
A ARBAL 10Y 20 17 42DSPATR(HI)
A EDTCDE(J)

```

Figure 183. DDS for display device file SNAMMENU

The DDS for the SNAMMENU display device file contains seven record formats: HEAD, FOOT1, FOOT2, PROMPT, SUBFILE, SUBCTL, and CUSDSP.

The PROMPT record format requests the user to enter a zip code and search name. If no entry is made, the display starts at the beginning of the file. The user can press F3, which sets on indicator 03, to end the program.

The SUBFILE record format must be defined immediately preceding the subfile-control record format SUBCTL. The subfile-record format defined with the keyword SFL, describes each field in the record, and specifies the location where the first record is to appear on the display (here, on line 9).

The subfile-control record format SUBCTL contains the following unique keywords:

- SFLCTL identifies this format as the control record format and names the associated subfile record format.
- SFLCLR describes when the subfile is to be cleared of existing records (when indicator 55 is on). This keyword is needed for additional displays.

- SFLDSPCTL indicates when to display the subfile-control record format (when indicator 55 is off).
- SFLDSP indicates when to display the subfile (when indicator 55 is off).
- SFLSIZ specifies the total size of the subfile. In this example, the subfile size is 13 records that are displayed on lines 9 through 21.
- SFLPAG defines the number of records on a page. In this example, the page size is the same as the subfile size.
- ROLLUP indicates that indicator 95 is set on in the program when the roll up function is used.

The OVERLAY keyword defines this subfile-control record format as an overlay format. This record format can be written without the OS/400 system erasing the screen first. F4 is valid for repeating the search with the same name. (This use of F4 allows a form of roll down.)

The CUSDSP record format displays information for the selected customers.

SCHNAM: RPG Source

```

*****
* PROGRAM NAME: SCHNAM *
* RELATED FILES: CUSMSTL3 (LOGICAL FILE) *
* SNAMMENU (WORKSTN FILE) *
* DESCRIPTION: THIS PROGRAM SHOWS A CUSTOMER MASTER SEARCH *
* PROGRAM USING WORKSTN SUBFILE PROCESSING. *
* THIS PROGRAM PROMPTS THE USER FOR THE CUSTOMER *
* NAME AND USES IT TO POSITION THE CUSMSTL3 *
* FILE BY THE SETLL OPERATION. THEN IT DISPLAYS *
* THE RECORDS USING SUBFILES. *
* TO FILL ANOTHER PAGE, PRESS THE ROLL UP KEY. *
* TO DISPLAY CUSTOMER DETAIL, ENTER 'X' BESIDE *
* THAT CUSTOMER AND PRESS ENTER. *
* TO QUIT THE PROGRAM, PRESS PF3. *
*****

FCUSMSTL3 IF E K DISK
FSNAMMENU CF E WORKSTN SFILE(SUBFILE:RECNUM)

C CSTKEY KLIST
C KFLD SRCNAM
C ZIPKEY KLIST
C KFLD NAME

```

Figure 184 (Part 1 of 3). Source for module SCHNAM


```

*****
*   MAINLINE                                           *
*****

C           WRITE   FOOT1
C           WRITE   HEAD
C           EXFMT   PROMPT
C           DOW     NOT *IN03
C   CSTKEY   SETLL   CUSREC
C           EXSR    SFLPRC
C           EXSR    SFLCHG
C           IF      (NOT *IN03) AND (NOT *IN04)
C           WRITE   FOOT1
C           WRITE   HEAD
C           EXFMT   PROMPT
C           ENDIF
C           ENDDO
C*
C           SETON                                     LR

*****
*   SUBROUTINE - SFLPRC                               *
*   PURPOSE   - PROCESS SUBFILE AND DISPLAY          *
*****

C   SFLPRC   BEGSR
C   NXTPAG   TAG
C           EXSR   SFLCLR
C           EXSR   SFLFIL
C   SAMPAG   TAG
C           WRITE  FOOT2
C           WRITE  HEAD
C           EXFMT  SUBCTL
C           IF     *IN95
C           IF     NOT *IN71
C           GOTO   NXTPAG
C           ELSE
C           GOTO   SAMPAG
C           ENDIF
C           ENDIF
C           ENDSR

*****
*   SUBROUTINE - SFLFIL                               *
*   PURPOSE   - FILL SUBFILE                          *
*****

C   SFLFIL   BEGSR
C           DOW   NOT *IN21
C           READ  CUSREC                                71
C           IF    *IN71
C           MOVE  *ON          *IN21
C           ELSE
C           ADD   1            RECNUM
C           MOVE  *BLANK       SEL
C           WRITE SUBFILE                                21
C           ENDIF
C           ENDDO
C           ENDSR

```

Figure 184 (Part 2 of 3). Source for module SCHNAM

```

*****
*   SUBROUTINE - SFLCLR                               *
*   PURPOSE    - CLEAR SUBFILE RECORDS                *
*****

C   SFLCLR      BEGSR
C               MOVE      *ON          *IN55
C               WRITE     SUBCTL
C               MOVE      *OFF         *IN55
C               MOVE      *OFF         *IN21
C               Z-ADD     *ZERO        RECNUM          5 0
C               ENDSR

*****
*   SUBROUTINE - SFLCHG                               *
*   PURPOSE    - CUSTOMER RECORD SELECTED             *
*****

C   SFLCHG      BEGSR
C               READC     SUBFILE          98
C               IF       NOT *IN98
C   ZIPKEY      CHAIN     CUSREC          71
C               EXFMT    CUSDSP
C               ENDIF
C               ENDSR

```

Figure 184 (Part 3 of 3). Source for module SCHNAM

The file description specifications identify the disk file to be searched and the display device file to be used (SNAMMENU). The SFILE keyword for the WORKSTN file identifies the record format (SUBFILE) to be used as a subfile. The relative-record-number field (RECNUM) specifies which record within the subfile is being accessed.

The program displays the PROMPT record format and waits for the workstation user's response. F3 sets on indicator 03, which controls the end of the program. The name (NAME) is used as the key to position the CUSMSTL3 file by the SETLL operation. Notice that the record format name CUSREC is used in the SETLL operation instead of the file name CUSMSTL3.

The SFLPRC subroutine handles the processing for the subfile: clearing, filling, and displaying. The subfile is prepared for additional requests in subroutine SFLCLR. If indicator 55 is on, no action occurs on the display, but the main storage area for the subfile records is cleared. The SFLFIL routine fills the subfile with records. A record is read from the CUSMSTL3 file, the record count (RECNUM) is incremented, and the record is written to the subfile. This subroutine is repeated until either the subfile is full (indicator 21 on the WRITE operation) or end of file occurs on the CUSMSTL3 file (indicator 71 on the READ operation). When the subfile is full or end of file occurs, the subfile is written to the display by the EXFMT operation by the subfile-control record control format. The user reviews the display and decides:

- To end the program by pressing F3.
- To restart the subfile by pressing F4. The PROMPT record format is not displayed, and the subfile is displayed starting over with the same name.

- To fill another page by pressing the ROLL UP keys. If end of file has occurred on the CUSMSTL3 file, the current page is displayed again; otherwise, the subfile is cleared, and the next page is displayed.
- To display customer detail by entering X, and pressing ENTER. The user can then return to the PROMPT screen by pressing ENTER, display the subfile again by pressing F4, or end the program by pressing F3.

In Figure 185, the user responds to the initial prompt by entering a customer name.

22:35:26	<u>CUSTOMER SEARCH & INQUIRY BY NAME</u>	9/30/94
Enter Search Name	JUDAH GOULD	
ENTER - Continue	F3 - End Job	

Figure 185. 'Customer Search and Inquiry by Name' prompt screen

The user requests more information by entering an X as shown in Figure 186 on page 370.

```

22:35:43                CUSTOMER SEARCH & INQUIRY BY NAME                9/30/94

Search Name  JUDAH GOULD

Select
  "X"
-----
X           JUDAH GOULD           00012       70068
           JUDAH GOULD           00209       31088

ENTER - Continue      F3 - End Job      F4 - Restart Name

```

Figure 186. 'Customer Search and Inquiry by Name' information screen

The detailed information for the customer selected is shown in Figure 187. 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 187. 'Customer Search and Inquiry by Name' detailed information screen

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. IGNDECDDTA 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. 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 of the main source file, 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.
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 non-RPG programs or procedures that end without an exception.
5. When recursion is detected, OPM RPG/400 displays inquiry message RPG8888. ILE RPG signals escape message RNX8888; no inquiry message is displayed for this condition. Note that this only applies to main procedures. Recursion is allowed for subprocedures.
6. 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.

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 247.
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.

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. You cannot modify the MR indicator using the MOVE or SETON operations. (RPG III only prevents using SETON with MR.)
3. The File Type entry on the File specification no longer dictates the type of I/O operations 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.
4. ILE RPG will allow record blocking even if the COMMIT keyword is specified on the file description specification.
5. In RPG IV, a file opened for update will also be opened as delete capable. You do not need any DELETE operations to make it delete capable.
6. In RPG IV, you do not have to code an actual number for the number of devices 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.
7. In ILE RPG, the ACQ and REL operation codes can be used with single device files.
8. 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.

9. 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".

10. 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.

11. 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.

12. 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 313 for more information.

13. 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.
14. 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.

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 382 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 398.

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 28 lists the various source member types, indicates whether the member type can be converted, and indicates the output source member type.

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 389.

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 188. The recommended record length of 112 characters also corresponds to the maximum amount of information that fits on a line of a compiler listing.

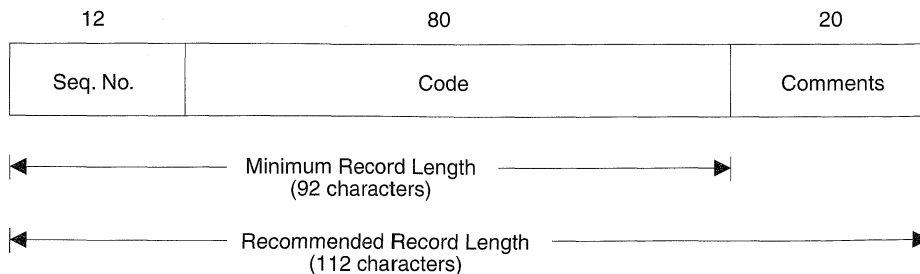


Figure 188. RPG IV Record Length Breakdown

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 QRPGLSRC and have a member name the same as the unconverted member name. (The library list will be searched for the file QRPGLSRC.)

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-supplied "model" database file QARNCVTLG in library QRPGL. Figure 195 on page 397 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(QRPGL) 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 396.

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 *ILE RPG for AS/400 Reference* for more information.

2. Create a log file, if necessary.

Unless you specify LOGFILE(*NONE), there must be a log file for the Conversion Aid to access. If you do not have one, then you can create one by using the CRTDUPOBJ command. For more information, see “The Log File” on page 381 and “Using the Log File” on page 396.

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 QRPGLSRC and 112 characters respectively. For additional file information see “File Considerations” on page 380.

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 QRPGLSRC. 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 383 for more information.

5. Check the log file or the error report for any errors. For more information, see “Analyzing Your Conversion” on page 393.
6. If there are errors, correct them and go to step 4 on page 382.
7. If there are no errors, create your program. For information on how to create ILE RPG programs, see Chapter 6, “Creating a Program with the CRTBNDRPG Command” on page 57.
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 398 for further information.
9. Once your converted source member has compiled successfully, retest the program before putting it back into production.

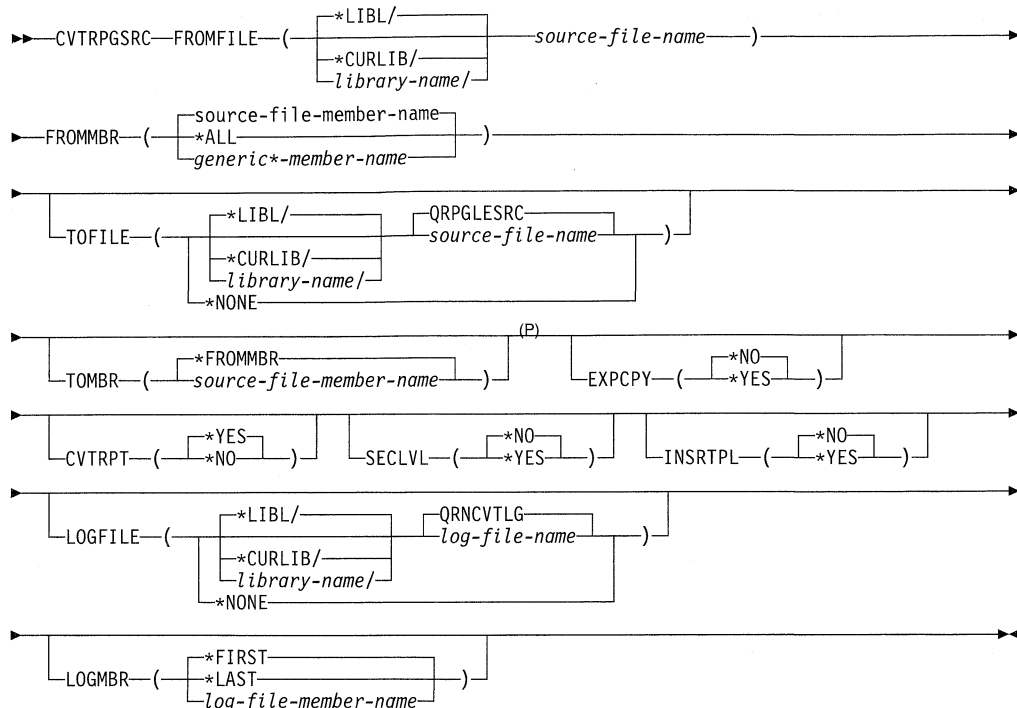
The CVTRPGSRC Command

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. Table 29 shows the parameters of the command based on their function.

<i>Table 29. CVTRPGSRC Parameters and Their Default Values Grouped by Function</i>	
Program Identification	
FROMFILE	Identifies library and file name of RPG source to be converted
FROMMBR	Identifies which source members are to be converted
TOFILE(*LIBL/QRPGLESRC)	Identifies library and file name of converted output
TOMBR(*FROMMBR)	Identifies file member names of converted source
Conversion Processing	
TOMBR	If *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
Conversion Feedback	
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

The syntax for the CVTRPGSRC command is shown below.

Job: B,I Pgm: B,I REXX: B,I Exec



Note:

^P All parameters preceding this point can be specified by position.

The parameters and their possible values follow the syntax diagram. If you need prompting, type CVTRPGSRC and press F4. The CVTRPGSRC screen appears, lists the parameters, and supplies default values. For a description of a parameter on the display, place your cursor on the parameter and press F1. Extended help for all of the parameters is available by pressing F1 on any parameter and then pressing F2.

FROMFILE

Specifies the name of the source file that contains the RPG III or RPG 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 LIBL 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.

***FROMMBR**

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.

***NO**

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.

***YES**

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.

***NO**

Second-level message text is not printed in the conversion report.

***YES**

Second-level message text is printed in the conversion report.

INSRTPL

Specifies if the ILE RPG specification templates (H-, F-, D-, I-, C- and/or O-specification template), are inserted in the converted source member(s). The default value is *NO.

***NO**

A specification template is not inserted in the converted source member.

***YES**

A specification template is inserted in the converted source member. Each specification template is inserted at the beginning of the appropriate specification section.

LOGFILE

Specifies the name of the log file that is used to track the conversion information. Unless *NONE is specified, there must be a log file. The file must already exist, and it must be a physical data file. Create the log file by using the CPYF command with the "From object" file QARNCVTLG in library QRPGL 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 QRPGLSRC 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 QRPGLSRC 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 QRPGLSRC, you would enter:

```
CVTRPGSRC FROMFILE(OLDRPG/QRPGSRC)
          FROMMBR(*ALL)
          TOFILE(NEWRPGL/QRPGLSRC)
          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 NEWRPGL in the source physical file QRPGLSRC.

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 28 on page 380).

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(NEWRPGL/QRPGLSRC)
          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 NEWRPGL in the source physical file QRPGLSRC.

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 393.

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.

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 188 on page 380.) 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 189 on page 391 shows the source of the RPG III version.


```

H                                                    TSTPGM
FFILE1  IF  E                                DISK      COMM1
FQSYSPRT 0  F    132    OF  LPRINTER
LQSYSPRT 60FL 560L
E                ARR1  3  3  1                COMM2
E                ARR2  3  3  1
IFORMAT1
I                OLDNAME                      NAME
I* DATA STRUCTURE COMMENT
IDS1        DS
I                1  3 FIELD1
I* NAMED CONSTANT COMMENT
I                'XYZ'                        C        CONST1      COMM3
I                4  6 ARR1
C                ARR1,3  DSPLY
C                READ FORMAT1                01
C                NAME    DSPLY
C                SETON                        LR
C                EXCPTOUTPUT
OQSYSPRT E  01                OUTPUT
O                ARR2,3  10
**
123
**
456

```

Figure 189. 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 190 on page 392.

```

1 .....H*unctions+++++Comments+++++
2   H DFTNAME(TSTPGM)
3 .....F*ilename++IPEASFRlen+LKlen+AIDevice+.Functions+++++Comments+++++
4   FFILE1   IF   E           DISK                               COMM1
5   FQSYSVRT 0   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+SPFfrom+To+++DcField+++++L1M1FrP1MnZr.....Comments+++++
19   IFORMAT1
20   I             OLDNAME           NAME
21 .....C*ON01Factor1+++++Opcod(E)+Factor2+++++Resul+++++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   OQSYSVRT   E           OUTPUT           01
28   O           ARR2(3)           10
29 **CTDATA ARR1
30 123
31 **CTDATA ARR2
32 456

```

Figure 190. 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 191 shows the command summary for a sample conversion.

```
5769RG1 V4R2M0 980228RN          IBM ILE RPG          AS400S01  12/30/97 20:41:35    Page    1
Command . . . . . : CVTRPGSRC
Issued by . . . . . : DAVE

From file . . . . . : QRPGSRC
Library . . . . . : MYLIB
From member . . . . . : REPORT

To file. . . . . : QRPGLSRC
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 191. 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 192 on page 395 shows the source section for the sample conversion.

```

5769RG1 V4R2M0 980228RN      IBM ILE RPG      AS400S01  12/30/97 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 192. 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 193 shows the messages section for the sample conversion, including second-level messages.

```

5769RG1 V4R2M0 980228RN      IBM ILE RPG      AS400S01  12/30/97 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 193. 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 194 on page 396 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 194. 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 QRPGL 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 195 on page 397 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.

```

CPYF FROMFILE(QRPGL/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	LGFMR	10A	COLHDG('FROM' 'MBR')
A			TEXT('From Member')
A	LGFRT	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 195. DDS for model log file QARNCVTLG in library QRPGL

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 389. 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 373.

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 390 for the RPG III source member TEST1. If you compare the placement of the data structure DS1 in Figure 189 on page 391 and in Figure 190 on page 392, 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 196 and Figure 197 show the source for TEST2 and COPYDS1 respectively.

```

H                                                    TSTPGM
FFILE1  IF  E                                DISK      COMM1
FQSYSPRT 0  F    132    OF  LPRINTER
LQSYSPRT 60FL 560L
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 196. 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 197. 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 *ILE RPG for AS/400 Reference*.

Other examples of this type of problem include:

- Line Specifications and Record Address Files

In RPG III the line counter specification and the Record Address File of the extension specification are changed to keywords (RAFDATA, FORMLEN, and FORMOFL) on the file description specification. If the content of a /COPY member contains only the line counter specification and/or the Record Address File of the extension specification but not the corresponding file description specification, the Conversion Aid does not know where to insert the keywords.

- Extension Specification Arrays and Data Structure Subfields

As mentioned in “Example of Source Conversion” on page 390, 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 190 on page 392), 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 190 on page 392), 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 198. 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 199. 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 198 on page 400 to the code shown in Figure 200. 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 200. 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 *ILE RPG for AS/400 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 201 is included twice in Figure 202. 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 203. 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 204.

```
A      R RECORD
A      CHARACTER    10
A      ARRAY        10
```

Figure 201. DDS for external data structure

```
E              ARRAY    10  1
IDSONE      E DSEXTREC
C          CHAR      DSPLY
C              SETON                      LR
```

Figure 202. 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 203. 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 204. 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 205 on page 403. The converted source also contains two source lines, as shown in Figure 206 on page 403. 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 207 on page 403, 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 205. RPG III source with renamed and initialized external subfield

```

D DSONE      E DS           EXTNAME(EXTREC)
D CHAR      E           EXTFLD(CCHARACTER)
D CHAR      E           INZ('XYZ')
C CHAR      DSPLY
C           SETON              LR

```

Figure 206. RPG IV source with two definitions for renamed subfield

```

D DSONE      E DS           EXTNAME(EXTREC)
D CHAR      E           EXTFLD(CCHARACTER) INZ('XYZ')
C CHAR      DSPLY
C           SETON              LR

```

Figure 207. 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 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 split across these two activation groups, you are mixing OPM behavior with ILE behavior and your results may be hard to predict. Refer to Chapter 3, "Program Creation Strategies" on page 23 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 Reference*.

Using CL Commands

Control Language (CL) commands, parameters, and keywords can be entered in either uppercase or lowercase characters. In the syntax diagram they are shown in uppercase (for example, PARAMETER, PREDEFINED-VALUE). Variables appear in lowercase italic letters (for example, *user-defined-value*). Variables are user-defined names or values.

How to Interpret Syntax Diagrams

The syntax diagrams in this book use the following conventions:

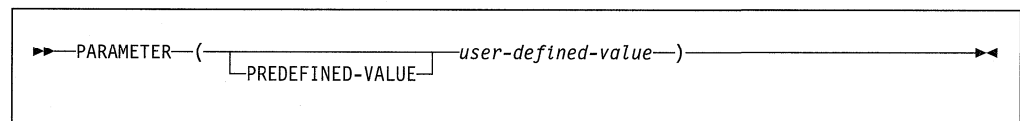


Figure 208. Structure of a Syntax Diagram

Read the syntax diagram from left to right, and from top to bottom, following the path of the line.

The ▶▶ symbol indicates the beginning of the syntax diagram.

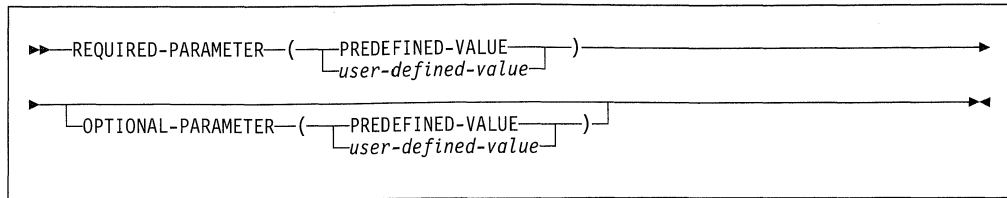
The ◀◀ symbol indicates the end of the syntax diagram.

The —▶ symbol indicates that the statement syntax is continued on the next line.

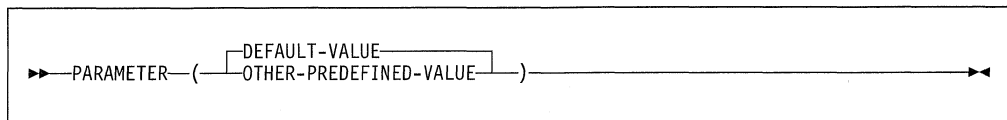
The ▶— symbol indicates that a statement is continued from the previous line.

The —(—) symbol indicates that the parameter or value must be entered in parentheses.

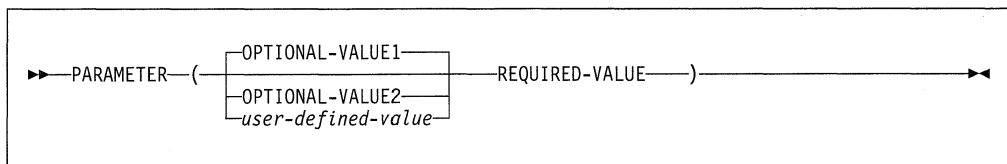
Required parameters appear on the base line and must be entered. **Optional parameters** appear below the base line and do not need to be entered. In the following sample, you must enter REQUIRED-PARAMETER and a value for it, but you do not need to enter OPTIONAL-PARAMETER or a value for it.



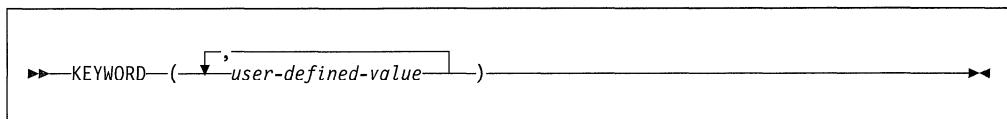
Default values appear above the base line and do not need to be entered. They are used when you do not specify a parameter. In the following sample, you can enter DEFAULT-VALUE, OTHER-PREDEFINED-VALUE, or nothing. If you enter nothing, DEFAULT-VALUE is assumed.



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).



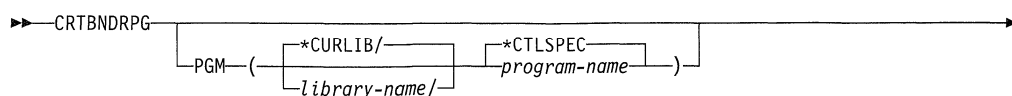
Repeated values can be specified for some parameters. The comma (,) in the following sample indicates that each *user-defined-value* must be separated by a comma.

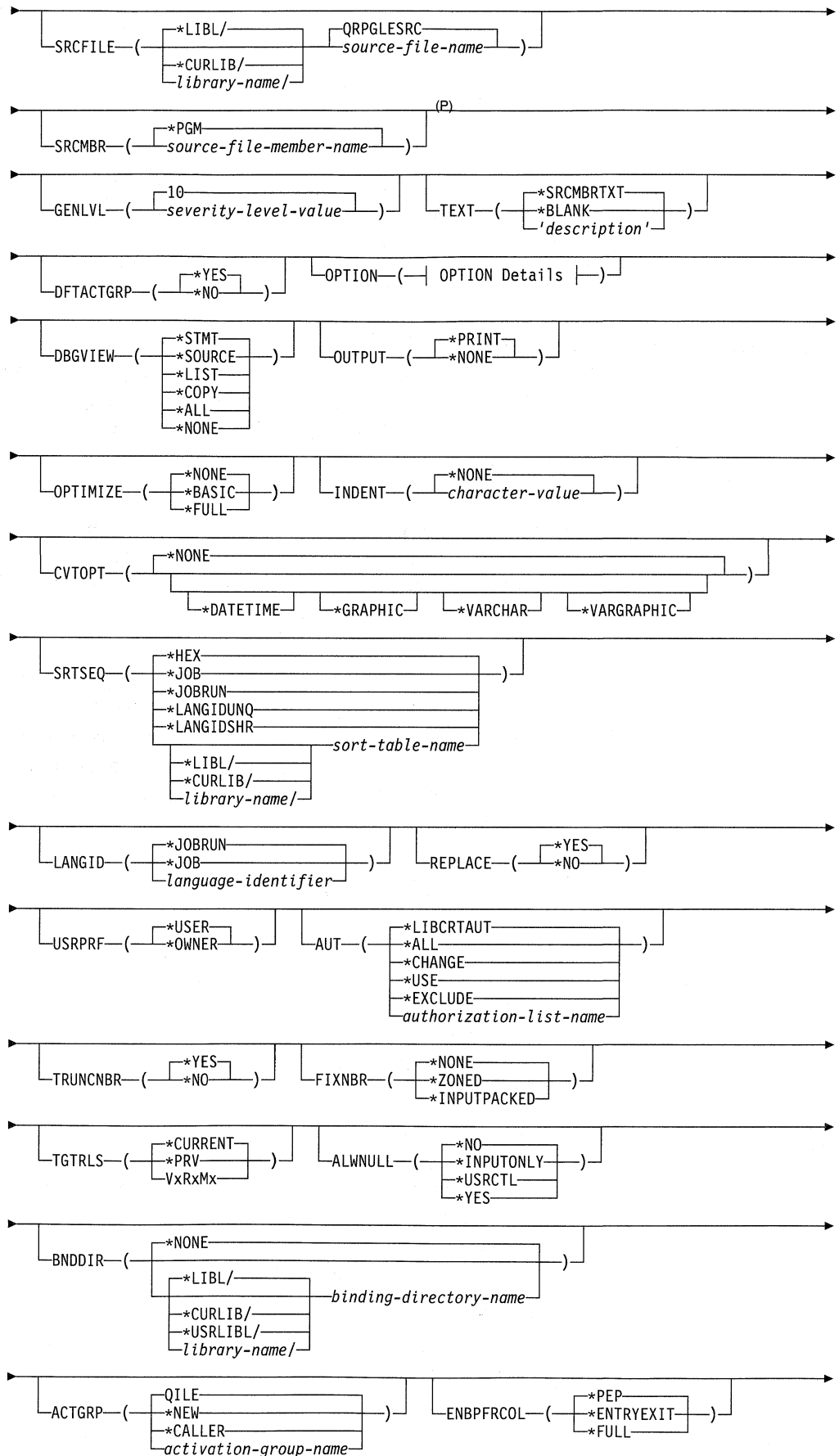


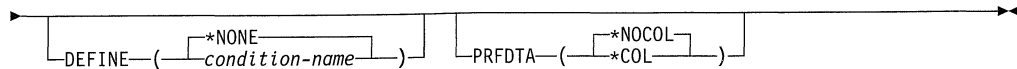
CRTBNDRPG Command

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



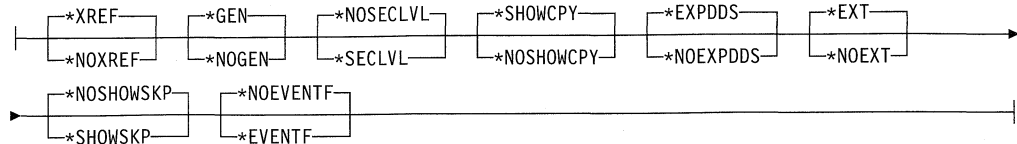




Note:

^P All parameters preceding this point can be specified by position.

OPTION Details:



Description of the CRTBNDRPG Command

The parameters, keywords, and variables of the CRTBNDRPG command are listed below. The same information is available online. Enter the command name on a command line, press PF4 (Prompt) and then press PF1 (Help) for any parameter you want information on.

PGM

Specifies the program name and library name for the program object (*PGM) you are creating. The program name and library name must conform to AS/400 naming conventions. If no library is specified, the created program is stored in the current library.

***CTLSPEC**

The name for the compiled program is taken from the name specified in the DFTNAME keyword of the control specification. If the program name is not specified on the control specification and the source member is from a database file, the member name, specified by the SRCMBR parameter, is used as the program name. If the source is not from a database file then the program name defaults to RPGPGM.

program-name

Enter the name of the program object.

***CURLIB**

The created program object is stored in the current library. If you have not specified a current library, QGPL is used.

library-name

Enter the name of the library where the created program object is to be stored.

SRCFILE

Specifies the name of the source file that contains the ILE RPG source member to be compiled and the library where the source file is located. The recommended source physical file length is 112 characters: 12 for the sequence number and date, 80 for the code and 20 for the comments. This is the maximum amount of source that is shown on the compiler listing.

QRPGLESRC

The default source file QRPGLESRC contains the ILE RPG source member to be compiled.

source-file-name

Enter the name of the source file that contains the ILE RPG source member to be compiled.

***LIBL**

The system searches the library list to find the library where the source file is stored. This is the default.

***CURLIB**

The current library is used to find the source file. If you have not specified a current library, QGPL is used.

library-name

Enter the name of the library where the source file is stored.

SRCMBR

Specifies the name of the member of the source file that contains the ILE RPG source program to be compiled.

***PGM**

Use the name specified by the PGM parameter as the source file member name. The compiled program object will have the same name as the source file member. If no program name is specified by the PGM parameter, the command uses the first member created in or added to the source file as the source member name.

source-file-member-name

Enter the name of the member that contains the ILE RPG source program.

GENLVL

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.

DFACTGRP

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.

***XREF**

Produces a cross-reference listing (when appropriate) for the source member.

***NOXREF**

A cross-reference listing is not produced.

***GEN**

Create a program object if the highest severity level returned by the compiler does not exceed the severity specified in the GENLVL option.

***NOGEN**

Do not create a program object.

***NOSECLVL**

Do not print second-level message text on the line following the first-level message text.

***SECLVL**

Print second-level message text on the line following the first-level message text in the Message Summary section.

***SHOWCPY**

Show source records of members included by the /COPY compiler directive.

***NOSHOWCPY**

Do not show source records of members included by the /COPY compiler directive.

***EXPDDS**

Show the expansion of externally described files in the listing and display key field information.

***NOEXPDDS**

Do not show the expansion of externally described files in the listing or display key field information.

***EXT**

Show the list of external procedures and fields referenced during the compile on the listing.

***NOEXT**

Do not show the list of external procedures and fields referenced during the compilation on the listing.

***NOSHOWSKP**

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.

***NOEVENTF**

Do not create an Event File for use by CoOperative Development Environment/400 (CODE/400). CODE/400 uses this file to provide error feedback integrated with the CODE/400 editor. An Event File is normally created when you create a module or program from within CODE/400.

***EVENTF**

Create an Event File for use by CoOperative Development Environment/400 (CODE/400). The Event File is created as a member in file EVFEVENT in the library where the created module or program object is to be stored. If the file EVFEVENT does not exist it is automatically created. The Event File member name is the same as the name of the object being created.

CODE/400 uses this file to provide error feedback integrated with the CODE/400 editor. An Event File is normally created when you create a module or program from within CODE/400.

DBGVIEW

Specifies which level of debugging is available for the compiled program object, and which source views are available for source-level debugging.

***STMT**

Allows the program object to be debugged using the Line Numbers shown on the left-most column of the source section of the compiler listing.

***SOURCE**

Generates the source view for debugging the compiled program object. This view is not available if the root source member is a DDM file. Also, if changes are made to any source members after the compile and before attempting to debug the program, the views for those source members may not be usable.

***LIST**

Generates the listing view for debugging the compiled program object. The information contained in the listing view is dependent on whether *SHOWCPY and *EXPDDS are specified for the OPTION parameter.

Note: The listing view will not show any indentation which you may have requested using the Indent option.

***COPY**

Generates the source and copy views for debugging the compiled program object. The source view for this option is the same source view generated for the *SOURCE option. The copy view is a debug view which has all the /COPY source members included. These views will not be available if the root source member is a DDM file. Also, if changes are made to any source members after the compile and before attempting to debug the program, the views for those source members may not be usable.

***ALL**

Generates the listing, source and copy views for debugging the compiled program object. The information contained in the listing view is dependent on whether *SHOWCPY and *EXPDDS are specified for the OPTION parameter.

***NONE**

Disables all of the debug options for debugging the compiled program object.

OUTPUT

Specifies if a compiler listing is generated.

***PRINT**

Produces a compiler listing, consisting of the ILE RPG program source and all compile-time messages. The information contained in the listing is dependent on whether *XREF, *SECLVL, *SHOWCPY, *SHOWSKP, *EXPDDS, and *EXT are specified for the OPTION parameter.

***NONE**

Do not generate the compiler listing.

OPTIMIZE

Specifies the level of optimization, if any, of the program.

***NONE**

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)`.

***NONE**

Structured operations will not be indented in the source listing.

character-value

The source listing is indented for structured operation clauses. Alignment of statements and clauses are marked using the characters you choose. You can choose any character string up to 2 characters in length. If you want to use a blank in your character string, you must enclose 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.

***NONE**

Ignores variable-length database data types and use the native RPG date, time, timestamp and graphic data types.

***DATETIME**

Specifies that date, time, and timestamp database data types are to be declared as fixed-length character fields.

***GRAPHIC**

Specifies that double-byte character set (DBCS) graphic data types are to be declared as fixed-length character fields.

***VARCHAR**

Specifies that variable-length character data types are to be declared as fixed-length character fields.

***VARGRAPHIC**

Specifies that variable-length double-byte character set (DBCS) graphic data types are to be declared as fixed-length character fields.

SRTSEQ

Specifies the sort sequence table that is to be used in the ILE RPG source program.

***HEX**

No sort sequence table is used.

***JOB**

Use the SRTSEQ value for the job when the *PGM is created.

***JOB RUN**

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.

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.

***JOB RUN**

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 Reference*

***LIBCRTAUT**

The public authority for the object is taken from the CRTAUT keyword of the target library (the library that contains the object). The value is determined when the object is created. If the CRTAUT value for the library changes after the create, the new value will not affect any existing objects.

***ALL**

Authority for all operations on the program object, except those limited to the owner or controlled by authorization list management authority. The user can control the program object's existence, specify this security for it, change it, and perform basic functions on it, but cannot transfer its ownership.

***CHANGE**

Provides all data authority and the authority to perform all operations on the program object except those limited to the owner or controlled by object authority and object management authority. The user can change the object and perform basic functions on it.

***USE**

Provides object operational authority and read authority; 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, V4R2M0 is version 4, release 2, modification level 0.

Valid values for this parameter change every release.

The possible values are:

***CURRENT**

The object is to be used on the release of the operating system currently running on your system. For example, if V4R2M0 is running on the system, *CURRENT means you intend to use the object on a system with V4R2M0 installed. You can also use the object on a system with any subsequent release of the operating system installed.

***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.

ALWNULL

Specifies how the ILE RPG module will be allowed to use records containing null-capable fields from externally described database files.

***NO**

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.

BNDDIR

Specifies the list of binding directories that are used in symbol resolution.

***NONE**

No binding directory is specified.

binding-directory-name

Specify the name of the binding directory used in symbol resolution.

The directory name can be qualified with one of the following library values:

***LIBL**

The system searches the library list to find the library where the binding directory is stored.

***CURLIB**

The current library for the job is searched. If no library is specified as the current library for the job, library QGPL is used.

***USRLIBL**

Only the libraries in the user portion of the job's library list are searched.

library-name

Specify the name of the library to be searched.

ACTGRP

Specifies the activation group this program is associated with when it is called.

QILE

When this program is called, it is activated into the named activation group QILE.

***NEW**

When this program is called, it is activated into a new activation group.

***CALLER**

When this program is called, it is activated into the caller's activation group.

activation-group-name

Specify the name of the activation group to be used when this program is called.

ENBPFCOL

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.

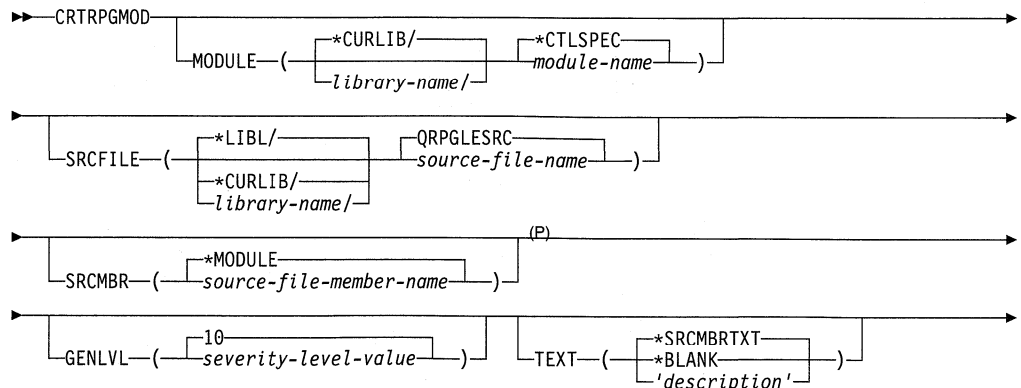
***COL**

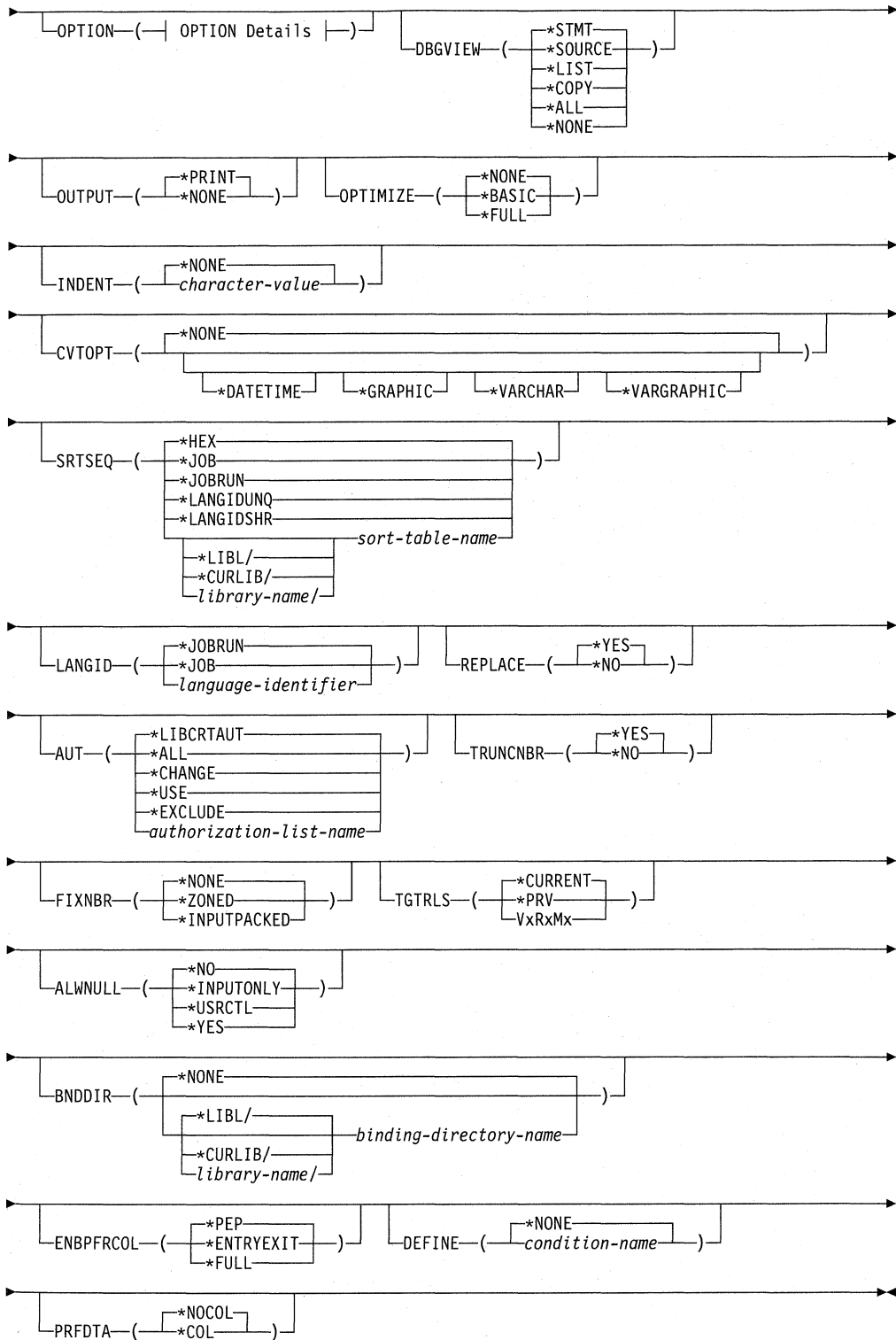
The program is enabled to collect profiling data. *COL can be specified only when the optimization level of the module is *FULL.

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

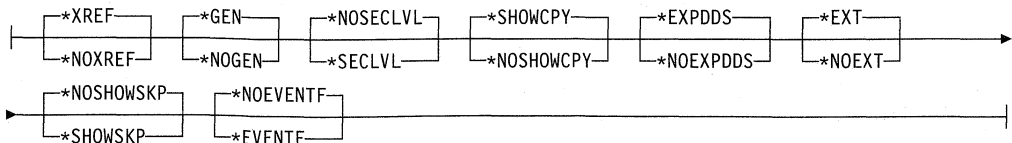




Note:

^P 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, DFACTGRP,
| USRPRF.)

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 param-
eter 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 63 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 30 summarizes the keyword specifications and their associated compiler listing information.

<i>Table 30 (Page 1 of 2). Sections of the Compiler Listing</i>		
Listing Section	OPTION¹	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
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

Table 30 (Page 2 of 2). Sections of the Compiler Listing

Listing Section	OPTION ¹	Description
Code generation errors ²		Errors (if any) which occur during code generation phase.
Binding section ²		Errors (if any) which occur during binding phase for CRTBNDRPG command

Notes:

1. The OPTION column indicates what value to specify on the OPTION parameter to obtain this information. A blank entry means that the information will always appear if OUTPUT(*PRINT) is specified.
2. The sections containing the code generation errors and binding errors appear only if there are errors. There is no option to suppress these sections.

Reading a Compiler Listing

The following text contains a brief discussion and an example of each section of the compiler listing. The sections are presented in the order in which they appear in a listing.

Prologue

The prologue section summarizes the command parameters and their values as they were processed by the CL command analyzer. If *CURLIB or *LIBL was specified, the actual library name is listed. Also indicated in the prologue is the effect of overrides. Figure 209 on page 425 illustrates how to interpret the Prologue section of the listing for the program MYSRC, which was compiled using the CRTRPGMOD command.

```

5769RG1 V4R2M0 980228 RN      IBM ILE RPG      MYLIB/MYSRC 1      AS400S01  98/02/30 10:03:43  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 . . . . . : QRPGLSRC 4
Library . . . . . : MYLIB
CCSID . . . . . : 37
Text 'description' . . . . . : Text specified on the Source Member
Last Change . . . . . : 98/01/19 23:22:52

Generation severity level . . . : 10
Default activation group . . . . : *NO
Compiler options . . . . . : *XREF *GEN *NOSECLVL *SHOWCPY 5
                             *EXPDDS *EXT *SHOWSKP *NOEVENTF

Debugging views . . . . . : *NONE
Output . . . . . : *PRINT
Optimization level . . . . . : *NONE
Source listing indentation . . . : '| ' 6
Type conversion options . . . . . : *DATETIME *VARGRAPHIC
Sort sequence . . . . . : *HEX
Language identifier . . . . . : *JOB RUN
Replace program . . . . . : *NO
User profile . . . . . : *USER
Authority . . . . . : *LIBCRTAUT
Truncate numeric . . . . . : *YES
Fix numeric . . . . . : *ZONED *INPUTPACKED
Target release . . . . . : V4R2M0
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

```

Figure 209. Sample Prologue for CRTBNDRPG

1 Page Heading

The page heading information includes the product information line and the text supplied by a /TITLE directive. “Customizing a Compiler Listing” on page 64 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.

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.

Note that the line numbers in the listing reflect the compiled source. For example, Program `MYSRC` has a `/COPY` statement in line 26. In the root source member, the next line is a `DOWEQ` operation. In the listing, however, the `DOWEQ` operation is on line 30. The three intervening lines shown in the listing are from the `/COPY` source member.

Figure 210 on page 427 shows the source section for `MYSRC`.

Line Number	Source Specifications	Comments	Do Num	Page Line	Change Date	Src Id	Seq Number
5769RG1	V4R2M0 980228 RN	IBM ILE	MYLIB/MYSRC	AS400S01	01/25/98 10:03:55	Page 2	
1	H DFACTGRP(*NO) ACTGRP('Srcactgrp')					980125	000500
2	H BNDDIR('SRCLIB1/BNDDIR1' : 'SRCLIB2/BNDDIR2' : 'ext.nam')					980125	000200
3	H ALTSEQ(*SRC)					980125	000100
4	H CVTOPT(*NODATETIME: *GRAPHIC) FIXNBR(*ZONED)					980125	000300
5	H TEXT('Text specified on the Control Specification')					980125	000400
-----*							
* Compiler Options in Effect: *							
-----*							
* Text 'description' :							
* Text specified on the Control Specification *							
* Generation severity level : 10 *							
* Default activation group : *NO *							
* Compiler options : *XREF *GEN *							
* *NOSECLVL *SHOWCPY *							
* *EXPDDS *EXT *							
* *NOSHOWSKP *NOEVENTF *							
* Optimization level : *NONE *							
* Source listing indentation : ' ' *							
* Type conversion options : *GRAPHIC *VARGRAPHIC *							
* Sort sequence : *HEX *							
* Language identifier : *JOB RUN *							
* 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 *							
-----*							
6	InFile IF E DISK					980125	000600
-----*							
* RPG name External name *							
* File name : INFILE MYLIB/INFILE *							
* Record format(s) : INREC INREC *							
-----*							
7	FKEYL6 IF E K DISK					980125	000700
-----*							
* RPG name External name *							
* File name : KEYL6 MYLIB/KEYL6 *							
* Record format(s) : REC1 REC1 *							
* REC2 REC2 *							
-----*							
8	FOutFile O E DISK					980125	000800
-----*							
* RPG name External name *							
* File name : OUTFILE MYLIB/OUTFILE *							
* Record format(s) : OUTREC OUTREC *							
-----*							
9	D Blue S 4 DIM(5) CTDATA PERRCD(1)					980125	000900
10	D Green S 2 DIM(5) ALT(Blue)					980125	001000
11	D Red S 4 DIM(2) CTDATA PERRCD(1)					980125	001100
12	D DSEXT1 E DS 100 PREFIX(BI_)					980125	001200

Figure 210 (Part 1 of 3). Sample Source Part of the Listing

```

*-----* 4
* Data structure . . . . . : DSEXT1 * 1
* Prefix . . . . . : BI * 1
* External format . . . . . : REC1 : MYLIB/DSEXT1 * 1
* Format text . . . . . : BUFFER INFO * 1
*-----* 1

13=D BI_FLD1          5A  EXTFLD (FLD1)          FORMAT          1  1
14=D BI_FLD2          10A EXTFLD (FLD2)          MAPPING          1  2
15=D BI_FLD3          18A EXTFLD (FLD3)          TABLE ENTRY     1  3

16=IINREC                2  1
*-----*
* RPG record format . . . . . : INREC * 2
* External format . . . . . : INREC : MYLIB/INFILE * 2
*-----* 2
17=I                    1  25  FLDA                2  2
18=I                    26  90  FLDB                2  3
19=IREC1                3  1
*-----*
* RPG record format . . . . . : REC1 * 3
* External format . . . . . : REC1 : MYLIB/KEYL6 * 3
*-----* 3
20=I                    *ISO-D  1  10  FLD12           3  2
21=I                    11  13  FLD13           3  3
22=I                    14  17  FLD14           3  4
23=I                    18  22  FLD15           3  5
24=IREC2                4  1
*-----*
* RPG record format . . . . . : REC2 * 4
* External format . . . . . : REC2 : MYLIB/KEYL6 * 4
*-----* 4
25=I                    *ISO-D  1  10  FLD22           4  2
26=I                    11  13  FLD23           4  3
27=I                    14  17  FLD24           4  4
28=I                    18  22  FLD25           4  5

Line <----- Source Specifications -----><----- Comments -----> Src Seq
Number . . .1...+...2...+<----- 26 - 35 ----->...4...+...5...+...6...+...7...+...8...+...9...+...10 Id Number
29 C          MOVE          '123'          BI_FLD1          980125          001300
30 C/COPY MYCPY          *-----* 6
* RPG member name . . . . . : MYCPY * 5
* External name . . . . . : MYLIB/QRPGLESRC(MYCPY) * 5
* Last change . . . . . : 09/30/94 11:55:20 * 5
* Text 'description' . . . . . : * 5
*-----*
7
31+C  Blue(1)  DSPLY          5 000100
32+C  Green(4) DSPLY          5 000200
33+C  Red(2)   DSPLY          5 000300
8
34 C  *in20    doweq          *OFF          001500
35 C          | READ          InRec          ----20      001600
36 C          | if           NOT *in20          001700
37 C  FLDA    | DSPLY          001800
38 C          | endif        001900
39 C          enddo          002000
40 C          write         outrec          002100
9
41 C          SETON          LR-----      002200
42 C/DEFINE ABC          980125      002300
43 C/IF DEFINED(ABC)    980125      002400
44 C          MOVEL 'x'      Y          10          002500
45 C          MOVEL 'x'      Z          10          002600
46 C/ELSE          980125      002700
10
----- C          MOVEL ' '      Y          10          002800
----- C          MOVEL ' '      Z          10          002900
47 C/ENDIF          980125      003000

```

Figure 210 (Part 2 of 3). Sample Source Part of the Listing

Line Number	Source Specifications	Comments	Do Num	Page Line	Change Date	Src Id	Seq Number
48=0	OUTREC					6	1
	* RPG record format : OUTREC					6	
	* External format : OUTREC : MYLIB/OUTFILE					6	
49=0	FLDY		100	CHAR	00	6	2
50=0	FLDZ		132	CHAR	32	6	3
	* * * * * END OF SOURCE * * * * *						

Figure 210 (Part 3 of 3). Sample Source Part of the Listing

1 Source Heading

Line Number

Starts at 1 and increments by 1 for each source or generated record. Use this number when debugging using statement numbers.

Ruler Line

This line adjusts when indentation is specified.

Do Number

Identifies the level of the structured operations. This number will not appear if indentation is requested.

Page Line

Shows the first 5 columns of the source record.

Source Id

Identifies the source (either /COPY or DDS) of the record. For /COPY members, it can be used to obtain the external member name from the /COPY member table.

Sequence Number

Shows the SEU sequence number of the record from a member in a source physical file. Shows an incremental number for records from a member in a physical file or records generated from DDS.

|
|
|

2 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.

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 of the source which is in error. Figure 211 shows an example.

```
Additional Diagnostic Messages

Msg id Sv Number Message text
*RNF7086 00      2 RPG handles blocking for file INFILE. INFDS is updated only
              when blocks of data are transferred.
*RNF7086 00      4 RPG handles blocking for file OUTFILE. INFDS is updated
              only when blocks of data are transferred.
*RNF7066 00      0 Record-Format REC1 not used for input or output.
*RNF7066 00      0 Record-Format REC2 not used for input or output.

***** END OF ADDITIONAL DIAGNOSTIC MESSAGES *****
```

Figure 211. Sample Additional Diagnostic Messages

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 212 shows an example of an Output Buffer Position table.

```
Output Buffer Positions

Line Start End Field or Constant
Number Pos Pos
80      1      3 RPTNUM
81      4     103 RPTNAME
82     104     203 RPTADDR
83     204     209 AMOUNT
84     210     219 DUEDATE

***** END OF OUTPUT BUFFER POSITION *****
```

Figure 212. 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 213 shows an example.

/ Copy Members								
Line	Src	RPG name	<----- External name ----->			CCSID	<- Last change ->	
Number	Id		Library	File	Member		Date	Time
26	5	MYCPY	MYLIB	QRPGLESRC	MYCPY	37	95/12/30	11:55:20
***** END OF / COPY MEMBERS *****								

Figure 213. 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 214 on page 432.

```

Compile Time Data
27 **                                     940609      001800
*-----*
* Alternate Collating Sequence Table Data: *
*-----*
31 ALTSEQ      1122ACAB4B7C36F83A657D73      940609      001900
*-----*
* Alternate Collating Sequence Table:      *
* Number of characters with an altered sequence . . . . . : 6 1 *
* 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_ *
*-----*
32 **                                     000000      002000
*-----*
* Array . . . : BLUE 4      Alternating Array . . . : GREEN *
*-----*
33 1234ZZ                                     000000      002100
34 ABCDYY                                     000000      002200
35 5432XX                                     000000      002300
36 EDCBWW                                     000000      002400
*RNF8042 00      37 Not enough source records provided to load array or table.
37 **                                     940608      002500
*-----*
* Array . . . : RED *
*-----*
38 3861                                     940608      002600
39 TKJL                                     940608      002700
* * * * * END OF COMPILE TIME DATA * * * * *

```

Figure 214. 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 215 shows an example.

Key Field Information			
File	Internal	External	Attributes
Record	field name	field name	
2 KEYL6			
Common Keys:			
			DATE *ISO- 10
			CHAR 3
REC1			
	FLD12		DATE *ISO- 10
	FLD13		CHAR 3
	FLD15		CHAR 5
REC2			
	FLD22		DATE *ISO- 10
	FLD23		CHAR 3
***** END OF KEY FIELD INFORMATION *****			

Figure 215. 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 216 on page 434 shows an example for the module TRANSRPT, which has two subprocedures.

Cross Reference					
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	A(100)	59	81		
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)			
DAYSULATE	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			
***** END OF CROSS REFERENCE *****					

Figure 216. Sample Cross-Reference Table

External References List

The External References section lists the external procedures and fields which are required from or available to other modules at bind time. This section is shown whenever the source contains statically bound procedures, imported Fields, or exported fields.

The statically bound procedures portion contains the procedure name, and the references to the name on a CALLB operation or %PADDR built-in function, 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 217 on page 435 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
***** END OF EXTERNAL REFERENCES *****

```

Figure 217. 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 218 shows an example.

```

                M e s s a g e   S u m m a r y
Msg id Sv Number Message text
*RNF7031 00    12 The name or indicator is not referenced.
*RNF7066 00    2 Record-Format name of Externally Described file is not
used.
*RNF8042 00    1 Not enough source records provided to load array or table.
***** END OF MESSAGE SUMMARY *****

```

Figure 218. 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 219 on page 436 shows an example.

```

                F i n a l   S u m m a r y

Message Totals:

Information (00) . . . . . : 17
Warning    (10) . . . . . :  0
Error      (20) . . . . . :  0
Severe Error (30+) . . . . . :  0
-----
Total . . . . . : 17

Source Totals:

Records . . . . . : 50
Specifications . . . . . : 39
Data records . . . . . : 7
Comments . . . . . : 0

        * * * * *   E N D   O F   F I N A L   S U M M A R Y   * * * * *

Program MYSRC placed in library MYLIB. 00 highest severity. Created on 95/12/30 at 17:44:55.

        * * * * *   E N D   O F   C O M P I L A T I O N * * * * *

```

Figure 219. Sample Final Summary

Code Generation and Binding Errors

Following the final summary section, you may find a section with code generation errors and/or binding errors.

The code generation error section will appear only if errors occur while the compiler is generating code for the module object. Generally, this section will not appear. The binding errors section will appear whenever there are messages arising during the binding phase of the CRTBNDRPG command. A common error is the failure to specify the location of *all* the external procedures and fields referenced in the source at the time the CRTBNDRPG command was issued.

Bibliography

For additional information about topics related to ILE RPG programming on the AS/400 system, refer to the following IBM AS/400 publications:

- *ADTS/400: Application Development Manager User's Guide*, SC09-2133, describes creating and managing projects defined for the Application Development Manager feature as well as using the program to develop applications.
- *ADTS/400: Programming Development Manager*, SC09-1771, 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/400: Source Entry Utility*, SC09-2605, provides information about using the Application Development ToolSet for AS/400 Source Entry Utility (SEU) to create and edit source members. The manual explains how to start and end an SEU session and how to use the many features of this full-screen text editor. The manual contains examples to help both new and experienced users accomplish various editing tasks, from the simplest line commands to using pre-defined prompts for high-level languages and data formats.
- *Application Display Programming*, SC41-5715, provides information about:
 - Using DDS to create and maintain displays for applications;
 - Creating and working with display files on the system;
 - Creating online help information;
 - Using UIM to define panels and dialogs for an application;
 - Using panel groups, records, or documents
- *Backup and Recovery – Advanced*, SC41-4305, 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, provides a wide-ranging discussion of AS/400 programming topics including a general discussion on objects and libraries, CL programming, controlling flow and communicating between programs, working with objects in CL programs, and creating CL programs. Other topics include predefined and impromptu messages and message handling, defining and creating user-defined commands and menus, application testing, including debug mode, breakpoints, traces, and display functions.
- *CL Reference*, SC41-5722, provides a description of the AS/400 control language (CL) and its OS/400 commands. (Non-OS/400 commands are described in the respective licensed program publications.) Also provides an overview of *all* the CL commands for the AS/400 system, and it describes the syntax rules needed to code them.
- *Communications Management*, SC41-5406, provides information about work management in a communications environment, communications status, tracing and diagnosing communications problems, error handling and recovery, performance, and specific line speed and subsystem storage information.
- *Data Management*, SC41-5710, provides information about using files in application programs. Includes information on the following topics:
 - Fundamental structure and concepts of data management support on the system
 - Overrides and file redirection (temporarily making changes to files when an application program is run)
 - Copying files by using system commands to copy data from one place to another
 - Tailoring a system using double-byte data
- *DB2 for AS/400 Database Programming*, SC41-5701, provides a detailed discussion of the AS/400 database organization, including information on how to create, describe, and update database files on the system. Also describes how to define files to the system using AS/400 data description specifications (DDS) keywords.
- *DB2 for AS/400 SQL Programming*, SC41-5611, provides information about how to use DB2 for AS/400 Query Manager and SQL Development kit licensed program. Shows how to access data in a database library and prepare, run, and test an application program that contains embedded SQL state-

ments. Contains examples of SQL statements and a description of the interactive SQL function.

Describes common concepts and rules for using SQL statements in COBOL/400®, ILE COBOL, PL/I, ILE C, FORTRAN/400®, RPG/400, ILE RPG, and REXX.

- *DB2 for AS/400 SQL Reference*, SC41-5612, provides information about how to use Structured Query Language DB2 for AS/400 statements and gives details about the proper use of the statements. Examples of statements include syntax diagrams, parameters, and definitions. A list of SQL limits and a description of the SQL communication area (SQLCA) and SQL descriptor area (SQLDA) are also provided.
- *DDS Reference*, SC41-5712, provides detailed descriptions for coding the data description specifications (DDS) for file that can be described externally. These files are physical, logical, display, print, and intersystem communication function (ICF) files.
- *Distributed Data Management*, SC41-5307, provides information about remote file processing. It describes how to define a remote file to OS/400 distributed data management (DDM), how to create a DDM file, what file utilities are supported through DDM, and the requirements of OS/400 DDM as related to other systems.
- *Experience RPG IV Multimedia Tutorial*, SK2T-2700, is an interactive self-study program explaining the differences between RPG III and RPG IV and how to work within the new ILE environment. An accompanying workbook provides additional exercises and doubles as a reference upon completion of the tutorial. ILE RPG code examples are shipped with the tutorial and run directly on the AS/400.
- *GDDM Programming Guide*, SC41-3717, provides information about using OS/400 graphical data display manager (GDDM) to write graphics application programs. Includes many example programs and information to help users understand how the product fits into data processing systems.
- *GDDM Reference*, SC41-3718, provides information about using OS/400 graphical data display manager (GDDM) to write graphics application programs. This manual provides detailed descriptions of all graphics routines available in GDDM. Also provides information about high-level language interfaces to GDDM.
- *ICF Programming*, SC41-5442, provides information needed to write application programs that use AS/400 communications and the OS/400 intersystem communications function (OS/400-ICF). Also contains information on data description specifications (DDS) keywords, system-supplied formats, return codes, file transfer support, and program examples.
- *IDDU Use*, SC41-5704, describes how to use the AS/400 interactive data definition utility (IDDU) to describe data dictionaries, files, and records to the system. Includes:
 - An introduction to computer file and data definition concepts
 - An introduction to the use of IDDU to describe the data used in queries and documents
 - Representative tasks related to creating, maintaining, and using data dictionaries, files, record formats, and fields
 - Advanced information about using IDDU to work with files created on other systems and information about error recovery and problem prevention.
- *ILE C for AS/400 Programmer's Guide*, SC09-2515, provides information on how to develop applications using the ILE C for AS/400 language. It includes information about creating, running and debugging programs. It also includes programming considerations for interlanguage program and procedure calls, locales, handling exceptions, database, externally described and device files. Some performance tips are also described. An appendix includes information on migrating source code from EPM C/400® or System C/400 to ILE C.
- *ILE COBOL for AS/400 Programmer's Guide*, SC09-2540, provides information about how to write, compile, bind, run, debug, and maintain ILE COBOL programs on the AS/400 system. 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, explains concepts and terminology pertaining to the Integrated Language Environment (ILE) architecture of the OS/400 licensed program. Topics covered include creating modules, binding, running programs, debugging programs, and handling exceptions.
- *ILE RPG for AS/400 Reference*, SC09-2508, provides information about the ILE RPG for AS/400 programming language. This manual describes, position by position and keyword by keyword, the valid entries for all RPG IV specifications, and provides a detailed description of all the operation codes and built-in functions. This manual also contains information on the RPG logic cycle, arrays and tables, editing functions, and indicators.
- *ILE RPG for AS/400 Reference Summary*, SX09-1315, provides information about the RPG III

and RPG IV programming language. This manual contains tables and lists for all specifications and operations in both languages. A key is provided to map RPG III specifications and operations to RPG IV specifications and operations.

- *Printer Device Programming*, SC41-5713, provides information to help you understand and control printing. Provides specific information on printing elements and concepts of the AS/400 system, printer file and print spooling support for printing operations, and printer connectivity. Includes considerations for using personal computers, other printing functions such as Business Graphics Utility (BGU), advanced function printing (AFP®), and examples of working with the AS/400 system printing elements such as how to move spooled output files from one output queue to a different output queue. Also includes an appendix of control language (CL) commands used to manage printing workload. Fonts available for use with the AS/400 system are also provided. Font substitution tables provide a cross-reference of substituted fonts if attached printers do not support application-specified fonts.
- *Security - Basic*, SC41-5301, explains why security is necessary, defines major concepts, and provides information on planning, implementing, and monitoring basic security on the AS/400 system.
- *Security - Reference*, SC41-5302, tells how system security support can be used to protect the system and the data from being used by people who do not have the proper authorization, protect the data from intentional or unintentional damage or destruction, keep security information up-to-date, and set up security on the system.
- *Software Installation*, SC41-5120, provides step-by-step procedures for initial installation, installing licensed programs, program temporary fixes (PTFs), and secondary languages from IBM. This manual is also for users who already have an AS/400 system with an installed release and want to install a new release.
- *System API Reference*, SC41-5801, provides information for the experienced programmer on how to

use the application programming interfaces (APIs) to such OS/400 functions as:

- Dynamic Screen Manager
- Files (database, spooled, hierarchical)
- Message handling
- National language support
- Network management
- Objects
- Problem management
- Registration facility
- Security
- Software products
- Source debug
- UNIX®-type
- User-defined communications
- User interface
- Work management

Includes original program model (OPM), Integrated Language Environment (ILE), and UNIX-type APIs.

- *System Operation*, SC41-4203, provides information about handling messages, working with jobs and printer output, devices communications, working with support functions, cleaning up your system, and so on.
- *Basic System Operation, Administration, and Problem Handling*, SC41-5206, provides information about the system unit control panel, starting and stopping the system, using tapes and diskettes, working with program temporary fixes, as well as handling problems.
- *Tape and Diskette Device Programming*, SC41-5716, provides information to help users develop and support programs that use tape and diskette drives for I/O. Includes information on device files and descriptions for tape and diskette devices.

Index

Special Characters

- = (equal) R:402
- / (division) R:402
- /COPY statement R:12
 - conversion problems PG:389, PG:398
 - COPY debug view PG:168
 - in a conversion report PG:394
 - recognizing a compiler /COPY R:13
 - table in compiler listing PG:431
- /DEFINE R:14
- /EJECT R:11
- /ELSE R:16
- /ELSEIF condition-expression R:15
- /ENDIF R:16
- /EOF R:16
- /IF condition-expression R:15
- /SPACE R:12
- /TITLE R:11
- /UNDEFINE R:14
- <= (less than or equal) R:402
- < (less than) R:402
- <> (not equal) R:402
- \$ (fixed or floating currency symbol)
 - in body of edit word R:211
 - use in edit word R:211
 - with combination edit codes R:201
- * (asterisk)
 - in body of edit word R:211
 - with combination edit codes R:201
- * (multiplication) R:402
- * (pointer data type entry) R:272
- ** (double asterisk)
 - alternate collating sequence table R:164
 - arrays and tables R:145
 - file translation table R:105
 - for program described files R:302
 - lookahead fields R:302, R:303
- *ALL R:339
- *ALL'x..' R:120
- *ALLG'oK1K2i' R:120
- *ALLX'x1..' R:120
- *BLANK/*BLANKS R:120
- *CANCL PG:238, R:22, R:80
- *CYMD, *CMDY, and *CDMY date formats
 - description R:176
 - with MOVE operation R:440, R:545, R:565
 - with MOVEL operation R:440
 - with TEST operation R:647
- *DATE R:8
- *DAY R:8
- *DETC PG:238
 - file exception/error subroutine (INFSR) R:80
 - flowchart R:22
 - program exception/errors R:83
- *DETL PG:238
 - file exception/error subroutine (INFSR) R:80
 - flowchart R:20
 - program exception/errors R:83
- *DTAARA DEFINE R:494
- *END R:630
- *ENTRY PLIST R:590
- *EQUATE R:107
- *EXT R:506
- *FILEbb R:106
- *GETIN PG:238
 - file exception/error subroutine (INFSR) R:80
 - flowchart R:20
 - program exception/errors R:83
- *HIVAL R:120
- *IN R:59
- *IN(xx) R:59
- *INIT R:83
- *INxx R:60
- *INZSR R:26
 - See also initialization subroutine (*INZSR)
- *JOBRUN
 - date format example R:545
 - date format, DATFMT R:176
 - date separator, DATSEP R:176
 - decimal format, DECFMT R:234
 - language identifier, LANGID PG:414, R:164, R:238
 - sort sequence, SRTSEQ PG:414, R:145, R:240
 - time separator, TIMSEP R:178
- *LDA R:494
- *LIKE DEFINE R:492
- *LONGJUL date format
 - description R:176
 - with MOVE operation R:440, R:545, R:565
 - with MOVEL operation R:440
 - with TEST operation R:647
- *LOVAL R:120
- *M R:506
- *MONTH R:8
- *NOIND R:260
- *NOKEY (with CLEAR operation) R:483
- *NOKEY (with RESET operation) R:610
- *NULL R:120, R:182
- *OFL PG:238
 - file exception/error subroutine (INFSR) R:80
 - flowchart R:22
 - program exception/errors R:83

***ON/*OFF** R:120
***PDA** R:494
***PLACE** R:333
***PSSR** R:87
See also program exception/error subroutine
***ROUTINE** R:427
***START** R:630
***TERM** R:83
***TOTC** PG:238
 flowchart R:22
 program exception/errors R:80
***TOTL** PG:238
 file exception/error subroutine (INFSR) R:80
 flowchart R:22
 program exception/errors R:83
***VAR data attribute**
 output specification R:306, R:337
***YEAR** R:8
***ZERO/*ZEROS** R:120
& (ampersand)
 in body of edit word R:213
 in status of edit word R:210
 use in edit word R:210, R:213
%ABS (Absolute Value of Expression) R:352
%ADDR (Get Address of Variable)
 data types supported R:402
 description R:353
 example R:353
 omitted parameters PG:142
%ADDR debug built-in PG:206
%CHAR (Convert to Character Data) R:355
%DEC (Convert to Packed Decimal Format) R:356
%DECH (Convert to Packed Decimal Format with Half Adjust) R:356
%DECPOS (Get Number of Decimal Positions)
 description R:357
 example R:357, R:373
%EDITC (Edit Value Using an Editcode) R:358
%EDITFLT (Convert to Float External Representation) R:361
%EDITW (Edit Value Using an Editword) R:362
%ELEM (Get Number of Elements) R:363, R:402
%EOF (Return End or Beginning of File Condition) R:364
%EQUAL (Return Exact Match Condition) R:366
%ERROR (Return Error Condition) R:368
%FLOAT (Convert to Floating Format) R:369
%FOUND (Return Found Condition) R:370
%INDEX debug built-in PG:206
%INT (Convert to Integer Format) R:372
%INTH (Convert to Integer Format with Half Adjust) R:372
%LEN (Get Length) R:373
%NULLIND (Query or Set Null Indicator) R:376
%OPEN (Return File Open Condition) R:377

%PADDR (Get Procedure Address) R:378, R:402
%PARMS (Return Number of Parameters) R:379
 checking for number of parameters PG:143
%REPLACE (Replace Character String) R:381
%SCAN (Scan for Characters) R:383
%SIZE (Get Size in Bytes) R:384, R:402
%STATUS (Return File or Program Status) R:386
%STR (Get or Store Null-Terminated String) R:389
%SUBST (Get Substring)
 data types supported R:402
 description R:391
 example R:392
 use with EVAL R:391
%SUBSTR debug built-in
 changing values PG:208
 examples PG:207
%TRIM (Trim Blanks at Edges) R:393, R:402
%TRIML (Trim Leading Blanks) R:394, R:402
%TRIMR (Trim Trailing Blanks) R:395, R:402
%UNS (Convert to Unsigned Format) R:396
%UNSH (Convert to Unsigned Format with Half Adjust) R:396
 - (unary operator) R:402
 + (unary operator) R:402
 >= (greater than or equal) R:402
 > (greater than) R:402

Numerics

01-99 indicators
 displaying while debugging PG:204
 in a sample formatted dump PG:257
1P (first page) indicator
 conditioning output R:329, R:333
 general description R:46
 restrictions R:46
 setting R:62
 with initialization subroutine (*INZSR) R:26

A

abnormal program/procedure end PG:155
absolute notation R:123, R:271
absolute value R:352
access path
 example of PG:288
 for externally described DISK file PG:282
 for indexed file PG:288
ACQ (acquire) operation code R:433, R:452
ACTGRP keyword R:229
ACTGRP parameter
 CRTBNDRPG command PG:58, PG:418
 CRTPGM command PG:82
 specifying PG:110
 specifying on control specifications R:229
 using PG:60

activation group

- *NEW PG:155
- *CALLER PG:111
 - running in OPM default PG:111
 - specifying PG:111
- definition PG:109
- deleting PG:112
- identifying PG:82, PG:110
- managing PG:109
- named PG:82
 - deleting PG:110
 - specifying PG:110
- *NEW PG:82
 - ending PG:110
 - specifying PG:110
- OPM default PG:111
- QILE PG:82, PG:110
- role in exception handling PG:218

activation, program PG:109

ADD operation code R:418, R:453

add records

- file description specifications entry (A) R:247
- output specification entry (ADD) R:327

Add Reply List Entry (ADDRPLYE) command

- adding to system reply list PG:108

ADDUR (add duration) operation code

- adding dates R:431, R:454
- general discussion R:430
- unexpected results R:432

adding date-time durations R:431, R:454

adding factors R:453, R:454

adding objects to a debug session PG:172

additional diagnostic messages section of compiler

listing PG:430

address

- of based variable R:353
- of procedure pointer R:378

ADDRPLYE command

- See Add Reply List Entry (ADDRPLYE) command

ADTS PG:13

ALIGN keyword

- aligning subfields R:124
- description R:274
- float fields R:168
- integer fields R:168
- unsigned fields R:170

alignment

- of basing pointers R:182
- of integer fields R:170

alignment of forms R:237

ALLOC (allocate storage) operation code PG:113, R:436, R:456

allocate storage (ALLOC) operation code R:456

allocating storage R:456

allocating storage for a run-time array PG:120

ALT keyword R:274

altering overflow logic R:28

alternate collating sequence

- changing collating sequence R:164
- coding form R:163
- control specification entry R:163
- control specification keyword ALTSEQ R:275
- control-specification keyword ALTSEQ R:229
- debug considerations PG:184
- definition specification keyword ALTSEQ R:163
- input record format R:164
- operations affected R:163

alternating format (arrays and tables)

- definition specification keyword ALT R:274
- example R:149

ALTSEQ keyword

- **ALTSEQ R:144, R:164
- changing collating sequence R:163
- control-specification description R:229
- definition specification description R:275
- specifying in source R:164

ALWNULL keyword R:230

ALWNULL parameter

- See also null value support
- CRTBNDRPG command PG:58, PG:417
- CRTRPGMOD command PG:74
- specifying on control specifications R:230

ampersand (&)

- in body of edit word R:213
- in status of edit word R:210
- use in edit word R:210, R:213

analyzing your conversion PG:393

AND relationship

- calculation specifications R:317
- input specifications R:305
- output specifications R:327, R:338
- conditioning indicators R:329

ANDxx operation code R:427, R:445, R:457

API

- See application programming interface (API)

apostrophe

- use with edit word R:214
- use with output constant R:337

application design

- See creating programs

Application Development Manager PG:13

Application Development ToolSet PG:13

Application Dictionary Services PG:14

application programming interface (API)

- calling non-bindable PG:130
- parsing system built-in names R:426
- QMHSNDPM PG:374
- Retrieve Message (QMHRVTM) API PG:158

area parameter for SPECIAL PLIST PG:328

arithmetic built-in functions

- %ABS (Absolute Value of Expression) R:352

arithmetic operation codes R:453

See also calculation operations
 ADD R:418
 DIV (divide) R:418, R:497
 ensuring accuracy R:419
 general information R:418
 integer arithmetic R:420
 MULT (multiply) R:418, R:575
 MVR (move remainder) R:418, R:576
 performance considerations R:420
 SQRT (square root) R:418, R:638
 SUB (subtract) R:418, R:639
 XFOOT (summing the elements of an array) R:418, R:666
 Z-ADD (zero and add) R:418, R:453, R:669
 Z-SUB (zero and subtract) R:418, R:670

array

alternating
 definition R:149
 examples R:149
 binary format R:165
 combined array file R:144, R:245
 compile-time
 arrangement in source program R:147
 definition of R:144
 conversion problems PG:402
 creating input records R:145
 definition R:141
 differences from table R:141
 displaying while debugging PG:202
 editing R:154
 elements R:141
 end position R:334
 even number of digits R:293
 file
 description of R:246
 file description specifications entry R:246
 file name (when required on file description specifications) R:244
 float format R:166
 initialization of R:148
 loading PG:403
 compile-time R:144
 from more than one record R:144
 from one record R:143
 LOOKUP operation code R:538
 prerun-time R:147
 run-time R:143
 moving (MOVEA operation code) R:559
 name
 in compare operation codes R:428
 output specifications R:332
 rules for R:146
 number of elements R:278, R:363
 order in source program R:147
 output R:154

array (continued)

packed format R:169
 prerun-time arrays PG:403, R:144
 rules for loading R:147
 referring to in calculations R:152
 run-time
 definition of R:142
 rules for loading R:143
 with consecutive elements R:144
 with scattered elements R:143
 searching with an index R:151
 searching without an index R:150
 size of R:384
 square root (SQRT) operation code R:638
 summing elements of (XFOOT) operation code R:666
 to file name R:262
 types R:141
 XFOOT operation code R:666
array operations R:421
 general information R:421
 LOOKUP (look up) R:421, R:538
 MOVEA (move array) R:421, R:559
 SORTA (sort an array) R:421, R:636
 XFOOT (summing the elements of an array) R:421, R:666

arrival sequence access path PG:282

ASCEND keyword R:275

ascending sequence

definition specification keyword ASCEND R:275
 file description specifications entry R:248

assigning match field values (M1-M9) R:98

asterisk fill

in body of edit word R:203
 with combination edit codes R:203

ATTR debug command

definition PG:164
 example PG:210
 using PG:210

audit file PG:381

See also log file

AUT keyword R:230

AUT parameter

CRTBNDRPG command PG:58, PG:415
 CRTTRPGMOD command PG:74
 specifying on control specifications R:230

authority to commands PG:xxi

auto report program

converting to ILE RPG PG:389

automatic storage R:113

avoiding a loop in an error subroutine PG:236

B

BASED keyword R:276

based variable

address of R:353
 and basing pointers R:182, R:183
 defining R:276

begin a select group (SELECT) operation

code R:623

begin/end entry in procedure specification R:343

BEGSR (beginning of subroutine) operation

code R:448, R:458

behavior of bound ILE RPG modules PG:80

behavioral differences between OPM RPG/400 and ILE RPG PG:373

bibliography PG:437, R:679

binary field

definition R:165
 EXTBININT keyword R:235
 input specifications R:166, R:307
 output specifications R:166, R:336

binary format

definition R:165
 input field R:307
 input field specification R:165
 output field R:336
 output field specification R:165

binary operations

data types supported R:402
 precedence of operators R:399

binary operators R:459, R:460

binary relative-record number R:253

bindable APIs

calling conventions PG:157
 CEE4ABN PG:156
 CEECRHP (Create Heap) PG:20, PG:120
 CEECZST (Reallocate Storage) PG:20
 CEEDSHP (Discard Heap) PG:20, PG:120
 CEEFRST (Free Storage) PG:20
 CEEGTST (Get Heap Storage) PG:20, PG:120
 CEEHDLR (Register ILE Condition Handler) PG:239
 CEEHDLU (Unregister ILE Condition Handler) PG:239
 CEERTX (Register Call Stack Entry Termination User Exit Procedure) PG:244
 CEETREC PG:156
 CEETSTA (Check for Omitted Argument) PG:142
 CEEUTX (Call Stack Entry Termination User Exit Procedure) PG:244
 Create Heap (CEECRHP) PG:20, PG:120
 description PG:157
 Discard Heap (CEEDSHP) PG:20, PG:120
 Free Storage (CEEFRST) PG:20
 Get Descriptive Information About a String Argument (CEESGI) PG:141
 Get Heap Storage (CEEGTST) PG:20, PG:120
 overview PG:20
 passing operational descriptors to PG:140
 Reallocate Storage (CEECZST) PG:20

bindable APIs (continued)

Retrieve Operational Descriptor Information (CEEDOD) PG:141
 returning from a procedure PG:156
 sample coding PG:157

binder language

example PG:97
 reasons for using PG:92

binder listing

as maintenance resource PG:86
 basic PG:100
 creating PG:85
 determining exports in service program PG:91
 sections of PG:85

binding

after modifying a module PG:86
 definition PG:81
 modules into a program PG:81
 service program to a program PG:98

binding errors in compiler listing PG:436

binding multiple modules PG:84

bit operation codes R:421

bit operations

BITOFF (set bits off) R:421, R:459
 BITON (set bits on) R:421, R:460
 general information R:421
 TESTB (test bit) R:421, R:649

bit testing (TESTB) R:649

BITOFF (set bits off) operation code R:421, R:459

BITON (set bits on) operation code R:421, R:460

blank after

definition R:334
 output specifications R:334

blanks, removing from a string R:393

BLOCK keyword R:255

blocking/unblocking records PG:287, R:77

BNDDIR keyword R:231

BNDDIR parameter on CRTBNDRPG

CRTBNDRPG command PG:58, PG:418
 CRTRPGMOD command PG:74
 specifying on control specifications R:231
 static binding PG:60

body (of an edit word) R:210

branching operations R:422

CABxx (compare and branch) R:422, R:462
 ENDSR (end of subroutine) R:512
 EXCEPT (calculation time output) R:515
 general description R:422
 GOTO (go to) R:422, R:524
 ITER (iterate) R:422, R:531
 LEAVE (leave a structured group) R:422, R:536
 TAG (tag) R:422, R:646

branching within logic cycle R:462

BREAK debug command

definition PG:164
 example PG:183

BREAK debug command *(continued)*

using PG:179, PG:181, PG:185

breakpoints

conditional

setting and removing for job PG:181

setting and removing for thread PG:187

removing all PG:187

setting and removing PG:178

setting using statement numbers PG:185

testing PG:178

unconditional

setting and removing for job PG:179

setting and removing for thread PG:180

browsing a compiler listing using SEU PG:68

built-in functions

%ADDR PG:142

arithmetic

%ABS (Absolute Value of Expression) R:352

data conversion

%CHAR (Convert to Character Data) R:355

%DEC (Convert to Packed Decimal

Format) R:356

%DECH (Convert to Packed Decimal Format with Half Adjust) R:356

%EDITC (Edit Value Using an Editcode) R:358

%EDITFLT (Convert to Float External Representation) R:361

%EDITW (Edit Value Using an Editword) R:362

%FLOAT (Convert to Floating Format) R:369

%INT (Convert to Integer Format) R:372

%INTH (Convert to Integer Format with Half Adjust) R:372

%UNS (Convert to Unsigned Format) R:396

%UNSH (Convert to Unsigned Format with Half Adjust) R:396

data information

%DECPOS (Get Number of Decimal Positions) R:357

%ELEM (Get Number of Elements) R:363

%LEN (Get Length) R:373

%SIZE (Get Size in Bytes) R:384

data types supported R:402

editing

%EDITC (Edit Value Using an Editcode) R:358

%EDITFLT (Convert to Float External Representation) R:361

%EDITW (Edit Value Using an Editword) R:362

example R:347

exception/error handling

%ERROR (Return Error Condition) R:368

%STATUS (Return File or Program Status) R:386

feedback

%EOF (Return End or Beginning of File Condition) R:364

%EQUAL (Return Exact Match Condition) R:366

%ERROR (Return Error Condition) R:368

built-in functions *(continued)*

feedback *(continued)*

%FOUND (Return Found Condition) R:370

%NULLIND (Query or Set Null Indicator) R:376

%OPEN (Return File Open Condition) R:377

%PARMS (Return Number of Parameters) R:379

%STATUS (Return File or Program

Status) R:386

list of R:352

on definition specification R:267

pointer

%ADDR (Get Address of Variable) R:353

%PADDR (Get Procedure Address) R:378

string

%REPLACE (Replace Character String) R:381

%SCAN (Scan for Characters) R:383

%STR (Get or Store Null-Terminated String) R:389

%SUBST (Get Substring) R:391

%TRIM (Trim Blanks at Edges) R:393

%TRIML (Trim Leading Blanks) R:394

%TRIMR (Trim Trailing Blanks) R:395

syntax R:347

table of R:349

C

CABxx (compare and branch) operation

code R:422, R:427, R:462

calculating R:219

calculating date durations R:431

calculating date-time durations R:641

calculation

indicators

AND/OR relationship R:54, R:317

conditioning R:54, R:315

control level R:52, R:317

resulting R:44, R:321

operation codes R:318, R:322

summary of R:413

specifications

entries for factor 1 R:318

entries for result field R:320

relationship between positions 7 and 8 and 9-11 R:317

summary of R:315

summary of operation codes R:413

subroutines

BEGSR (beginning of subroutine) operation code R:458

coding of R:448

ENDSR (end of subroutine) operation

code R:512

EXSR (invoke subroutine) operation code R:519

SR identifier R:317

calculation specifications

control level R:316
 decimal positions R:320
 extended factor 2 field continuation R:224
 factor 1 R:318
 factor 2 R:320
 field length R:320
 general description PG:3, R:315
 indicators R:318
 operation R:318, R:322
 operation extender R:318, R:322
 program-described WORKSTN file PG:340
 result field R:320
 resulting indicators R:321
 summary of R:315

calculation-time output (EXCEPT) operation code R:515

CALL (call a program) operation code

call operations R:422
 description R:464
 in a conversion report PG:394
 using PG:150

CALL CL command

example passing parameters PG:104
 passing parameters PG:103
 running a program PG:103

call operations

CALL (call a program) R:422, R:464
 CALLB (call a bound procedure) R:422, R:465
 calling programs PG:150
 CALLP (call a prototyped procedure) R:422, R:466
 DSPPGMREF PG:150
 fixed-form call PG:150
 FREE (deactivate a program) R:422
 free-form call PG:135, PG:136
 general description R:422
 PARM (identify parameters) R:422, R:587
 parsing program names R:424
 parsing system built-in names R:426
 PLIST (identify a parameter list) R:422, R:590
 query names of called procedures PG:150
 RETURN (return to caller) R:422, R:616
 special routines PG:158
 using PG:135

call stack PG:131, PG:218

Call Stack Entry Termination User Exit Procedure (CEEUTX) PG:244

CALLB (call a bound procedure) operation code

call operations R:422
 calling programs PG:150
 description R:465
 using PG:150

***CALLER PG:111**

calling a graphics routine PG:158

calling programs/procedures

abnormal program/procedure end PG:155

calling programs/procedures (continued)

call stack PG:131
 calling bindable APIs PG:157
 calling graphics PG:158
 calling procedures PG:130
 calling programs PG:130
 calling special routines PG:158
 fixed-form call PG:150
 free-form call PG:135, PG:136
 interlanguage calls PG:149
 normal program/procedure end PG:154
 operational descriptors R:424
 overview PG:129
 parameter passing methods PG:137
 prototyped call R:423
 recursive calls PG:132
 returning from a called program or procedure PG:153
 returning values PG:136
 returning without ending PG:155
 static calls PG:130
 using the CALL operation PG:150
 using the CALLB operation PG:150
 within ILE PG:19

CALLP (call a prototyped program or procedure) operation code

call operations R:422
 description R:466
 using PG:135
 with expressions R:397

cancel handler PG:217

CEERTX (Register Call Stack Entry Termination User Exit Procedure) PG:244
 CEEUTX (Call Stack Entry Termination User Exit Procedure) PG:244
 example PG:245
 using PG:244

CASxx (conditionally invoke subroutine) operation code R:427, R:448, R:469

CAT (concatenate two character strings) operation code R:444, R:471

CEE4ABN PG:156

CEECRHP (Create Heap) bindable API PG:20, PG:120

CEECZST (Reallocate Storage) bindable API PG:20

CEEDOD (Retrieve Operational Descriptor Information) PG:94

example PG:141

operational descriptors PG:141

CEEDSHP (Discard Heap) bindable API PG:20, PG:120

CEEFRST (Free Storage) bindable API PG:20

CEEGTST (Get Heap Storage) bindable API PG:20, PG:120

CEEHDLR (Register ILE Condition Handler) PG:239

ILE RPG for AS/400 Combined Index

CEEHDLU (Unregister ILE Condition Handler) PG:239

CEERTX (Register Call Stack Entry Termination User Exit Procedure) PG:244

CEESGI (Get Descriptive Information About a String Argument) PG:141

CEETREC PG:156

CEETSTA (Check for Omitted Argument) PG:142

CEEUTX (Call Stack Entry Termination User Exit Procedure) PG:244

century formats

description R:176

with MOVE operation R:440, R:545, R:565

with MOVE operation R:440

with TEST operation R:647

CHAIN (random retrieval from a file based on record number or key value) ope R:474

CHAIN (random retrieval from a file based on record number or key value) operation code R:433

Change Module (CHGMOD) command PG:88

removing observability PG:88

Change Program (CHGPGM) command

optimization parameters PG:88

removing observability PG:88

Change Service Program (CHGSRVPGM)

command PG:100

changing a module PG:86

changing a program PG:86

changing a service program PG:93

changing between character fields and numeric fields R:438

changing field values while debugging PG:208

changing optimization level

See also optimization

of a program or module PG:87

changing the debug view of a module PG:176

character

allowed formats

description R:159

fixed length R:160

indicator R:162

variable length R:160

collating sequence R:164

converting to R:355

definition specification R:272

displaying while debugging PG:205

in record identification code R:305

indicator literals R:115

keys in record address type R:251

literals R:115

replace or insert string R:381

valid set R:3

CHECK (check) operation code R:444, R:477

Check for Omitted Argument (CEETSTA) PG:142

checking for the number of passed parameters PG:143

checking, level PG:270

CHECKR (check reverse) operation code R:444, R:480

CHGMOD command

See Change Module (CHGMOD) command

CHGPGM command

See Change Program (CHGPGM) command

CHGSRVPGM

See Change Service Program (CHGSRVPGM) command

CL commands

Add Program (ADDPGM) PG:172

additional service program commands PG:93

ADDRPLYE PG:108

authority PG:xxi

CALL PG:103

Change Job (CHGJOB) command R:46

Change Module (CHGMOD) PG:88

CHGPGM PG:88

commonly used commands PG:12

Create Job Description (CRTJOB) command R:46

CRTPGM command PG:82

CRTRPGMOD PG:74

CVTRPGSRC PG:383

Display Module Source (DSPMODSRC) PG:171,

PG:172, PG:173, PG:175

DSPMOD PG:150

DSPPGMREF PG:150

End Debug (ENDDBG) PG:169

module-related PG:80

MONMSG PG:247

program-related PG:85

RCLACTGR PG:110

RCLRSC PG:112

reading syntax diagrams PG:405

Remove Program (RMVPGM) PG:172

Start Debug (STRDBG) PG:169, PG:171

UPDPGM PG:87

using PG:405

WRKRPLYE PG:109

clear command PG:334

CLEAR debug command

definition PG:164

removing all PG:187

using PG:179, PG:182, PG:186

CLEAR operation code R:114, R:435, R:483

CLOSE (close files) operation code R:433, R:487

closing a file R:487

code conversion constraints PG:398

code generation errors in compiler listing PG:436

code part

in record identification code for program described

file R:304

coding subroutines R:448

collating sequence

See also alternate collating sequence

collating sequence (*continued*)
 alternate R:163
 EBCDIC R:675
 normal R:163

combination edit codes (1-4, A-D, J-Q) R:202

combined file PG:341
 description R:245

command attention (CA) keys PG:332
 corresponding indicators R:52

command definition PG:108

command function (CF) keys PG:332
 corresponding indicators R:52

comments
 on array input records R:145

COMMIT (commit) operation code R:433
 commitment control PG:309
 description R:488
 system considerations PG:309
 with multiple devices PG:309

COMMIT keyword
 description R:256

commitment control PG:307
 COMMIT operation PG:309
 conditional PG:311, R:256
 example PG:310
 in program cycle PG:312
 locks PG:308
 scoping PG:308
 specifying files PG:309
 starting and ending PG:307

common entries to all specifications R:220

communication
 accessing other programs and systems PG:331

COMP (compare) operation code R:427, R:489

compare and branch (CABxx) operation code R:462

compare operations
 ANDxx (and) R:427, R:457
 CABxx (compare and branch) R:427, R:462
 CASxx (conditionally invoke subroutine) R:427, R:469
 COMP (compare) R:427, R:489
 DOU (do until) R:427, R:500
 DOUxx (do until) R:427, R:501
 DOW (do while) R:427, R:503
 DOWxx (do while) R:427, R:504
 EVAL (evaluate) R:513
 general information R:427
 IF (if) R:427, R:526
 IFxx (if/then) R:427, R:527
 ORxx (or) R:427, R:584
 WHEN (when true then select) R:427
 When (When) R:660
 WHENxx (when true then select) R:427, R:661

comparing bits R:649

comparing factors R:462, R:489
See also CABxx operation code

compatibility differences between OPM RPG/400 and ILE RPG PG:373

compilation errors, correcting PG:66

compile time array or table
See also array
 definition specification keyword CTDATA R:277
 general description R:144
 number of elements per record R:293
 rules for loading R:144
 section in compiler listing PG:431
 specifying external data type R:279

compiler
 directives R:11

compiler directives
 /COPY R:12
 /EJECT R:11
 /SPACE R:12
 /TITLE R:11
 changing a listing heading PG:64
 conditional compilation directives
 /DEFINE R:14
 /ELSE R:16
 /ELSEIF condition-expression R:15
 /ENDIF R:16
 /EOF R:16
 /IF condition-expression R:15
 /UNDEFINE R:14

compiler listing
 additional diagnostic messages PG:68
 browsing using SEU PG:68
 coordinating listing options with debug view options PG:69
 correcting compilation errors PG:66
 correcting run-time errors PG:68
 default information PG:63
 in-line diagnostic messages PG:67
 indenting structured operations PG:65
 obtaining PG:63
 reading PG:423
 sample listing PG:424
 sections of PG:63, PG:424
 specifying the format of PG:64
 using PG:63
 using as documentation PG:69

compiling
 creating modules PG:73
 differences between ILE RPG and OPM RPG/400 PG:373
 in ILE PG:17
 using CRTBNDRPG command PG:57

composite key operation codes
 KFLD (define parts of a key) R:534
 KLIST (define a composite key) R:534

- compressing an object** PG:88
- concatenate two strings (CAT) operation**
 - code R:471
- condition expressions** R:15
- condition handler** PG:217
 - example PG:239
 - overview PG:239
 - percolate an exception PG:240
 - recursive calls PG:239
 - registering PG:239
- conditional breakpoint**
 - definition PG:178
 - setting PG:182
 - setting and removing for job PG:181
 - setting and removing for thread PG:187
 - using statement numbers PG:185
- conditional commitment control, specifying** PG:311
- conditional file open** R:257, R:265
- conditionally invoke subroutine (CASxx) operation**
 - code R:469
- conditioning indicators**
 - calculation
 - general description R:52
 - positions 7 and 8 R:52
 - positions 9 through 11 R:53
 - specification of R:318
 - file
 - general description R:48
 - rules for R:48
 - general description R:48
- conditioning output**
 - explanation of R:56
 - for fields of a record R:332
 - for records R:329
 - overflow indicators PG:317
- consecutive processing** PG:292
- CONST keyword**
 - description R:276
- constants** R:114
 - See also* edit words
 - constant/editword field continuation R:225
 - defining using CONST R:276
 - entries for factor 2 R:114
 - figurative R:120
 - *ALL'x..', *ALLX'x1..', *BLANK/*BLANKS,
 - *HIVAL/*LOVAL, *ZERO/*ZEROS,
 - *ON/*OFF R:120
 - named R:119
 - rules for use on output specification R:337
 - size of R:384
- continuation rules for specifications** R:221
- control boundary** PG:218
- control break**
 - See also* control field
 - example PG:319
 - general description R:36
- control break** (*continued*)
 - how to avoid unwanted R:38
 - on first cycle R:36
 - unwanted R:38
- control entries**
 - in output specification R:326
- control field**
 - See also* control break
 - assigning on input specifications
 - externally described file R:313
 - program described file R:308
 - general information R:37
 - overlapping R:38
 - split R:41
- control group**
 - See also* control break
 - general information R:36
- control language (CL) program**
 - See also* ILE CL
 - as module in ILE program PG:27
 - commands used with ILE RPG PG:12
 - commonly used commands PG:12
 - in OPM-compatible application PG:23
- control level (L1-L9) indicators** R:317
 - See also* conditioning calculations
 - as field record relation indicator R:49, R:310
 - as record identifying indicator R:302, R:311
 - assigning to input fields R:309, R:313
 - conditioning calculations R:315
 - conditioning output R:329
 - examples R:38, R:42
 - general description R:36
 - in calculation specification R:316
 - rules for R:37
 - setting of R:62
- control specification keywords**
 - ALTSEQ R:229
 - compile-option keywords
 - ACTGRP R:229
 - ALWNULL R:230
 - AUT R:230
 - BNDDIR R:231
 - compiler listing example PG:424
 - CVTOPT R:232
 - DFACTGRP R:234
 - ENBPFRCOL R:235
 - FIXNBR R:236
 - GENLVL R:237
 - INDENT R:238
 - LANGID R:238
 - OPTIMIZE R:239
 - OPTION R:239
 - PRFDTA R:240
 - SRTSEQ R:240
 - TEXT R:241
 - TRUNCNBR R:241
 - USRPRF R:241

control specification keywords *(continued)*

COPYNEST R:231
 COPYRIGHT R:231
 CURSYM R:232
 DATEDIT R:233
 DATFMT R:233
 DEBUG R:233
 DECEDIT R:234
 DFTNAM R:235
 EXPROPTS R:235
 EXTBININT R:235
 FLTDIV R:237
 FORMSALIGN R:237
 FTRANS R:237
 NOMAIN R:238
 TIMFMT R:241

control specifications

continuation line R:223
 conversion considerations PG:382
 data area (DFTLEHSPEC) R:227
 data area (RPGLEHSPEC) R:227
 example PG:7
 form type R:228
 general description PG:3, R:227

control-record format, subfile PG:335

controlling input of program R:29

controlling spacing of compiler listing R:12

Conversion Aid

See converting to RPG IV

conversion reports

obtaining PG:389
 sections of PG:394
 using PG:394

conversion, analyzing PG:393

converting a character to a date field R:441

converting to RPG IV

analyzing your conversion PG:393
 constraints PG:382
 conversion problems PG:398
 converting PG:382
 converting all file members PG:388
 converting auto report source members PG:389
 converting some file members PG:388
 converting source from a data file PG:390
 converting source members with embedded SQL PG:390
 CVTRPGSRC command PG:383
 example PG:390
 file and member names PG:381
 file considerations PG:380
 file record length PG:380
 log file PG:381
 obtaining conversion reports PG:389
 overview PG:379
 performing a trial conversion PG:388
 requirements PG:381

converting to RPG IV *(continued)*

using a conversion error report PG:394
 using the log file PG:396
 valid source member types PG:380

CoOperative Development/400 (CODE/400)

description PG:14
 event file PG:411

coordinating listing options with debug view options PG:69

COPYNEST keyword R:231

COPYRIGHT keyword R:231

correcting compilation errors PG:66

correcting run-time errors PG:68

CR (negative balance symbol)

with combination edit code R:202
 with edit words R:213

Create Bound RPG Program (CRTBNDRPG)

command

and ILE PG:18
 coordinating listing options with debug view PG:69
 creating programs PG:57
 default parameter values PG:58
 examples
 OPM-compatible program PG:61
 program for source debugging PG:59
 program with static binding PG:60
 parameter description PG:408
 parameters grouped by function PG:58
 program creation strategy PG:23, PG:25
 RETURNCODE data area PG:70
 syntax diagram PG:406
 using PG:57

Create Heap (CEECRHP) bindable API PG:20, PG:120

Create Program (CRTPGM) command PG:27

and ILE PG:18
 creating a program PG:73
 examples PG:98
 binding multiple modules PG:84
 parameters PG:83
 system actions PG:83
 using PG:82

Create RPG Module (CRTRPGMOD) command

and ILE PG:18
 default values of parameters PG:74
 defaults PG:75
 examples PG:97, PG:98
 parameter description PG:421
 parameter grouping by function PG:74
 program creation strategy PG:27
 syntax diagram PG:419
 using PG:74

Create Service Program (CRTSRVPGM) command

and ILE PG:18
 example PG:97
 parameters PG:92

creating a binder listing PG:85

creating a debug view
 COPY PG:168
 listing PG:168
 root source PG:167
 statement PG:169

creating a library PG:51

creating a module
 general discussion PG:73
 using CRTRPGMOD PG:74
 using CRTRPGMOD defaults PG:75

creating a program with the CRTBNDRPG command PG:57

creating a source physical file PG:51

creating programs
 coding considerations PG:45, PG:46
 examples of PG:59, PG:60, PG:61, PG:84
 OPM-compatible
 creating PG:23
 strategy to avoid PG:31
 strategies for PG:23
 CRTPGM command PG:82
 ILE application using CRTRPGMOD PG:27
 OPM-compatible PG:23
 strategy to avoid PG:31
 using CRTBNDRPG PG:25
 using CRTRPGMOD and CRTPGM PG:73
 using the one-step process PG:57

creating service programs
 about PG:91
 strategies PG:92

cross-reference listing PG:433

CRTBNDRPG command
 See Create Bound RPG Program (CRTBNDRPG) command

CRTPGM command
 See Create Program (CRTPGM) command

CRTRPGMOD command
 See Create RPG Module (CRTRPGMOD) command

CRTRPTPGM (create auto report program) command
 converting auto report members PG:389

CRTSRVPGM command
 See Create Service Program (CRTSRVPGM) command

CTDATA keyword
 **CTDATA R:144, R:164
 description R:277

currency symbol
 specifying R:232

CURSYM keyword R:232

CVTOPT keyword R:232

CVTOPT parameter
 CRTBNDRPG command PG:58, PG:413
 CRTRPGMOD command PG:74
 specifying on control specifications R:232

CVTRPGSRC (Convert RPG Source) command
 default parameter values PG:383
 example PG:388
 parameter description PG:384
 syntax diagram PG:383
 using the command defaults PG:387

CVTRPT parameter PG:386, PG:389, PG:394

cycle-free module PG:75, R:95

cycle, program
 commitment control PG:312
 detailed description R:23
 fetch overflow logic PG:320, R:28
 general description PG:4, R:19
 last cycle PG:5
 with initialization subroutine (*INZSR) R:26
 with lookahead R:29
 with match fields R:27
 with RPG IV exception/error handling R:29

D

data area data structure
 general information R:125
 statement
 externally described R:122
 program described R:122

data areas
 defining R:278, R:492, R:494
 DFTLEHSPEC data area R:227
 local data area (LDA) R:494
 PIP data area (PDA) R:492
 restrictions R:494
 retrieval
 explicit R:529
 implicit R:20, R:125
 RETURNCODE PG:70
 RPGLEHSPEC data area R:227
 unlocking
 explicit R:582
 implicit R:22, R:125
 UNLOCK operation code R:656
 writing
 explicit R:586
 implicit R:22, R:125

data attributes
 input specification R:306
 output specification R:337

data conversion built-in functions
 %CHAR (Convert to Character Data) R:355
 %DEC (Convert to Packed Decimal Format) R:356
 %DECH (Convert to Packed Decimal Format with Half Adjust) R:356
 %EDITC (Edit Value Using an Editcode) R:358
 %EDITFLT (Convert to Float External Representation) R:361
 %EDITW (Edit Value Using an Editword) R:362

data conversion built-in functions *(continued)*

- %FLOAT (Convert to Floating Format) R:369
- %INT (Convert to Integer Format) R:372
- %INTH (Convert to Integer Format with Half Adjust) R:372
- %UNS (Convert to Unsigned Format) R:396
- %UNSH (Convert to Unsigned Format with Half Adjust) R:396

data file, converting source from PG:390

data format

- binary R:165
- definition specification R:272
- external R:279, R:335
- float R:166
- integer R:168
- internal R:157
- packed-decimal R:169
- specifying external date or time format R:159
- specifying external numeric format R:158
- unsigned R:170
- zoned-decimal R:170

data information built-in functions

- %DECPOS (Get Number of Decimal Positions) R:357
- %ELEM (Get Number of Elements) R:363
- %LEN (Get Length) R:373
- %SIZE (Get Size in Bytes) R:384

data management feedback area

See file information data structure

data management operations PG:271

data structures

- alignment of R:124
- data area R:125
- defining R:123
- definition keyword summary R:296
- definition type entry R:270
- examples R:126
- externally described R:122
- file information R:125
- file information data structure R:65
- general information R:122
- indicator R:126
- multiple-occurrence
 - displaying while debugging PG:203
 - number of occurrences R:285, R:363
 - size of R:384
- overlying storage R:123
- printer control R:261
- program described R:122
- program-status R:125
- rules R:125
- rules for R:4
- saving for attached device R:263
- special R:125
- subfields
 - alignment of R:124
 - conversion problems PG:402

data structures *(continued)*

- subfields *(continued)*
 - defining R:123, R:270
 - displaying while debugging PG:203
 - external definition R:280
 - name prefixing R:123, R:261, R:293
 - overlying storage R:123, R:291
 - renaming R:123, R:279
- type of R:270
- using EVAL debug command PG:203
- with OCCUR operation code R:578

data type

- allowed for built-in functions R:402
- basing pointer R:182
- character R:159
- data mapping errors R:199
- date R:174, R:233, R:256, R:264, R:277
- definition specification R:272
- graphic R:179
- numeric R:165
- of return value R:616
- procedure pointer R:188
- supported by binary operations R:402
- supported by unary operations R:402
- supported in expressions R:402
- time R:177, R:241, R:294
- timestamp R:179

data-area operations

- DEFINE (field definition) R:492
- general information R:429
- IN (retrieve a data area) R:429, R:529
- OUT (write a data area) R:429, R:586
- UNLOCK (unlock a data area) R:429, R:656

database data

- null values R:190
- updating while debugging PG:171
- variable-length fields R:195

database file

- See also DISK file
- data file PG:281
- field level description PG:281
- general discussion PG:281
- physical and logical files PG:281
- record level description PG:281
- source member PG:281

date data field

- DATFMT R:256
- DATFMT on control specification R:233
- DATFMT on definition specification R:277
- effect of end position R:204
- general discussion R:174
- moving R:439
- unexpected results R:432
- zero suppression R:202

date data format

- *JOB RUN date separator and format R:176

date data format *(continued)*

- *LONGJUL format R:176
- 3-digit year century formats R:176
- control specification R:233
- definition specification R:277
- description R:174
- file description specification R:256
- initialization R:177
- input specification R:306
- internal format on definition specification R:272
- output specification R:335
- separators R:177
- table of external formats R:176
- table of RPG-defined formats R:175

date-time operations

- ADDUR (add duration) R:430, R:454
- adding or subtracting dates R:431
- calculating date durations R:431
- EXTRCT (extract date/time) R:430, R:520
- general information R:430
- SUBDUR (subtract duration) R:430, R:640
- TEST (test date/time/timestamp) R:430
- unexpected results R:432

date, user R:8

- *DATE, *DAY, *MONTH, *YEAR R:8
- UPDATE, UDAY, UMONTH, UYEAR R:8

DATEDIT keyword R:233

DATFMT keyword

- control specification R:233
- definition specification R:277
- file description specification R:256

DB2 for AS/400 SQL

- entering SQL statements PG:55

DBCS

- See also* graphic data type
- in RPG IV character fields PG:377
- NLSS debug considerations PG:184

DBGVIEW parameter

- coordinating with listing options PG:69
- CRTBNDRPG command PG:58, PG:411
- CRTRPGMOD command PG:74
- preparing a program for debugging PG:166
- using PG:59
- values for viewing source PG:174

DDM

- See* distributed data management (DDM)

DEALLOC (free storage) operation code PG:113, R:436, R:490

deallocate storage (DEALLOC) operation code R:490

debug commands

- ATTR PG:210
- CLEAR PG:179
- DISPLAY PG:175
- EQUATE PG:210
- equating with a name while debugging PG:210

debug commands *(continued)*

- EVAL PG:199, PG:208
- general discussion PG:164
- STEP PG:194, PG:195
- STEP INTO PG:196
- STEP OVER PG:195
- WATCH PG:188

debug data

- creating PG:166
- effect on object size PG:166
- none PG:166
- removing from a module PG:88

DEBUG keyword R:233

debug view

- changing while debugging PG:176
- COPY source PG:168
- default PG:169
- definition PG:166
- listing PG:168
- root source PG:167
- statement PG:169

debugging

- adding an object to a session PG:172
- built-in functions
 - %ADDR PG:206
 - %INDEX PG:206
 - %SUBSTR PG:206
 - %VARS PG:206
 - changing values using %SUBSTR PG:208
 - examples PG:207
 - general discussion PG:206
- changing field values PG:208
- changing modules while debugging PG:175
- coordinating with listing options PG:69
- creating a program for debugging PG:59
- differences between ILE RPG and OPM RPG/400 PG:374
- displaying attributes of PG:210
- displaying data addressed by pointers PG:206
- displaying data and expressions PG:199
- displaying fields as hexadecimal values PG:205
- displaying fields in character format PG:205
- displaying indicators PG:204
- displaying multiple-occurrence data structures PG:203
- displaying the contents of a table PG:202
- displaying the contents of an array PG:202
- general discussion PG:163
- National Language Support PG:211
- NLSS considerations PG:184
- obtaining a formatted dump PG:251
- OPM program limit in debug session PG:172
- optimization effects PG:87, PG:164
- overview PG:20
- preparing a program PG:166
- removing an object from a session PG:172, PG:173

debugging *(continued)*

- rules for assigning values using EVAL PG:208
- setting and removing breakpoints PG:178
- setting debug options PG:171
- setting watch conditions PG:188
- starting the source debugger PG:169
- stepping through PG:194
- unexpected results PG:201
- updating production files PG:171
- viewing shorthand names PG:211
- viewing source PG:174

DECEDIT keyword R:234

decimal point character R:234

decimal positions

- calculation specifications R:320
- get with %DECPOS R:357
- input specifications
 - field description entry for program described file R:308
 - program-described WORKSTN file PG:340
 - with external descriptions PG:267
 - with arithmetic operation codes R:418

declarative operations

- DEFINE (field definition) R:432, R:492
- general information R:432
- KFLD (define parts of a key) R:432, R:533
- KLIST (define a composite key) R:432
- PARM (identify parameters) R:432, R:587
- PLIST (identify a parameter list) R:432, R:590
- TAG (tag) R:432, R:646

decompressing an object PG:88

default activation group PG:23, PG:31, PG:111

- running in PG:111

default data formats

- date R:175, R:233, R:277
- time R:177, R:241, R:294
- timestamp R:179

default exception handler, RPG PG:220

default heap PG:113

DEFINE (field definition) operation code R:432, R:492

define a composite key (KLIST) operation code R:534

DEFINE parameter

- CRTBNDRPG command PG:58, PG:419
- CRTRPGMOD command PG:74

define parts of a key (KFLD) operation code R:533

defining a field as a data area R:492

defining a field based on attributes R:492

defining a file R:218

defining a symbolic name for the parameter list R:590

defining an alternate collating sequence R:163

defining indicators R:33

defining like

- DEFINE operation R:492

defining like *(continued)*

- LIKE keyword R:283
- subfields R:123

defining parameters R:587

definition specification keywords

- ALIGN R:274
- ALT R:274
- ALTSEQ R:275
- ASCEND R:275
- BASED R:276
- CONST R:276
- continuation line R:224
- CTDATA R:277
- DATFMT R:277
- DESCEND R:277
- DIM R:278
- DTAARA R:278
- EXPORT R:278
- EXTFLD R:279
- EXTFMT R:279
- EXTNAME R:280
- EXTPGM R:280
- EXTPROC R:281
- FROMFILE R:282
- IMPORT R:282
- INZ R:282
- LIKE R:283
- NOOPT R:284
- OCCURS R:285
- OPDESC R:286
- OPTIONS R:286
- OVERLAY R:291
- PACKEVEN R:293
- PERRCD R:293
- PREFIX R:293
- PROCPTR R:294
- specifying R:273
- STATIC R:294
- TIMFMT R:294
- TOFILE R:294
- VALUE R:295
- VARYING R:295

definition specifications

- decimal positions R:273
- entry summary by type R:295
- external description R:269
- form type R:268
- from position R:271
- general R:267
- general description PG:3
- internal format R:272
- keyword summary by type R:296
- keywords R:273
- name R:269
- to position / length R:271
- type of data structure R:270

definition specifications *(continued)*

type of definition R:270

DELETE (delete record) operation code R:433, R:496

delete a record

DELETE (delete record) operation code R:496

output specifications entry (DEL) R:327

deleting an activation group PG:112

DESCEND keyword R:277

descending sequence

definition specification keyword ASCEND R:277

file description specifications entry R:248

describe data structures R:299

describing arrays R:218

See also definition specifications

describing data structures

See definition specifications

describing record address files

See definition specifications

describing tables R:218

See also definition specifications

describing the format of fields R:325

See also output, specifications

describing the record R:325

See also output, specifications

describing when the record is written R:325

See also output, specifications

description of parameters

CRTBNDRPG command PG:408

CRTRPGMOD command PG:421

CVTRPGSRC command PG:384

descriptors, operational

definition PG:140

example PG:94

minimal R:379

OPDESC keyword R:286

detail (D) output record R:327

DETAIL parameter

creating a binder listing PG:85

detailed program logic R:23

DETC PG:238

file exception/error subroutine (INFSR) R:80

flowchart R:22

program exception/errors R:83

detecting errors in a program PG:163

DETL PG:238

file exception/error subroutine (INFSR) R:80

flowchart R:20

program exception/errors R:83

device files

device dependence PG:261

device independence PG:261

DISK files PG:281

general discussion PG:315

multiple-device PG:342

PRINTER files PG:316

device files *(continued)*

SEQ files PG:326

workstation files PG:331

device name

specifying R:256

device name, function of PG:262

devices

maximum number of R:259

on file description specification R:254

saving data structure R:263

saving indicators R:263

WORKSTN PG:331

DEVID keyword R:256

DFTACTGRP keyword R:234

DFTACTGRP parameter on CRTBNDRPG

CRTBNDRPG command PG:58

description PG:409

running in OPM default PG:111

specifying on control specifications R:234

using PG:57, PG:60, PG:61

DFTLEHSPEC data area R:227

DFTNAM keyword R:235

diagnosing errors in a program PG:163

differences between OPM and ILE RPG

behavioral differences PG:373

exception handling PG:222

different views of a module PG:176

DIM keyword R:278

Discard Heap (CEEDSHP) bindable API PG:20, PG:120

disconnecting a file from the program R:487

DISK file

externally described

access path PG:282

as program-described PG:264

examples PG:283

general description PG:282

record format specifications PG:282

file operation codes allowed

for keyed processing methods PG:306

for non-keyed processing methods PG:306

general description PG:281

processing methods R:265

consecutive processing PG:292

overview PG:291

random-by-key processing PG:299

relative-record-number processing PG:303

sequential-by-key processing PG:293

sequential-within-limits processing PG:300

program-described

indexed file PG:288

processing PG:291, R:265

record-address file PG:290

sequential file PG:290

record-format specifications PG:282

summary of processing methods R:265

DISPLAY debug command
 definition PG:164
 using PG:175
 viewing shorthand names PG:211

display function (DSPLY) operation code R:506

Display Module (DSPMOD) command PG:150
 copyright information R:231

Display Module Source (DSPMODSRC)
 command PG:171, PG:172, PG:173, PG:175

Display Program (DSPPGM) command
 copyright information R:231
 determining optimization level PG:88

Display Program References (DSPPGMREF)
 command PG:150

Display Service Program (DSPSRVPGM)
 command PG:91
 copyright information R:231

displaying attributes of a field PG:210

displaying data and expressions while debugging PG:199

distributed data management (DDM)
 files PG:313

DIV (divide) operation code R:418, R:497

dividing factors R:497

division operator (/) R:402

DO operation code R:445, R:498

DO-group
 general description R:445

documentation of programs PG:69

DOU (do until) operation code R:397, R:427, R:445, R:500

double asterisk ()**
 alternate collating sequence table R:164
 arrays and tables R:145
 file translation table R:105
 for program described files R:302
 lookahead fields R:302, R:303

double byte character set
 See also graphic data type
 in RPG IV character fields PG:377
 NLSS debug considerations PG:184

DOUxx (do until) operation code R:427, R:445, R:501

DOW (do while) operation code R:397, R:427, R:445, R:503

DOWxx (do while) operation code R:427, R:445, R:504

DSPLY (display function) operation code R:437, R:506

DSPMOD command
 See Display Module (DSPMOD) command

DSPMODSRC command
 See Display Module Source (DSPMODSRC) command

DSPPGM command
 See Display Program (DSPPGM) command

DSPPGMREF command
 See Display Program References (DSPPGMREF) command

DSPSRVPGM command
 See Display Service Program (DSPSRVPGM) command

DTAARA keyword R:278

DUMP (program dump) operation code R:435, R:509
 obtaining a formatted dump PG:251
 using PG:251

dump, formatted PG:251

dynamic array
 allocating storage during run-time PG:120
 definition of R:142
 rules for loading R:143
 with consecutive elements R:144
 with scattered elements R:143

dynamic calls PG:19, PG:130
 See also program/procedure call
 using CALLP R:466

dynamic storage PG:113

E

EBCDIC
 collating sequence R:675

edit codes
 combination (1-4, A-D, J-Q) R:202
 description R:202
 effect on end position R:204
 simple (X, Y, Z) R:202
 summary tables R:202, R:206
 unsigned integer field R:204
 user-defined (5-9) R:204
 using %EDITC R:358
 zero suppression R:202

edit source (STRSEU) command PG:52

edit word
 constant/editword field continuation R:225
 formatting R:209, R:213
 on output specifications R:337
 parts of R:209
 body R:209
 expansion R:210
 status R:210
 rules for R:214
 using %EDITW R:362

edit, date R:202

editing
 built-in functions
 %EDITC (Edit Value Using an Editcode) R:358
 %EDITFLT (Convert to Float External Representation) R:361
 %EDITW (Edit Value Using an Editword) R:362
 date fields R:202

editing *(continued)*

- decimal point character R:234
- externally described files R:214
- non-printer files R:204

elements

- number of in array or table R:278, R:363
- number per record R:293
- size of field or constant R:384

eliminating errors in a program PG:163

ELSE (else do) operation code R:445, R:510

else do (ELSE) operation code R:510

ENBPFCOL keyword R:235

ENBPFCOL parameter

- CRTBNDRPG command PG:58, PG:418
- CRTRPGMOD command PG:74
- specifying on control specifications R:235

end a group (ENDyy) operation code R:511

End Debug (ENDDBG) command PG:169

End Job (ENDJOB) R:595

end of file

- built-in function R:364
- file description specifications entry R:247
- with primary file R:47

end position

- effect of edit codes on R:208
- in output record
- for RPG IV output specifications R:334

End Subsystem (ENDSBS) R:595

End System (ENDSYS) R:595

ending a group of operations (CASxx, DO, DOUxx, DOWxx, IFxx, SELECT) R:511

ending a program or procedure

- abnormal end PG:155
- after system call PG:109
- normal end PG:154
- return overview PG:153
- returning without ending PG:155
- using bindable APIs PG:156

ending a program, without a primary file R:29

ending a subroutine R:512

ending commitment control PG:307

ENDSR (end of subroutine) operation code R:448, R:512

- return points R:80
- specifying a return point PG:238

ENDyy (end a group) operation code R:511

ENDyy (end a group) operation code R:445

ENTMOD parameter PG:82

entry module PG:28, PG:82

See also program entry procedure (PEP)

***ENTRY PLIST** PG:153

environment

See Integrated Language Environment (ILE)

equal operator (=) R:402

EQUATE debug command

- definition PG:164

EQUATE debug command *(continued)*

- example PG:210
- using PG:210

equating a name with a field, expression, or command PG:210

error handling

- See also* exception/error handling
- major/minor error return codes R:79
- steps R:32

error indicators

- specifying PG:227

error logic

- error handling routine R:32

error subroutines

- avoiding a loop PG:236
- for file errors PG:229
- program PG:233
- using PG:228

errors

- See also* exception
- correcting compilation PG:66
- correcting run-time PG:68
- file PG:220
- program PG:220

escape messages

- definition PG:218
- unhandled PG:224

EVAL (evaluate expression) operation code

- description R:513
- structured programming R:445
- use with %SUBST R:391
- with expressions R:397

EVAL debug command

- changing values PG:208
- contents of a table PG:202
- contents of an array PG:202
- definition PG:164
- displaying data structures PG:203
- example PG:200, PG:208
- in character format PG:205
- indicators PG:204
- rules for assigning values PG:208
- using PG:199

event file for CODE/400 PG:411

examples

- compiling
 - binding multiple modules PG:84
 - OPM-compatible program PG:61
 - program for source debugging PG:59
 - program with static binding PG:60
 - sample binder listing PG:100
 - service program PG:94
- converting to RPG IV
 - all members in a file PG:388
 - performing a trial conversion PG:388
 - sample conversion PG:390
 - some members in a file PG:388

examples *(continued)*

debugging

- adding a service program to a session PG:172
- changing field values PG:208
- changing the debug view of a module PG:176
- displaying attributes of a field PG:210
- displaying data addressed by pointers PG:206
- displaying fields as hexadecimal values PG:205
- displaying fields in character format PG:205
- displaying indicators PG:204
- displaying multiple-occurrence data structures PG:203
- displaying the contents of a table PG:202
- displaying the contents of an array PG:202
- removing programs from a session PG:173
- setting a conditional breakpoint PG:182
- setting an unconditional breakpoint PG:179
- setting debug options PG:171
- source for debug examples PG:211
- using %SUBSTR to display field values PG:207
- viewing a different module in a debug session PG:175

handling exceptions

- *PSSR error subroutine PG:233
- avoiding a loop in an error subroutine PG:236
- cancel handler PG:244
- file error subroutine PG:230
- unhandled escape message PG:224
- unhandled function check PG:225
- using a cancel handler PG:245
- using a condition handler PG:239

I/O

- data maintenance PG:349
- inquiry by zip code and search on name PG:364
- inquiry program PG:346
- subfile processing PG:358
- interactive application PG:345
- managing your own heap PG:120
- module with multiple procedures PG:41
- passing parameters using the CL CALL command PG:104
- program/procedure call
 - checking number of passed parameters PG:143
 - using omitted parameters PG:94
- sample ILE RPG program PG:6
- subprocedures PG:37, PG:39
 - creating a NOMAIN module PG:75

examples of program exception/errors R:82

EXCEPT (calculation time output) operation

code R:433, R:515

EXCEPT name

- on output specifications R:330
- rules for R:4

exception

- monitoring during run time PG:109
- nested, PG:223

exception (E) output records R:327

exception handler

- priority of PG:223
- RPG-specific PG:220, PG:227

exception messages

- percolation PG:218
- types of PG:218
- unexpectedly handled by CL MONMSG PG:247
- unhandled PG:223

exception/error codes

- file status codes R:77
- program status codes R:85

exception/error handling

- *PSSR error subroutine PG:233
- avoiding a loop PG:236
- built-in functions
 - %ERROR (Return Error Condition) R:368
 - %STATUS (Return File or Program Status) R:386
- cancel handler PG:244
- condition handler PG:239
- data mapping errors R:199
- differences between ILE RPG and OPM RPG/400 PG:222, PG:374
- error indicators PG:227
- error/exception subroutine overview PG:228
- file error/exception (INFSR) subroutine PG:229
- file exception/error subroutine R:80
- file information data structure R:65
- flowchart R:31
- general considerations PG:223
- INFSR R:80
- NOOPT keyword PG:226
- optimization considerations PG:226
- overview PG:217
- percolation PG:218
- program exception/error subroutine (*PSSR) R:87
- program status data structure R:82
- RPG-specific PG:220
- specifying a return point PG:238
- status codes R:77, R:85
 - file R:77
 - program R:82, R:85
- types of PG:217
- unhandled PG:223
- using 'E' extender PG:227
- EXFMT (write/then read format) operation**
 - code PG:342, R:433, R:517
- expansion (of an edit word)** R:210, R:213
- EXPCPY parameter** PG:386
- exponent operator (**)** R:402
- EXPORT keyword**
 - definition specification R:278
 - duplicate names PG:84
 - procedure specification R:343

exported data, defining R:278
exporting a procedure R:343
exporting a program R:343
expression-using operation codes
 CALLP (call prototyped procedure) R:397
 DOU (do until) R:397
 DOW (do while) R:397
 EVAL (evaluate) R:397
 general information R:397
 IF (if/then) R:397
 RETURN (return) R:397
 WHEN (when true then select) R:397
expressions
 data type of operands R:402
 general rules R:398
 intermediate results R:404
 operators R:399
 order of evaluation of operands R:410, R:411
 precedence rules R:399
 precision rules R:404
 returning values PG:136
EXPROPTS keyword R:235
EXSR (invoke subroutine) operation code R:448, R:519
EXTBININT keyword
 and binary fields R:166
 description R:235
extended factor 2 field, continuation R:224
extension specifications
 conversion problems PG:393, PG:400
external (U1-U8) indicators
 as field indicator R:310, R:313
 as field record relation indicator R:49, R:310
 as record identifying indicator R:302, R:312
 conditioning calculations R:318
 conditioning output R:329
 general description R:46
 resetting R:46, R:311
 setting R:62
external data area
 defining R:278, R:492
external data format
 date R:256
 definition R:158
 in input specification R:307
 specifying using EXTFMT R:279
 specifying using TIMFMT R:294
 time R:264
external field name
 renaming R:312
external message queue (*EXT) R:506
external procedure name R:281
external program name R:280
external-references list in compiler listing PG:434
externally described file
 access path PG:282

externally described file (continued)
 adding to external description PG:265
 advantages PG:261
 as program-described PG:264
 as WORKSTN file PG:331, PG:335
 definition PG:263
 editing R:214
 file description specifications for PG:265
 input specifications for R:311
 output specifications for PG:269, R:338
 overriding PG:267
 physical and logical files PG:281
 record format
 for a subfile R:264
 ignoring R:258
 including R:258
 renaming R:263
 writing to a display R:264
 record format specifications PG:282
 renaming field names PG:266
 renaming fields R:261
 renaming record format PG:266
 specifications PG:265
externally described files, field description and control entries, output specifications
 field name R:339
 output indicators R:339
externally described files, field description entries, input specifications
 control level R:313
 external field name R:312
 field indicators R:313
 field name R:312
 general description R:312
 matching fields R:313
externally described files, record identification and control entries, output specifications
 EXCEPT name R:339
 logical relationship R:338
 output indicators R:339
 record addition R:339
 record name R:338
 release R:339
 type R:338
externally described files, record identification entries, input specifications
 form type R:311
 general description R:311
 record identifying indicator R:312
 record name R:311
EXTFLD keyword R:123, R:279
EXTFMT keyword R:279
EXTIND keyword R:257
EXTNAME keyword R:280
EXTPGM keyword R:269, R:280, R:466

EXTPROC keyword R:269, R:281
EXTRCT (extract date/time) operation code R:430, R:520

F

factor 1

as search argument R:538
 entries for, in calculation specification R:318
 in arithmetic operation codes R:418

factor 2

entries for, in calculation specification R:320
 in arithmetic operation codes R:418

feedback built-in functions

%EOF (Return End or Beginning of File Condition) R:364
 %EQUAL (Return Exact Match Condition) R:366
 %ERROR (Return Error Condition) R:368
 %FOUND (Return Found Condition) R:370
 %NULLIND (Query or Set Null Indicator) R:376
 %OPEN (Return File Open Condition) R:377
 %PARMS (Return Number of Parameters) R:379
 %STATUS (Return File or Program Status) R:386

FEOD (force end of data) operation code R:433, R:522

fetch overflow

See also overflow (OA-OG, OV) indicators
 entry on output specifications R:328
 general description PG:320, R:28, R:328
 logic PG:320, R:28
 relationship with AND line R:329
 relationship with OR line R:329

field

binary R:165
 on output specifications R:335
 changing the value while debugging PG:208
 control R:37
 defining as data area R:494
 defining like R:283
 defining new R:320
 description entries in input specification R:306, R:312
 displaying attributes of while debugging PG:210
 displaying while debugging
 as hexadecimal values PG:205
 in character format PG:205
 using EVAL PG:199
 equating with a name while debugging PG:210
 key R:250
 key, starting location of R:259
 location and size in record R:307
 location in input specification R:307
 lookahead
 with program described file R:302, R:303
 maintaining current values while debugging PG:164
 match R:97

field (continued)

name in input specification R:308
 null-capable R:190
 numeric
 on output specifications R:332
 packed R:169
 record address R:250
 renaming R:261, R:263
 result R:320
 size of R:384
 standalone R:114
 zeroing R:334, R:340

field definition (DEFINE) operation code R:492

field indicators (01-99, H1-H9, U1-U8, RT)

as halt indicators R:43
 assigning on input specifications
 for externally described files R:313
 for program described files R:310
 conditioning calculations R:318
 conditioning output R:329
 general description R:43
 numeric R:44
 rules for assigning R:43
 setting of R:62

field length

absolute (positional) notation R:123, R:271
 arithmetic operation codes R:418
 calculation operations R:320
 calculation specifications R:320
 compare operation codes R:427
 input specifications R:307
 key R:250
 length notation R:123, R:271
 numeric or alphanumeric R:307
 record address R:250

field location entry (input specifications)

for program described file R:307

field name

as result field R:320
 external R:312
 in an OR relationship R:306
 in input specification R:312
 on output specifications R:332
 rules for R:4
 special words as R:332
 special words as field name R:7

field record relation indicators (01-99, H1-H9, L1-L9, U1-U8)

assigning on input specifications R:310
 example R:51
 general description R:49
 rules for R:49

field-reference file, example of PG:283

figurative constants

*ALL'x..', *ALL'x1..', *BLANK/*BLANKS,
 *HIVAL/*LOVAL, *ZERO/*ZEROS,
 *ON/*OFF R:120

figurative constants *(continued)*

rules for R:121

file

adding records to R:247, R:327
 array R:246
 combined R:245
 conditioning indicators R:48
 deleting existing records from R:327
 deleting records from
 DEL R:327
 DELETE R:496
 description specifications R:243
 designation R:245
 device dependence PG:261
 device independence PG:261
 differences between ILE RPG and OPM
 RPG/400 PG:375
 DISK PG:281
 end of R:247
 exception/error codes R:77
 externally described PG:261
 externally described disk PG:282
 externally described, input specification for R:311
 feedback information in INFDS R:66
 feedback information in INFDS after POST R:67
 file organization R:253
 format R:249
 full procedural R:29, R:246
 general considerations PG:261
 indexed PG:288, R:253
 input R:245
 locking PG:275
 maximum number allowed R:243
 name
 entry on file description specifications R:244
 entry on input specifications R:300
 entry on output specifications R:326
 externally described PG:261, R:244
 override PG:267
 program described R:244
 program-described PG:271
 rules for R:4
 naming conventions PG:263
 nonkeyed program described R:253
 normal codes for file status R:77
 number allowed on file description
 specifications R:243
 open options PG:277
 output R:245
 override PG:267
 primary R:246
 PRINTER PG:316
 See also PRINTER file
 processing R:29
 processing charts
 sequential file PG:326
 SPECIAL file PG:328

file *(continued)*

processing charts *(continued)*

 WORKSTN file PG:342

program described PG:261, PG:271

record address R:246

redirection PG:262

rules for conditioning R:48

secondary R:246

SEQ PG:290, PG:326

See also SEQ file

sharing PG:277

status codes R:77

table R:246

types R:245

update R:245

valid keys PG:285

WORKSTN PG:331

See also WORKSTN file

file conditioning indicators R:45

 general description R:48

 specifying with EXTIND R:257

file description specification keywords

BLOCK R:255

COMMIT R:256

continuation line R:224

DATFMT R:256

DEVID R:256

EXTIND R:257

FORMLEN R:257

FORMOFL R:257

IGNORE R:258

INCLUDE R:258

INDDS R:258

INFDS R:259

INFSR (file exception/error subroutine) R:259

KEYLOC R:259

MAXDEV R:259

OFLIND R:259

PASS R:260

PGMNAME R:260

PLIST R:260

PREFIX R:261

PRTCTL R:261

RAFDATA R:262

RECNO R:262

RENAME R:263

SAVEDS R:263

SAVEIND R:263

SFILE R:264

SLN R:264

TIMFMT R:264

USROPN R:265

file description specifications

 commitment control PG:309

 device R:254

 end of file R:247

file description specifications *(continued)*

file addition R:247
 file designation R:245
 file format R:249
 file name R:244
 file organization R:253
 file type R:245
 for externally described files PG:265
 form type R:244
 general description PG:3, R:243
 key field starting location R:259
 length of key or record address R:250
 limits processing R:249
 maximum number of files allowed R:243
 overflow indicator R:259
 record address type R:250
 record length R:249
 sequence R:248

file exception/error subroutine (INFSR)

description PG:229, R:80
 example PG:230
 INFSR keyword R:259
 return points R:80
 specifications for PG:229, R:80

file exception/errors

definition PG:220
 example PG:230
 file information data structure (INFDS) R:65
 general information R:65
 how to handle subroutine (INFSR) R:80
 statement specifications R:303
 using an INFSR subroutine PG:229

file information data structure R:65, R:66

contents of file feedback information R:66
 contents of file feedback information after POST R:67
 continuation line option R:254
 entry on file description specifications R:254
 example PG:230
 general information R:125
 INFDS keyword R:259
 predefined subfields R:68
 status codes R:77
 subfields
 specifications R:125
 using in an error subroutine PG:229

file locking PG:275

file operations

ACQ (acquire) operation code R:433, R:452
 allowed with DISK file PG:306
 allowed with PRINTER file PG:316
 allowed with sequential file PG:326
 allowed with SPECIAL file PG:328
 allowed with WORKSTN file PG:341
 CHAIN (random retrieval from a file based on record number) R:433, R:474

file operations *(continued)*

CLOSE (close files) operation code R:433, R:487
 COMMIT (commit) operation code R:433, R:488
 DELETE (delete record) operation code R:433, R:496
 EXCEPT (calculation time output) operation code R:433, R:515
 EXFMT (write/then read format) operation code R:433, R:517
 FEOD (force end of data) operation code R:433, R:522
 FORCE (force a file to be read) operation code R:433, R:523
 general description R:433
 NEXT (next) operation code R:433, R:577
 OPEN (open file for processing) operation code R:433, R:582
 POST (post) operation code R:433, R:592
 READ (read a record) operation code R:433, R:594
 READC (read next modified record) operation code R:433, R:597
 READE (read equal key) operation code R:433, R:599
 READP (read prior record) operation code R:433, R:602
 READPE (read prior equal) operation code R:433, R:604
 REL (release) operation code R:433, R:608
 ROLBK (roll back) operation code R:433, R:619
 SETGT (set greater than) operation code R:433, R:625
 SETLL (set lower limits) operation code R:433, R:629
 UNLOCK (unlock a data area) operation code R:433, R:656
 UPDATE (modify existing record) operation code R:433
 WRITE (create new records) operation code R:433, R:664

file overrides PG:267

example PG:274
 general discussion PG:273, PG:304
 indicated in compiler listing PG:425

file record length, conversion

considerations PG:380

file sharing PG:277

file translation R:105

FTRANS keyword R:237
 table records R:108

final summary in compiler listing PG:435

FIND debug command PG:165

first page (1P) indicator

conditioning output R:329, R:333
 general description R:46
 restrictions R:46
 setting R:62

first program cycle R:19
FIXNBR keyword R:236
FIXNBR parameter
 CRTBNDRPG command PG:58, PG:416
 CRTRPGMOD command PG:74
 specifying on control specifications R:236
float format
 alignment of fields R:168
 considerations for using R:171
 converting to R:369
 definition R:166
 displaying as R:361
 external display representation R:167
 float keys R:253
 FLTDIV keyword R:237
 input field specification R:166
 output field specification R:166
float literals R:116
floating point representation R:166, R:404
flowchart
 detailed program logic R:23
 fetch-overflow logic PG:320, R:27
 general program logic R:19
 lookahead logic R:27
 match fields logic R:27
 RPG IV exception/error handling R:31
FLTDIV keyword R:237
FORCE (force a file to be read) operation code R:433, R:523
force a certain file to be read on the next cycle (FORCE) operation code R:523
force end of data (FEOD) operation code R:522
form type
 externally described files R:311
 in calculation specification R:316
 on control specification R:228
 on description specifications R:243
 program described file R:300
format
 of file R:249
format name PG:339
format of compiler listing, specifying PG:64
format, data
 binary R:165
 definition specification R:272
 external R:279, R:335
 float R:166
 integer R:168
 internal R:157
 packed-decimal R:169
 specifying external date or time format R:159
 specifying external numeric format R:158
 unsigned R:170
 zoned-decimal R:170
formatted dump PG:251

formatting edit words R:213
FORMLEN keyword R:257
FORMOFL keyword R:257
FORMSALIGN keyword R:237
FREE (deactivate a program) operation code PG:397
Free Storage (CEEFRST) bindable API PG:20
free-form syntax
 CALLP (call a prototyped procedure) R:466
freeing resources of ILE programs PG:112
freeing storage R:490
FROMFILE keyword R:282
FROMFILE parameter PG:384
FROMMBR parameter PG:384, PG:388
FTRANS keyword R:237
 **FTRANS R:144, R:164
 description R:106
full procedural file
 description of R:246
 file description specifications entry R:245
 file operation codes R:433
function check
 definition PG:218
 unhandled PG:225
function key
 corresponding indicators R:51
function key indicators (KA-KN, KP-KY)
See also WORKSTN file
 corresponding function keys R:52
 general description R:51
 setting R:62
function keys
 indicators PG:334
 with WORKSTN file PG:334

G
GDDM PG:158
general (01-99) indicators R:34
general program logic R:19
generating a program R:218
See also compiling
See also control specifications
GENLVL keyword R:237
GENLVL parameter
 CRTBNDRPG command PG:58, PG:409
 CRTRPGMOD command PG:74
 specifying on control specifications R:237
Get Descriptive Information About a String Argument (CEESGI) PG:141
Get Heap Storage (CEEGTST) bindable API PG:20, PG:120
get/set occurrence of data structure R:578
global variables R:91, R:112
GOTO (go to) operation code R:422, R:524

graphic data type

- as compile-time data R:146, R:154
- concatenating graphic strings R:473
- definition specification R:272
- description R:179
- displaying R:508
- fixed length R:180
- moving R:438, R:545
- NLSS debug considerations PG:184
- rules for assigning values using EVAL PG:208
- size of R:384
- substrings R:391
- variable length R:180
- verifying with CHECK R:477, R:479

graphic support PG:158

Graphical Data Display Manager(GDDM) PG:158

greater than operator (>) R:402

greater than or equal operator (>=) R:402

H

H1-H9

- See halt (H1-H9) indicators

half adjust

- on calculation specifications R:318, R:322
- operations allowed with R:318, R:322

halt (H1-H9) indicators

- as field indicators R:310, R:313
- as field record relation indicator R:311
- as record identifying indicator R:302, R:312
- as resulting indicator R:321
- conditioning calculations R:318
- conditioning output R:329, R:332
- general description R:52
- setting R:62
- used to end a program/procedure PG:154, PG:155

handling exceptions/errors

- See also exception/error handling
- *PSSR error subroutine PG:233
- avoiding a loop PG:236
- built-in functions
 - %ERROR (Return Error Condition) R:368
 - %STATUS (Return File or Program Status) R:386
- cancel handler PG:244
- condition handler PG:239
- data mapping errors R:199
- differences between ILE RPG and OPM RPG/400 PG:222, PG:374
- error indicators PG:227
- error/exception subroutine overview PG:228
- file error/exception (INFSR) subroutine PG:229
- file exception/error subroutine R:80
- file information data structure R:65
- flowchart R:31
- general considerations PG:223

handling exceptions/errors (continued)

- INFSR R:80
- NOOPT keyword PG:226
- optimization considerations PG:226
- overview PG:217
- percolation PG:218
- program exception/error subroutine (*PSSR) R:87
- program status data structure R:82
- RPG-specific PG:220
- specifying a return point PG:238
- status codes R:77, R:85
 - file R:77
 - program R:82, R:85
- types of PG:217
- unhandled PG:223
- using 'E' extender PG:227

header specifications

- See control specifications

heading (H) output records R:327

heading information for compiler listing R:11

heap

- default heap PG:113
- definition PG:113
- example PG:120

help command key PG:334

hexadecimal values, displaying while

- debugging PG:205

home command key PG:334

I

I/O differences between ILE RPG and OPM RPG/400 PG:375

ICF communications file PG:331

identifying a parameter list R:590

identifying an activation group PG:110

IF (if/then) operation code R:397, R:427, R:445, R:526

if/then (IF) operation code R:527

IFxx (if/then) operation code R:427, R:445, R:527

IGNORE keyword PG:266, R:258

ignoring record format PG:266

ILE

- See Integrated Language Environment (ILE)

ILE C

- as ILE language PG:17
- in advanced application PG:30
- in mixed-language application PG:29
- parameter passing method PG:149
- source for module in debug example PG:216
- specifying lowercase name R:269

ILE CL

- as ILE language PG:17
- as module in ILE program PG:27
- calling ILE RPG program PG:28
- calling RPG program PG:25

ILE CL *(continued)*

- in advanced application PG:30
- in mixed-language application PG:29
- parameter passing method PG:149
- unexpectedly handling status and notify exceptions PG:247

ILE COBOL

- as ILE language PG:17
- parameter passing method PG:149

ILE RPG

- behavior of bound modules PG:80
- behavioral differences between OPM RPG/400 PG:373
- converting to PG:379
- data management operations PG:271
- device types supported PG:315
- exception handling overview PG:220
- logic chart PG:4
- overview of RPG IV language PG:3
- sample program PG:6

ILE RPG restrictions, summary R:673

ILE source debugger

- debug commands PG:164
- description PG:163
- starting PG:169

IMPORT keyword R:282

imported data, defining R:282

IN (retrieve a data area) operation code R:429, R:529

INCLUDE keyword R:258

include source view, creating PG:168

INDDS keyword R:258

INDENT keyword R:238

INDENT parameter PG:168

- CRTBNDRPG command PG:58, PG:413
- CRTRPGMOD command PG:74
- specifying on control specifications R:238

indentation bars in source listing R:527

indenting structured operations in the compiler listing PG:65

indexed file

- access path PG:288
- format of keys R:253
- general description PG:288
- key field R:259
- processing R:253
- valid search arguments PG:288

indicating calculations R:315

See also calculation, specifications

indicating length of overflow line R:218

indicator data structure

- general information R:126
- INDDS keyword R:258

indicator-setting operations

- general information R:435
- SETOFF (set off) R:435, R:633

indicator-setting operations *(continued)*

- SETON (set on) R:435, R:634

indicators

- See also* individual operation codes as error indicators PG:227
- calculation specifications R:321
- command key (KA-KN, KP-KY)
 - See also* WORKSTN file
- conditioning output R:56
- general description R:51
- setting R:62
- conditioning calculations R:52
- conditioning file open R:257
- conditioning output R:56
 - controlling a record R:329
 - controlling fields of a record R:332
 - general information R:48
 - specification of R:329
- control level R:317
- control level (L1-L9)
 - as field record relation indicator R:49, R:309
 - as record identifying indicator R:302, R:313
 - assigning to input fields R:309, R:313
 - conditioning calculations R:318
 - conditioning output R:329, R:332
 - examples R:38, R:42
 - general description R:36
 - rules for R:37, R:41
 - setting of R:62
- description R:33
- displaying while debugging PG:204
- error PG:227
- external (U1-U8)
 - as field indicator R:43
 - as field record relation indicator R:49, R:310
 - as record identifying indicator R:34
 - conditioning calculations R:318
 - conditioning output R:329
 - general description R:46
 - resetting R:46, R:310
 - rules for resetting R:46, R:49
 - setting R:62
- field
 - as halt indicators R:43
 - assigning on input specifications R:310, R:313
 - conditioning calculations R:318
 - conditioning output R:329
 - general description R:43
 - numeric R:44
 - rules for assigning R:43
 - setting of R:62
- field record relation
 - assigning on input specifications R:310
 - example R:50
 - general description R:49
 - rules for R:49

indicators *(continued)*

file conditioning R:48
 first page (1P)
 conditioning output R:329, R:333
 general description R:46
 restrictions R:46
 setting R:62
 with initialization subroutine (*INZSR) R:26
 function key (KA-KN, KP-KY)
 with WORKSTN file PG:334
 halt (H1-H9)
 as field indicator R:43
 as field record relation indicator R:49, R:310
 as record identifying indicator R:34
 as resulting indicator R:44, R:321
 conditioning calculations R:318
 conditioning output R:329, R:332
 general description R:52
 setting R:62
 used to end a program/procedure PG:154,
 PG:155
 internal R:44
 first page (1P) R:46
 last record (LR) R:47
 matching record (MR) R:47
 return (RT) R:47
 last record (LR)
 as record identifying indicator R:34, R:302, R:312
 as resulting indicator R:44, R:321
 conditioning calculations R:317, R:318
 conditioning output R:329, R:332
 general description PG:5, R:47
 setting R:62
 used to end a program/procedure PG:154,
 PG:155
 level zero (L0)
 calculation specification R:52, R:316
 matching record (MR)
 See also multfile processing
 as field record relation indicator R:49, R:310
 assigning match fields R:97
 conditioning calculations R:318
 conditioning output R:329, R:332
 general description R:47
 setting R:62
 on RPG IV specifications R:33
 output
 AND/OR lines R:332
 assigning R:329
 examples R:57, R:58
 general description R:57
 restriction in use of negative indicators R:57,
 R:329
 overflow
 assigning on file description specifications R:260
 conditioning calculations R:52, R:318
 conditioning output R:329, R:332

indicators *(continued)*

overflow *(continued)*
 examples PG:319, PG:320
 fetch overflow logic PG:320, R:28
 general description PG:316, R:34
 presence or absence of PG:317
 relation to program cycle PG:320
 setting of PG:320, R:62
 with exception lines R:330, R:515
 with PRINTER file PG:316
 passing or not passing R:260
 record identifying
 assigning on input specifications R:35
 conditioning calculations R:318
 conditioning output R:329, R:332
 general description R:34
 rules for R:35
 setting on and off R:62
 summary R:61
 with file operations R:34
 return (RT) R:47
 as field indicator R:43
 as record identifying indicator R:312
 as resulting indicator R:44, R:321
 conditioning calculations R:318
 conditioning output R:56
 used to end a program/procedure PG:154,
 PG:155
 rules for assigning R:35
 rules for assigning resulting indicators R:43
 saving for attached device R:263
 setting of R:62
 status
 program exception/error R:82
 summary chart R:61
 used as data R:59
 using PG:5, R:48
 when set on and set off R:62

INFDS

See file information data structure

INFDS keyword R:259

information operations

DUMP (program dump) R:435, R:509
 general information R:435
 SHTDN (shut down) R:435, R:635
 TIME (time of day) R:435, R:654

INFSR

See file exception/error subroutine (INFSR)

INFSR keyword R:259

initialization

inside subprocedures R:92
 of arrays R:148
 of fields R:282
 overview R:114
 subroutine (*INZSR) R:26
 subroutine with RESET operation code R:609

initialization operations

- CLEAR (clear) R:483
- general information R:435
- RESET (reset) operation R:609

initialization subroutine (*INZSR)

- and subprocedures R:92
- description R:26
- with RESET operation code R:609

input

- file PG:341, R:245

input field

- as lookahead field R:303
- decimal positions R:308
- external name R:311
- format of R:307
- location R:307
- name of R:308
- RPG IV name of R:312

input record

- unblocking PG:287

input specifications

- See also indicators
- control level indicators R:313
- external field name R:312
- field indicators R:313
- general description PG:3
- location and size of field R:307
- match fields R:313
- record identifying indicator R:312
- record name R:311
- RPG IV field name R:312

input specifications for program described file

- field
 - decimal positions R:308
 - format R:307
 - name R:308
- filename R:300
- indicators
 - control level R:309
 - field R:307
 - field record relation R:310
 - record identifying R:302
- lookahead field R:303
- number of records R:302
- option R:302
- record identification codes R:303
- sequence checking R:301

inquiry messages

- list of PG:108
- replying to PG:108

inserting records during a compilation R:12

inserting specification templates PG:390

INSRTPL parameter PG:386, PG:390

integer arithmetic R:420

integer format

- alignment of fields R:124, R:168, R:274

integer format (continued)

- arithmetic operations R:420
- considerations for using R:171
- converting to R:372
- definition R:168
- definition specification R:272
- editing an unsigned field R:214
- editing unsigned field R:204
- integer arithmetic R:420
- output specification R:335
- TRUNCNBR parameter PG:416

Integrated Language Environment (ILE)

- effect on
 - OPM-compatible program PG:24
 - program using CRTBNDRPG PG:26
- ending an ILE program PG:109
- family of ILE compilers PG:17
- interlanguage calls PG:149
- internal structure of program PG:81
- overview PG:17
- program call PG:19
- program creation PG:17
- program creation strategies PG:23, PG:25, PG:27
- program management PG:19

interlanguage calls PG:149

intermediate results in expressions R:404

internal data format

- arithmetic operations R:420
- default date R:233
- default formats R:158
- default time R:241
- definition R:157
- definition specification R:272
- for external subfields R:123

internal indicators R:44

- first page (1P) R:46
- last record (LR) R:47
- matching record (MR) R:47
- return (RT) R:47

Intersystem Communications Function (ICF) PG:331

INVITE DDS keyword R:594

invoke subroutine (EXSR) operation code R:519

INZ keyword

- description R:282

INZSR

- See initialization subroutine (*INZSR)

ITER (iterate) operation code R:422, R:445, R:531

J

job attributes

- See *JOB RUN

K

key

See also search argument
 composite PG:286
 for a record or a file PG:285
 partial PG:286

key field

alphanumeric R:250
 for externally described file R:251
 format of R:251
 graphic R:251
 length of R:250
 packed R:251
 starting location of R:259

key field information in compiler listing PG:433

keyed processing

access path PG:282
 indexed file PG:288, R:253
 record-address limits file PG:290
 sequential R:265
 sequential-within-limits PG:300
 specification of keys R:251

keyed-sequence access path PG:282

KEYLOC keyword R:259

keywords

ALT R:229
 DDS PG:281
 EXPORT PG:84
 for continuation line PG:281
 CLEAR PG:334
 HELP PG:334
 HOME PG:334
 PRINT PG:334
 ROLLDOWN PG:334
 ROLLUP PG:334
 for display device file
 CLEAR PG:334
 HELP PG:334
 HOME PG:334
 PRINT PG:334
 ROLLDOWN PG:334
 ROLLUP PG:334
 for program status data structure R:82
 *ROUTINE R:83
 *STATUS R:83
 NOOPT PG:87, PG:226
 *OMIT PG:142
 syntax R:220

KFLD (define parts of a key) operation code R:91, R:432, R:533

KLIST (define a composite key) operation

code R:91, R:432, R:534
 name, rules for R:4

L

L0 indicator

See level zero (L0) indicator

label, rules for R:5

LANGID keyword R:238

LANGID parameter

CRTBNDRPG command PG:58, PG:414
 CRTRPGMOD command PG:74
 specifying on control specifications R:238

languages, ILE PG:17

last program cycle R:19

last record (LR) indicator

as record identifying indicator R:302, R:312
 as resulting indicator R:44, R:321
 conditioning calculations
 positions 7 and 8 R:316, R:317
 positions 9-11 R:318
 conditioning output R:329, R:332
 general description R:47
 in calculation specification R:317
 setting R:62
 used to end a program/procedure PG:154, PG:155

leading blanks, removing R:393, R:394

LEAVE (leave a structured group) operation code R:422, R:445, R:536

length notation R:123, R:271

length of form for PRINTER file R:257

length of record in a file, conversion considerations PG:380

length, get using %LEN R:373

less than operator (<) R:402

less than or equal operator (<=) R:402

level checking PG:270

level zero (L0) indicator

calculation specification R:316
 calculation specifications R:52

library, creating PG:51

LIKE keyword R:123, R:283

limits processing, file description specifications R:249

limits records PG:283

line skipping R:327

line spacing R:327

listing view, creating PG:168

listing, binder

as maintenance resource PG:86
 basic PG:100
 creating PG:85
 determining exports in service program PG:91
 sections of PG:85

listing, compiler

additional diagnostic messages PG:68
 browsing using SEU PG:68
 coordinating listing options with debug view options PG:69

listing, compiler *(continued)*

- correcting compilation errors PG:66
- correcting run-time errors PG:68
- default information PG:63
- in-line diagnostic messages PG:67
- indenting structured operations PG:65
- obtaining PG:63
- reading PG:423
- sample listing PG:424
- sections of PG:63, PG:424
- specifying the format of PG:64
- using PG:63
- using as documentation PG:69

literals

- alphanumeric R:114
- character R:115
- date R:117
- graphic R:117
- hexadecimal R:115
- indicator format R:115
- numeric R:116
- time R:117
- timestamp R:117

local data area R:494

local variable

- in formatted dump PG:258
- scope R:91, R:112
- static storage R:294

locking

- file PG:275
- read without locking PG:276
- record locking wait time PG:276
- retry on timeout PG:276
- standalone PG:276
- under commitment control PG:308
- UNLOCK PG:276

locking/unlocking a data area or record R:656

log file

- about PG:381
- DDS for PG:396
- using PG:396

LOGFILE parameter PG:387

logic cycle, RPG

- detail R:23
- general R:19

logical file

- See also* DISK file
- general PG:281
- multi-format PG:281

logical relationship

- calculation specifications R:317
- input specifications R:305
- output specifications R:327, R:338

LOGMBR parameter PG:387

look-ahead function R:29

lookahead field R:303

LOOKUP (look up) operation code R:421

- arrays/tables R:538

loop, avoiding in an error subroutine PG:236

LR (last record) indicator

- See* last record (LR) indicator

M

M1-M9 (match field values) R:98

main procedure

- coding considerations PG:46
- overview PG:33
- returning from PG:153
- scope of files PG:80
- scope of parameters R:112
- specifications for R:217

main source section

- description R:217
- specifications for R:218

maintaining OPM compatibility PG:61, PG:111

major/minor return codes R:79

managing activation groups PG:109

managing programs PG:19

managing run-time storage PG:113

managing the default heap using RPG operations PG:113

manual code conversion PG:398

match fields

- See also* multifile processing
- alternate collating sequence R:163
- assigning values (M1-M9) to R:98
- description R:97
- dummy match field R:99, R:101
- example R:99, R:100
- in multi-file processing R:97
- input specifications for R:309, R:313
- logic R:27
- used for sequence checking R:98

match levels (M1-M9) R:98

matching record (MR) indicator

- See also* multifile processing
- as field record relation indicator R:49, R:310
- assigning match fields R:309, R:313
- conditioning calculations
 - positions 7 and 8 R:316
 - positions 9-11 R:318
- conditioning output R:329, R:332
- general description R:47
- setting R:62

MAXDEV keyword R:259

maximum number of devices R:259

maximum number of files allowed R:243

memory management operations

- ALLOC (allocate storage) R:456
- ALLOC (allocate storage) operation code R:436

memory management operations *(continued)*

- DEALLOC (free storage) R:490
- DEALLOC (free storage) operation code R:436
- general information R:436
- REALLOC (reallocate storage with new length) R:607
- REALLOC (reallocate storage with new length) operation code R:436

message identification R:506

message operations

- DSPLY (display function) R:437, R:506
- general information R:437

message summary in compiler listing PG:435

messages

- additional diagnostic PG:68
- exception
 - example PG:224
 - types of PG:218
 - unhandled PG:223
- in-line diagnostic PG:67
- inquiry
 - replying to PG:108

MHHZO (move high to high zone) operation code R:443, R:541

MHLZO (move high to low zone) operation code R:443, R:542

migrating to ILE RPG PG:379

See also converting to RPG IV

MLHZO (move low to high zone) operation code R:443, R:543

MLLZO (move low to low zone) operation code R:443, R:544

modifying a module PG:86

modifying an existing record R:658

module

- about PG:73
- behavior of bound ILE RPG PG:80
- binding into a program PG:81
- binding multiple PG:84
- changing optimization level PG:87
- changing while debugging PG:175
- creating PG:73
- creating a NOMAIN module PG:75
- CRTRPGMOD command PG:74
- determining the entry module PG:82
- different debug views PG:176
- effect of debug data on size PG:166
- information in dump listing PG:251
- modifying and rebinding PG:86
- NOMAIN R:95, R:238
- overview of multiple-procedure module PG:33
- preparing for debugging PG:166
- reducing size PG:88
- related CL commands PG:80
- relationship to program PG:81
- removing observability PG:88

module *(continued)*

- replacing in a program PG:87
- viewing source while debugging PG:174

module creation

- general discussion PG:73
- using CRTRPGMOD PG:74
- using CRTRPGMOD defaults PG:75

module observability PG:88

MODULE parameter PG:82

- CRTBNDRPG command PG:408
- CRTRPGMOD command PG:74

move array (MOVEA) operation code R:559

move high to high zone (MHHZO) operation code R:541

move high to low zone (MHLZO) operation code R:542

move left (MOVEL) operation code R:565

move low to high zone (MLHZO) operation code R:543

move low to low zone (MLLZO) operation code R:544

MOVE operation code R:438, R:545

move operations

- general information R:438
- MOVE R:438, R:545
- MOVEA (move array) R:438, R:559
- MOVEL (move left) R:438, R:565

move remainder (MVR) operation code R:576

move zone operations

- general information R:443
- MHHZO (move high to high zone) R:443, R:541
- MHLZO (move high to low zone) R:443, R:542
- MLHZO (move low to high zone) R:443, R:543
- MLLZO (move low to low zone) R:443, R:544

MOVEA (move array) operation code R:421, R:438, R:559

MOVEL (move left) operation code R:438, R:565

moving character, graphic, and numeric data R:438

moving date-time fields R:439

moving the remainder R:576

moving zones R:541

MR (matching record) indicator

See matching record (MR) indicator

MULT (multiply) operation code R:418, R:575

multifile logic R:27

multifile processing

- assigning match field values R:98
- FORCE operation code R:523
- logic R:27
- match fields R:97
- no match fields R:97
- normal selection, three files R:101, R:102

multiple devices attached to application

program PG:310

multiple-device file

- WORKSTN PG:342

multiplication operator (*) R:402
multiply (MULT) operation code R:575
multiplying factors R:575
MVR (move remainder) operation code R:418,
 R:576

N

name(s)

array R:4
 conditional compile R:4
 data structure R:4
 EXCEPT R:4, R:330
 field R:4
 on input specifications R:308, R:312
 on output specifications R:329
 file R:4
 for *ROUTINE
 with program status data structure R:82
 KLIST R:4
 labels R:5
 PLIST R:5
 prototype R:5
 record R:5
 rules for R:4
 subfield R:4
 subroutine R:5
 symbolic R:3
 table R:5

named activation group PG:110

named constant

defining a value using CONST R:276
 definition keyword summary R:296
 specifying R:119

named constants R:119

National Language Support (NLS) of source

debugger PG:211

negative balance (CR)

with combination edit code R:202

nested DO-group

example R:447

nested exceptions PG:223

nesting /COPY directives R:13

***NEW** PG:110

NEXT (next) operation code R:433, R:577

no debug data PG:166

NOMAIN keyword R:238

NOMAIN module R:95

coding considerations PG:46
 creating PG:75
 main source section R:217

nonkeyed processing PG:304, R:251

NOOPT keyword

and handling exceptions PG:226
 description R:284
 maintaining current values while debugging PG:164

NOOPT keyword (continued)

program optimization level PG:87

normal codes

file status R:77
 program status R:85

normal program cycle R:19

normal program/procedure end PG:154

NOT

as a special word R:7
 as operator in expressions R:402
 Behavioral difference between ILE RPG and
 RPG/400 PG:373

not equal operator (<>) R:402

null value support

ALWNULL(*NO) R:195
 description R:190
 displaying null-capable fields PG:206
 input-only R:195
 user controlled R:190
 input R:191
 keyed operations R:193
 output R:191
 query or set null indicator R:376

null-terminated string

get or store R:389
 passing R:286

number

of records for program described files R:302

number of devices, maximum R:259

number of elements

defining using DIM R:278
 determining using %ELEM R:363
 per record R:293

numeric data type

allowed formats R:165
 binary R:165
 considerations for using R:171
 float R:166
 integer R:168
 packed-decimal R:169
 representation R:172
 unsigned R:170
 zoned-decimal R:170

numeric fields

format R:157, R:170
 moving R:438
 punctuation R:201
 resetting to zeros R:334

numeric literals

considerations for use R:116
 length of R:373

O

observability PG:88

obtaining a compiler listing PG:63
 obtaining conversion reports PG:389
 OCCUR (set/get occurrence of a data structure)
 operation code R:578
 OCCURS keyword R:285
 OFL PG:238
 file exception/error subroutine (INFSR) R:80
 flowchart R:22
 program exception/errors R:83
 OFLIND keyword R:259
 *OMIT PG:141, PG:142
 omitted parameters PG:141
 *OMIT PG:142
 prototyped R:286
 one-step process of program creation PG:57
 online information
 for create commands PG:408
 for ILE source debugger PG:165
 OPDESC keyword R:286
 OPEN (open file for processing) operation
 code R:433, R:582
 specifications for R:582
 open data path
 sharing PG:277
 opening file for processing R:582
 conditional R:257
 OPEN operation code R:582
 user-controlled R:265
 operation codes PG:341
 allowed with DISK file PG:306
 allowed with PRINTER file PG:316
 allowed with sequential file PG:326
 allowed with SPECIAL file PG:328
 allowing 'E' extender PG:228
 allowing error indicators PG:228
 general discussion PG:6
 operation extender R:318, R:322
 operational descriptors
 definition PG:140
 example PG:94
 minimal R:379
 OPDESC keyword R:286
 operations, in calculation specification R:318,
 R:322
 operator precedence rules R:399
 operators
 binary R:399
 unary R:399
 OPM compatibility, maintaining PG:61, PG:111
 OPM default activation group PG:23, PG:31
 running in PG:111
 optimization
 definition PG:87
 effect on fields when debugging PG:164
 exception handling considerations PG:226
 level of
 changing a object's PG:87

optimization (continued)
 level of (continued)
 checking PG:88
 preventing R:284
 OPTIMIZE keyword R:239
 OPTIMIZE parameter
 CRTBNDRPG command PG:58, PG:412
 CRTRPGMOD command PG:74
 specifying on control specifications R:239
 OPTION keyword R:239
 OPTION parameter
 coordinating listing and debug view options PG:168
 coordinating with debug view options PG:69
 CRTBNDRPG command PG:58, PG:410
 CRTRPGMOD command PG:74
 specifying on control specifications R:239
 using PG:64, PG:69
 OPTIONS keyword
 *NOPASS PG:142, R:286
 *OMIT PG:142, R:286
 *STRING R:286
 *VARSIZE R:286
 OR lines
 on calculations R:318
 on input specifications R:306
 on output specifications R:327, R:338
 order of evaluation
 in expressions R:411
 on prototyped call PG:148
 ORxx operation code R:427, R:445, R:584
 OTHER (otherwise select) operation code R:445,
 R:585
 otherwise select (OTHER) operation code R:585
 OUT (write a data area) operation code R:429,
 R:586
 output
 conditioning indicators R:56, R:329
 field
 format of R:338
 name R:332
 file R:245
 record
 end position in R:334
 specifications
 *ALL R:339
 ADD records for externally described files R:339
 AND/OR lines for externally described
 files R:338
 DEL (delete) records for externally described
 files R:339
 detail record for program described file R:327
 EXCEPT name for externally described
 files R:339
 externally described files R:338
 field description control R:325
 field name R:339
 file name for program described file R:326

output *(continued)*

specifications *(continued)*

- for fields of a record R:332
- for records R:326
- general description R:325
- indicators for externally described files R:338
- program-described WORKSTN file PG:339
- record identification and control R:325
- record name for externally described files R:338
- record type for externally described files R:338
- specification and entry R:326

output buffer positions, in compiler listing PG:430

output file PG:341

OUTPUT parameter

- CRTBNDRPG command PG:58, PG:412
- CRTRPGMOD command PG:74
- using PG:63

output record

- blocking PG:287

output specifications

- constant/editword field R:225
- example PG:8
- for program described file
 - *IN, *INxx, *IN(xx) R:333
 - *PLACE R:333
 - ADD record R:327
 - AND/OR lines for program described file R:327
 - blank after R:334
 - conditioning indicators R:329
 - DEL (delete) record R:327
 - edit codes R:333
 - end position of field R:334
 - EXCEPT name R:330
 - exception record for program described file R:327
 - PAGE, PAGE1-PAGE7 R:332
 - UPDATE R:332
 - UDAY R:332
 - UMONTH R:332
 - UYEAR R:332
- general description PG:3
- program-described WORKSTN file PG:339
- with external descriptions PG:269

output spooling PG:279

overflow

- indicators PG:317
- line, indicating length of R:218
- page PG:316

overflow indicators

- assigning on file description specifications R:259
- conditioning calculations R:52, R:318
- conditioning output PG:317, R:329
- examples PG:319, PG:320
- fetch overflow logic R:28
- fetch-overflow logic PG:320
- general description PG:317, R:34

overflow indicators *(continued)*

- presence or absence of PG:317
- relation to program cycle PG:320
- setting of PG:320, R:62
- with exception lines R:330, R:510
- with PRINTER file PG:316

overlapping control fields R:38

OVERLAY keyword R:123, R:291

overlaying storage in data structures R:123, R:291

overrides, file PG:267

- example PG:274
- general discussion PG:273, PG:304
- indicated in compiler listing PG:425

overriding external description PG:267

P

packed decimal format

- array/table field R:169
- converting to R:356
- definition specification R:272
- description R:169
- input field R:169
- keys R:251
- output field R:169
- specifying even number of digits R:293

PACKEVEN keyword R:169, R:293

page headings PG:64

page number, in PRINTER file PG:316

page numbering R:9

See also PAGE, PAGE1-PAGE7

page overflow, in PRINTER file PG:316

PAGE, PAGE1-PAGE 7 R:333

parameter descriptions

- CRTBNDRPG command PG:408
- CRTRPGMOD command PG:421
- CVTRPGSRC command PG:384

parameter list

- See also* PARM (identify parameters) operation code created by PARM PG:153
- identifying PG:133
- rules for specifying PG:153

parameter table

- CRTBNDRPG command PG:58
- CRTRPGMOD command PG:74
- CVTRPGSRC command PG:383

parameters

- checking number passed PG:143
- match data type requirements PG:140
- omitted PG:141
- operational descriptors PG:140
- passing PG:133
- passing using the CL CALL command PG:103
- specifying PG:151

PARM (identify parameters) operation

code PG:104, R:432, R:587

PARM (identify parameters) operation code (*continued*)

calculation specifications R:587
 call operations R:422
 *OMIT PG:141, PG:142
 rules for specifying PG:152
 using PG:151

partial key PG:286

parts of an ILE RPG program PG:6

PASS keyword R:260

passing parameters

by read-only reference PG:139, R:276
 by reference PG:137
 by value PG:137, PG:138
 checking number passed PG:143
 example PG:104
 match data type requirements PG:140
 number of parameters R:379
 omitted parameters PG:141
 operational descriptors PG:140
 overview PG:133
 passing less data PG:148
 passing methods for ILE languages PG:149
 using PARM PG:151
 using PLIST PG:153
 using the CL CALL command PG:103
 with CONST keyword R:276

PEP

See program entry procedure (PEP)

percolate an exception

using a condition handler PG:240

percolation of an exception PG:218

performance considerations

arithmetic operations R:420
 subroutines vs. subprocedures PG:94

performance tips

call for LR-on PG:374
 program call PG:154

performing a quick conversion PG:387

performing a trial conversion PG:388

PERRCD keyword R:293

PGM parameter

CRTBNDRPG command PG:58

PGMNAME keyword R:260

physical file PG:281

PIP (Program Initialization Parameters) data

area R:494
 DEFINE (field definition) R:492
 IN (retrieve a data area) R:529
 OUT (write a data area) R:586
 UNLOCK (unlock a data area or record) R:656
 UNLOCK (unlock a data area) R:656

PLIST (identify a parameter list) operation

code PG:104, R:91, R:432, R:590

*ENTRY PLIST PG:153, R:590
 calculation specifications R:590

PLIST (identify a parameter list) operation code (*continued*)

call operations R:422
 for SPECIAL file R:260
 name, rules for R:5
 using PG:153

PLIST keyword R:260

pointers

basing pointer
 alignment R:182
 alignment of subfields R:124
 as result of %ADDR R:353
 comparison to *NULL R:429
 creating R:276
 data type R:182
 example R:183
 problems comparing pointers R:429, R:636
 built-in functions
 %ADDR (Get Address of Variable) R:353
 %PADDR (Get Procedure Address) R:378
 data type R:272
 pointer arithmetic R:183
 procedure pointer
 address of procedure entry point R:378
 alignment of subfields R:124
 data type R:188
 example R:188
 PROCPTR keyword R:294

position of record identification code R:304

positional notation R:123, R:271

POST (Post) operation code R:433, R:592

contents of file feedback information after use R:67

Power Down System (PWRDWN SYS) R:595

power operator R:402

precedence rules of expression operators R:399

precision of expression results

"Result Decimal Position" example R:409
 default example R:406
 intermediate results R:405
 precision rules R:404
 using the "Result Decimal Position" rules R:408
 using the default rule R:405

PREFIX keyword PG:266

definition specification R:123, R:293
 file description specification R:261

prefixing a name to a subfield R:123, R:293

preparing a program for debugging PG:166

prerun-time array or table

See also array
 coding R:146
 example of R:145
 input file name R:282
 number of elements per record R:293
 output file name R:294
 rules for loading R:147
 specifying external data format R:279

Presentation Graphics Routines (PGR) PG:158

prevent printing over perforation R:28

preventing printing over perforation PG:320

PRFDTA keyword R:240

PRFDTA parameter

CRTBNDRPG command PG:58, PG:419

CRTRPGMOD command PG:74

removing observability PG:88

specifying on control specifications R:240

primary file

ending a program without R:29

file description specifications R:246

general description R:246

print command key PG:334

printer control data structure R:261

PRINTER file

access current line value PG:323

device name R:254

fetch overflow logic R:28

fetch-overflow logic PG:320

file operation codes allowed PG:316

length of form R:257

maximum number of files allowed in program PG:316

modify forms control PG:323

overflow indicators PG:316

page overflow PG:316

PRTCTL (printer control) PG:323

procedure

abnormal ending PG:155

address of procedure entry point R:378

calling PG:129

dump information PG:251

exported R:12

external prototyped name R:281

normal ending PG:154

passing parameters PG:133

procedure pointer call PG:130

procedure specification R:341

PROCPTR keyword R:294

returning from PG:153

returning without ending PG:155

static procedure call PG:130

stepping over PG:195

procedure interface

defining R:90, R:139, R:341

definition keyword summary R:297

definition type entry R:270

procedure pointer calls PG:130

procedure specification

begin/end entry R:343

form type R:342

general R:341

keywords R:343

name R:342

procedure specification keywords

EXPORT R:343

processing methods

consecutive PG:292

for DISK file PG:291, R:265

nonkeyed PG:304

random-by-key PG:299

relative-record-number PG:303

sequential only PG:293, PG:304

sequential-by-key PG:293

sequential-within-limits PG:300

WORKSTN file PG:335, PG:341

PROCPTR keyword R:294

program

abnormal ending PG:155

advanced ILE PG:30

binding modules PG:81

calling PG:129, PG:130

calling using expressions PG:136

calling using the CALL operation PG:150

calling using the CALLP operation PG:135

changing PG:86

changing optimization level PG:87

changing while debugging PG:175

different debug views PG:176

effect of debug data on size PG:166

ending PG:109

entering source PG:51

entering SQL statements PG:55

example PG:6

freeing resources PG:112

internal structure PG:81

mixed-language PG:29

multiple-module

general creation strategy PG:27

normal ending PG:154

OPM-compatible

creation method PG:23

effect of ILE PG:24

example PG:24

program creation strategy PG:23, PG:31

passing parameters PG:133

preparing for debugging PG:166

program entry procedure PG:81

reducing size PG:88

related CL commands PG:85

removing observability PG:88

returning from PG:153

returning without ending PG:155

running PG:103

running from a menu-driven application PG:106

running in the OPM default activation group PG:111

running using a user-created command PG:108

setting watch conditions PG:188

single-language PG:28

effect of ILE PG:26

program (*continued*)

status, codes R:85
 status, exception/error codes R:85
 stepping into PG:196
 stepping over PG:195
 stepping through PG:194
 updating PG:87
 viewing source while debugging PG:174

program activation PG:109

program creation

coding considerations PG:45, PG:46
 examples of PG:59, PG:60, PG:61, PG:84
 OPM-compatible
 creating PG:23
 strategy to avoid PG:31
 strategies for PG:23
 CRTPGM command PG:82
 ILE application using CRTRPGMOD PG:27
 OPM-compatible PG:23
 strategy to avoid PG:31
 using CRTBNDRPG PG:25
 using CRTRPGMOD and CRTPGM PG:73
 using the one-step process PG:57

program cycle

commitment control PG:312
 defined R:19
 detail R:23
 detailed description R:23
 fetch overflow logic PG:320, R:28
 general R:19
 general description PG:4, R:19
 last cycle PG:5
 programmer control R:29
 with initialization subroutine (*INZSR) R:26
 with lookahead R:29
 with match fields R:27
 with RPG IV exception/error handling R:29

program described files, field description and control entries, output specifications

blank after R:334
 constant or edit word R:337
 data format R:335
 edit codes R:333
 end position R:334
 field name R:332
 output indicators R:332

program described files, field description entries, input specifications

data format R:307
 field location R:307
 general description R:306

program described files, record identification and control entries, output specifications

EXCEPT name R:330
 fetch overflow/release R:328
 file name R:326

program described files, record identification and control entries, output specifications (*continued*)

logical relationship R:327
 output indicators R:329
 record addition/deletion R:327
 skip after R:332
 skip before R:331
 space after R:331
 space and skip R:331
 space before R:331
 type R:327

program described files, record identification entries, input specifications

file name R:300
 general description R:300
 logical relationship R:301
 number R:302
 option R:302
 record identification codes R:303
 record identifying indicator, or ** R:302
 sequence R:301
 summary tables R:300

program device, specifying name R:256

program dump (DUMP) operation code R:509

program ending, without a primary file R:29

program entry procedure (PEP)

and the call stack PG:131
 definition PG:81
 determining PG:82

program exception/error subroutine

and subprocedures R:92
 description PG:233
 example PG:233

program exception/errors

avoiding a loop PG:236
 definition PG:220
 example PG:233, PG:239
 general information

indicators in positions 73 and 74 R:82
 indicators in positions 56 and 57 of calculation specifications R:65, R:82
 data structure R:82
 status information R:82
 return point entries R:80
 *CANCL R:80, R:83
 *DETC R:80, R:83
 *DETL R:80, R:83
 *GETIN R:80, R:83
 *OFL R:80, R:83
 *TOTC R:80, R:83
 *TOTL R:80
 blanks R:80, R:83
 subroutine R:87
 using a *PSSR subroutine PG:233

program generation R:227

program management PG:19

program name

- *FROMMBR parameter PG:386
- default R:235
- external prototyped name R:280
- for SPECIAL file R:260

program running R:227

program status data structure

- *ROUTINE R:82
- *STATUS R:82
- contents R:83
- defining R:125
- example PG:151, PG:233
- general information R:82
- predefined subfield R:82
- status codes R:85
- subfields
 - predefined R:82
- using in an error subroutine PG:233
- with OCCUR operation code R:578

program-described file

- as DISK file PG:288
- as WORKSTN file PG:338, PG:339, PG:340
- date-time data format R:159
- definition PG:263
- entries on
 - file description specifications R:243
 - input specifications R:299, R:300
 - output specifications R:325
- in output specification R:326
- length of key field R:250
- length of logical record R:249
- numeric data format R:158
- physical and logical files PG:281
- record identification entries R:300
- valid search arguments PG:288

program/procedure call

- abnormal program/procedure end PG:155
- call stack PG:131
- calling bindable APIs PG:157
- calling graphics PG:158
- calling procedures PG:130
- calling programs PG:130
- calling special routines PG:158
- fixed-form call PG:150
- free-form call PG:135, PG:136
- interlanguage calls PG:149
- normal program/procedure end PG:154
- operational descriptors R:424
- overview PG:129
- parameter passing methods PG:137
- prototyped call R:423
- recursive calls PG:132
- returning from a called program or procedure PG:153
- returning values PG:136

program/procedure call (continued)

- returning without ending PG:155
- static calls PG:130
- using the CALL operation PG:150
- using the CALLB operation PG:150
- within ILE PG:19

program/procedure end

- abnormal end PG:155
- after system call PG:109
- normal end PG:154
- return overview PG:153
- returning without ending PG:155
- using bindable APIs PG:156

programmer control of file processing R:29

programming tips R:227, R:590

- /EOF directive R:17
- checking parameter interface R:587
- creating NOMAIN module PG:92
- displaying copyright information R:231
- exported procedures R:12
- improving call performance R:82
- nested /COPY R:13
- reducing size of module R:95
- setting subprocedure breakpoints PG:196
- using prototypes R:139, R:269, R:284

prologue section of compiler listing PG:424

prototype

- and subprocedures R:89
- defining R:136
- definition keyword summary R:297
- definition type entry R:270
- description PG:34, R:423
- using PG:135

prototyped call

- defining R:136
- order of evaluation of parameters PG:148
- using call operations R:423

prototyped parameters

- defining R:137
- definition keyword summary R:297
- omitting on call R:286
- OPTIONS keyword R:286
- passing *OMIT R:286
- passing string shorter than defined length R:286
- requesting operational descriptors R:286
- VALUE keyword R:295

prototyped program or procedure

- as built-in function R:347
- calling in an expression R:424
- CALLP (call a prototyped procedure) R:466
- number of passed parameters R:379
- procedure specification R:341
- prototyped call PG:34, R:423
- RETURN (return to caller) R:616
- specifying external procedure name R:281
- specifying external program name R:280

PRTCTL (printer control)
 example PG:324
 general information PG:323
 specifying R:261
 with space/skip entries R:331
PRTCTL keyword R:261
PWRDWN SYS (Power Down System) R:595

Q

QSYSOPR R:506
QUAL debug command
 definition PG:164
 ILE RPG PG:208
querying names of called programs/procedures PG:150
queues
 *EXT (external message) R:506
 QSYSOPR R:506

R

RAFDATA keyword R:262
random retrieval from a file based on record number or key value (CHAIN)
 operation code R:474
 RECNO keyword R:262
random-by-key processing
 example PG:299
 general discussion PG:299
RCLACTGRP command
 See Reclaim Activation Group (RCLACTGRP) command
RCLRSC command
 See Reclaim Resources (RCLRSC) command
READ (read a record) operation code R:433, R:594
READC (read next modified record) operation code R:433, R:597
READE (read equal key) operation code R:433, R:599
reading a record PG:342, R:594
 specifications for R:594
reading next record
 specifications for R:597
 with WORKSTN subfile PG:337
reading prior record R:599
READP (read prior record) operation code R:433, R:602
READPE (read prior equal) operation code R:433, R:604
REALLOC (reallocate storage with new length) operation code PG:113, R:436, R:607
Reallocate Storage (CEEZST) bindable API PG:20
reallocate storage (REALLOC) operation code R:607

reallocating storage R:607
rebinding PG:86
Reclaim Activation Group (RCLACTGRP) command
 deleting activation groups PG:112
 named activation groups PG:110
Reclaim Resources (RCLRSC) command
 ILE program PG:26
 OPM-compatible program PG:24
 to free storage PG:112
RECNO keyword R:255, R:262
 with relative-record-number processing PG:303
record
 adding to a file R:248, R:327
 deleting from a file R:327, R:496
 detail (D) R:327
 exception (E) R:327
 with EXCEPT operation code R:515
 externally described R:338
 heading (H) R:327
 input specifications
 externally described file R:311
 program described file R:300
 length R:249
 limits PG:291
 locking PG:276
 output specifications
 externally described R:338
 program described R:326
 record line R:326
 releasing PG:276
 renaming R:263
 total (T) R:327
 valid keys PG:285
record address field, length R:250
record address file
 conversion problems PG:393, PG:400
 description R:246
 file description specifications entry R:245
 format of keys R:250
 length of record address field R:250
 RAFDATA keyword R:262
 RECNO keyword R:262
 relative-record number PG:290, R:253
 restrictions R:246
 S/36 SORT files R:249
 sequential-within-limits PG:290, R:249
 with limits records PG:291
 with relative record numbers PG:290
record address limits file
 See record address file
record address relative record number file
 See record address file
record address type R:250
record blocking R:255
record format
 for a subfile PG:335, R:264

ILE RPG for AS/400 Combined Index

record format *(continued)*

- ignoring PG:266, R:258
- including R:258
- renaming PG:266, R:263
- resetting R:610
- specifications for externally described file PG:282
- writing to a display R:264

record identification codes R:303

- for input specification R:312

record identification entries

- in output specification R:326
- input specifications R:300, R:311
- output specifications R:326, R:338

record identifying indicators (01-99, H1-H9, L1-L9, LR, U1-U8, RT)

- assigning on input specifications
 - for externally described file R:311
 - for program described file R:300
 - rules for R:35
- conditioning calculations R:316, R:318
- conditioning output R:329, R:332
- for input specification R:312
- for program described files R:302
- general description R:34
- setting on and off R:62
- summary R:61
- with file operations R:34

record length of files, conversion

- considerations PG:380

record line R:326

record locking PG:276

record name

- for externally described input file R:311
- for externally described output file R:338
- rules for R:5

records, alternate collating sequence table R:164

records, file translation table R:106

recursion

- calling condition handlers PG:239
- recursive calls PG:46, PG:132

redirection, file

- definition PG:262
- general description PG:262

reducing object size PG:88, PG:166

Register Call Stack Entry Termination User Exit Procedure (CEERTX) PG:244

Register ILE Condition Handler (CEEHDLR)

- API PG:239

REL (release) operation code R:433, R:608

relative record number record address file

- See record address file

relative-record number PG:291

relative-record-number processing PG:303

Release (output specifications) R:339

release (REL) R:608

release, output specifications R:328

releasing a locked record PG:276

removing blanks from a string R:393

removing breakpoints

- about PG:178
- all PG:187
- conditional job breakpoints PG:181
- conditional thread breakpoints PG:187
- unconditional job breakpoints PG:179
- unconditional thread breakpoints PG:180
- using statement numbers PG:185

removing objects from a debug session PG:172

removing observability PG:88

RENAME keyword PG:266, R:263

renaming field names PG:266

renaming fields PG:266, R:261

renaming record-format names PG:266

renaming subfields R:123, R:279

REPLACE parameter

- CRTBNDRPG command PG:58, PG:414
- CRTRPGMOD command PG:74

replacing modules in a program PG:87

reply list of messages

- adding to PG:108
- changing PG:109

replying to run-time inquiry messages PG:108

requester

- accessing with ID R:256

requirements of Conversion Aid PG:381

reserved words

- *ALL R:339
- *ALL'x..' R:120
- *ALLG'oK1K2i' R:120
- *ALLX'x1..' R:120
- *BLANK/*BLANKS R:120
- *CANCL PG:238, R:22, R:80
- *DATE, *DAY, *MONTH, *YEAR R:8
- *DETC PG:238, R:83
- *DETL PG:238, R:83
- *ENTRY PLIST R:587
- *GETIN PG:238, R:83
- *HIVAL/*LOVAL R:120
- *IN R:59
- *IN(xx) R:59
- *INIT R:83
- *INxx R:60
- *INZSR R:23
- *LDA R:494
- *NOKEY R:483
- *NULL R:120
- *OFL PG:238, R:83
- *ON/*OFF R:120
- *PDA R:494
- *PLACE R:333
- *ROUTINE R:83
- *STATUS R:83

reserved words *(continued)*

*TERM R:83
 *TOTC PG:238, R:83
 *TOTL PG:238, R:83
 *ZERO/*ZEROS R:120
 INFDS R:66
 PAGE R:333
 PAGE, PAGE1-PAGE7 R:9
 PAGE1-PAGE7 R:333
 UPDATE, UDAY, UMONTH, UYEAR R:8
RESET operation code R:114, R:435, R:609
reset value R:609
resetting variables R:609
result field
 length of R:320
 number of decimal positions R:320
 possible entries, in calculation specification R:320
resulting indicators (01-99, H1-H9, OA-OG, OV, L1-L9, LR, U1-U8, KA-KN, KP-KY, RT)
 See also individual operation codes
 as error indicators PG:227
 calculation specifications R:321
 general description R:44
 rules for assigning R:45
 setting of R:62
resume point PG:238
retrieval of data area
 explicit R:529
 implicit R:20, R:125
retrieval of record from full procedural file R:474
retrieve a data area (IN) operation code R:529
Retrieve Operational Descriptor Information (CEEDOD) PG:94
 example PG:141
 operational descriptors PG:141
retrieving randomly (from a file based on record number of key value) R:474
retry on a record lock timeout PG:276
RETURN (return to caller) operation code R:616
 call operations R:422
 returning a value R:91
 returning without ending PG:155
 role in abnormal end PG:155
 role in normal end PG:154
 with expressions R:397
return (RT) indicator
 as field indicator R:310, R:313
 as record identifying indicator R:302, R:312
 as resulting indicator R:44, R:321
 conditioning calculations R:318
 conditioning output R:329
 general description R:47
 setting of R:62
 used to end a program/procedure PG:154, PG:155
return point
 for program exception/error subroutine R:88

return points, specifying in ENDSR PG:238
return status parameter PG:328
return value
 data type R:616
 defining R:91
 RETURN (return to caller) R:616
 returning using expressions PG:136
RETURNCODE data area PG:70
returning from a called main procedure PG:153
returning from a called procedure PG:153
 RETURN (return to caller) R:616
returning from a main procedure PG:153
returning from a subprocedure PG:156
returning using ILE bindable APIs PG:156
returning without ending PG:155
ROLBK (roll back) operation code R:433, R:619
roll back (ROLBK) operation code R:619
rolldown command key PG:334
rollup command key PG:334
root source view, creating PG:167
RPG IV
 See also ILE RPG
 behavioral differences between RPG III PG:373
 converting to PG:23, PG:25, PG:379
 overview PG:3
 unsupported RPG III features PG:398
RPG logic cycle
 detail R:23
 general R:19
RPGLEHSPEC data area R:227
RT (return) indicator
 See return (RT) indicator
rules
 for naming objects R:3
run-time array
 allocating storage during run-time PG:120
 definition of R:142
 rules for loading R:143
 with consecutive elements R:144
 with scattered elements R:143
run-time errors, correcting with a compiler listing PG:68
run-time inquiry messages, replying to PG:108
run-time job attributes
 See *JOB RUN
run-time storage, managing PG:113
running a program
 See also program/procedure call
 differences between ILE RPG and OPM
 RPG/400 PG:374
 from a menu-driven application PG:106
 in the OPM default activation group PG:111
 overview PG:103
 using a user-created command PG:108
 using the CL CALL command PG:103

S

- S/36 SORT files** R:249
- SAA data types**
 - null value support R:190
 - variable-length fields R:195
- sample programs**
 - See examples
- SAVEDS keyword** R:263
- SAVEIND keyword** R:263
- SCAN (scan string) operation code** R:444, R:620
- scope**
 - *PSSR subroutine R:94
 - of definitions R:91, R:112
 - of files PG:80
- screen design aid (SDA)** PG:106
- search argument**
 - externally described file
 - description PG:285
 - referencing a partial key PG:286
 - valid PG:286
 - for record address type R:252
 - program-described file PG:288
- searching within a table** R:538
- searching within an array** R:538
- SECLVL parameter** PG:386
- secondary file**
 - file description specifications R:246
 - general description R:246
- SELECT (begin a select group) operation code** R:445, R:623
- SEQ file**
 - example PG:327
 - file operation codes allowed PG:326
 - general description PG:326
 - processing chart PG:326
 - restrictions PG:326
 - variable-length PG:326
- sequence**
 - ascending R:248
 - descending R:248
- sequence checking**
 - alternate collating sequence R:163
 - on input specifications PG:271, R:301
 - with match fields R:309
- sequential file** PG:290
- sequential-by-key processing**
 - examples PG:293
 - general discussion PG:293
- sequential-only processing** PG:292, PG:293
- sequential-within-limits processing**
 - examples PG:301
 - file description specifications entry R:249
 - general discussion PG:300
- service program**
 - adding to a debug session PG:172
 - service program (continued)*
 - binder language PG:97
 - binding with CRTBNDRPG PG:60
 - changing PG:93
 - creating PG:91
 - example PG:94
 - in advanced application PG:30
 - reasons for using PG:91
 - reclaiming resources PG:112
 - related CL commands PG:93
 - sample binder listing PG:100
 - strategies for creating PG:92
 - updating PG:100
- service program creation**
 - about PG:91
 - strategies PG:92
- set bits off (BITOFF) operation code** R:459
- set bits on (BITON) operation code** R:460
- SET debug command**
 - definition PG:164
- set on and set off operation codes** R:435
- set/get occurrence of data structure** R:578
- SETGT (set greater than) operation code** R:433, R:625
- SETLL (set lower limits) operation code** R:433, R:629
- SETOFF (set off) operation code** R:435, R:633
- SETON (set on) operation code** R:435, R:634
- setting breakpoints**
 - about PG:178
 - conditional job breakpoints PG:181
 - conditional thread breakpoints PG:187
 - example PG:179, PG:182
 - unconditional job breakpoints PG:179
 - unconditional thread breakpoints PG:180
 - using statement numbers PG:185
- setting debug options** PG:171
- SEU**
 - See source entry utility (SEU)
- SFILE keyword** R:264
- sharing an open data path for a file** PG:277
- SHTDN (shut down) operation code** R:435, R:635
- shut down (SHTDN) operation code** R:635
- simple edit codes (X, Y, Z)** R:202
- skipping**
 - after R:332
 - before R:331
 - for printer output R:331
- SLN keyword** R:264
- sort sequence**
 - affect of SRTSEQ parameter PG:279
 - ALTSEQ table in compiler listing PG:431
 - debug considerations PG:184
- SORTA (sort an array) operation code** R:421, R:636
- source debugging**
 - adding an object to a session PG:172

source debugging *(continued)*

- built-in functions
 - %ADDR PG:206
 - %INDEX PG:206
 - %SUBSTR PG:206
 - %VARS PG:206
- changing values using %SUBSTR PG:208
- examples PG:207
- general discussion PG:206
- changing field values PG:208
- changing modules while debugging PG:175
- coordinating with listing options PG:69
- creating a program for debugging PG:59
- differences between ILE RPG and OPM RPG/400 PG:374
- displaying attributes of PG:210
- displaying data addressed by pointers PG:206
- displaying data and expressions PG:199
- displaying fields as hexadecimal values PG:205
- displaying fields in character format PG:205
- displaying indicators PG:204
- displaying multiple-occurrence data structures PG:203
- displaying the contents of a table PG:202
- displaying the contents of an array PG:202
- general discussion PG:163
- National Language Support PG:211
- NLSS considerations PG:184
- obtaining a formatted dump PG:251
- OPM program limit in debug session PG:172
- optimization effects PG:87, PG:164
- overview PG:20
- preparing a program PG:166
- removing an object from a session PG:172, PG:173
- rules for assigning values using EVAL PG:208
- setting and removing breakpoints PG:178
- setting debug options PG:171
- setting watch conditions PG:188
- starting the source debugger PG:169
- stepping through PG:194
- unexpected results PG:201
- updating production files PG:171
- viewing shorthand names PG:211
- viewing source PG:174
- source entry utility (SEU) PG:51**
 - browsing a compiler listing PG:68
 - entering source PG:52
- source from a data file, converting PG:390**
- source listing with indentation bars R:527**
- source member types, conversion of PG:380**
- source physical file, creating PG:51**
- source program**
 - converting all members PG:388
 - converting auto report source members PG:389
 - converting some members PG:388
 - converting to ILE RPG PG:382

source program *(continued)*

- entering into system PG:51
- entering SQL statements PG:55
- file and member names when converting PG:381
- record length of when converting PG:380
- source member types when converting PG:380
- source section of compiler listing PG:426**
- spacing**
 - for printer output R:331
 - not with WRITE operation R:664
- special command keys PG:334**
- SPECIAL file**
 - deleting records from PG:328
 - general discussion PG:327, PG:329
 - parameter list R:260
 - program device name R:260
 - valid file operations PG:328
- special routines, calling PG:158**
- special words R:7**
- specification templates, inserting PG:390**
- specifications**
 - common entries to all R:220
 - continuation rules R:221
 - description of PG:3
 - externally described file PG:265
 - file description PG:265
 - order PG:3, R:217
 - record format PG:282
 - types PG:3, R:217
- specifying a return point PG:238**
- specifying an activation group PG:110**
- specifying error indicators PG:227**
- specifying the format of compiler listing PG:64**
- split control field R:42**
- spooling PG:278**
- SQL**
 - See DB2 for AS/400 SQL
- SQL statements R:315**
- SQRT (square root) operation code R:418, R:638**
- SR (subroutine identifier) R:316, R:317**
- SRCFILE parameter**
 - CRTBNDRPG command PG:58, PG:408
 - CRTRPGMOD command PG:74
- SRCMBR parameter**
 - CRTBNDRPG command PG:58, PG:409
 - CRTRPGMOD command PG:74
- SRTSEQ keyword R:240**
- SRTSEQ parameter**
 - affect on key comparisons PG:279
 - CRTBNDRPG command PG:58, PG:413
 - CRTRPGMOD command PG:74
 - debug considerations PG:184
 - specifying on control specifications R:240
- stack, call PG:131, PG:218**
- standalone fields R:114, R:270**

Start Debug (STRDBG) command PG:169

Update Production files (UPDPROD)
parameter PG:171

starting commitment control PG:307

starting location of key field R:259

starting the ILE source debugger PG:169

statement view

creating PG:169
using for debug PG:185

static binding

See binding

static calls PG:19, PG:130

using CALLP R:466

STATIC keyword R:113, R:294

static procedure call PG:130

static storage R:113, R:294

status (of an edit word) R:213

status codes

data management errors PG:376
in file information data structure (INFDS) R:77
in program status data structure R:85

STEP debug command

definition PG:165
into PG:196
over PG:195

stepping while debugging

into a program or procedure PG:196
over a program or procedure PG:195
through a program PG:194

storage management

allocating during run-time PG:120
dynamic storage PG:113
managing run-time PG:113

strategies for creating ILE programs PG:23

STRDBG command

See Start Debug (STRDBG) command

string

indexing R:620
null-terminated R:286, R:389
removing blanks R:393
scanning R:383, R:620

string built-in functions

%REPLACE (Replace Character String) R:381
%SCAN (Scan for Characters) R:383
%STR (Get or Store Null-Terminated String) R:389
%SUBST (Get Substring) R:391
%TRIM (Trim Blanks at Edges) R:393
%TRIML (Trim Leading Blanks) R:394
%TRIMR (Trim Trailing Blanks) R:395

string operations

CAT (concatenate two character strings) R:444, R:471
CHECK (check) R:444, R:477
CHECKR (check reverse) R:444, R:480
general information R:444
SCAN (scan string) R:444, R:620

string operations (continued)

SUBST (substring) R:444, R:643
XLATE (translate) R:444, R:667

STRSEU (edit source) command PG:52

structured operations

indenting PG:65

structured programming operations

ANDxx (and) R:445, R:457
CASxx (conditionally invoke subroutine) R:469
DO (do) R:445, R:498
DOU (do until) R:445, R:500
DOUxx (do until) R:445, R:501
DOW (do while) R:445, R:503
DOWxx (do while) R:445, R:504
ELSE (else do) R:445, R:510
ENDyy (end a group) R:511
ENDyy (end a group) R:445
EVAL (evaluate) R:445, R:513
general information R:445
IF (if/then) R:445
IF (If) R:526
IFxx (if/then) R:445, R:527
ITER (iterate) R:445, R:531
LEAVE (leave a structured group) R:445, R:536
ORxx (or) R:445, R:584
OTHER (otherwise select) R:445, R:585
SELECT (begin a select group) R:445, R:623
WHEN (when true then select) R:445
When (When) R:660
whenxx (when true then select) R:661
WHxx (when true then select) R:445

Structured Query Language (SQL)

See DB2 for AS/400 SQL

SUB (subtract) operation code R:418, R:639

SUBDUR (subtract duration) operation code

calculating durations R:431
general discussion R:430
possible error situations R:642
subtracting dates R:431, R:640, R:641
unexpected results R:432

subfields

defining R:270
external definition R:280
for file information data structure PG:253, PG:255
for program status data structure PG:252, R:82
for PRTCTL PG:323
name prefixing R:123, R:261, R:293
overlying storage R:291
renaming R:123, R:279

subfiles

control-record format PG:335
descriptions PG:335
examples PG:337
file operation codes allowed with PG:336
general description PG:335, PG:336
record format PG:335, R:264

subfiles *(continued)*

uses of **PG:337**

subprocedures

calculations coding **R:92**
 coding considerations **PG:46**
 comparison with subroutines **R:95**
 debugging **PG:198**
 definition **R:89**
 example **PG:9**
 exception/error processing sequence **R:93**
 information in compiler listing **PG:434**
 local data in dump listing **PG:258**
 logic flow **PG:5**
 NOMAIN module **R:95**
 normal processing sequence **R:92**
 number of passed parameters **R:379**
 overview **PG:33**
 procedure interface **R:90, R:139**
 procedure specification **R:341**
 RETURN (return to caller) **R:616**
 return values **R:91**
 returning from **PG:156, R:616**
 scope of files **PG:80**
 scope of parameters **R:91, R:112**
 specifications for **R:217, R:219**
 stepping into **PG:196**
 stepping over **PG:195**

SUBR23R3 (message retrieval) PG:158

SUBR40R3 (manipulating Double Byte Characters variables) PG:158

SUBR41R3 (manipulating Double Byte Characters variables) PG:158

subroutine identifier (SR) R:317

subroutine names R:5

subroutine operations

BEGSR (beginning of subroutine) **R:448, R:458**
 CASxx (conditionally invoke subroutine) **R:448, R:469**
 ENDSR (end of subroutine) **R:448, R:512**
 EXSR (invoke subroutine) **R:448, R:519**
 general information **R:448**

subroutines

avoiding a loop **PG:236**
 calculation specifications entry in positions 7 and 8 **R:316, R:317**
 calling SUBR routines **PG:158**
 comparison with subprocedures **R:95**
 description **R:448**
 error **PG:228**
 example **R:448**
 file error (INFSR) **PG:229**
 file exception/error (INFSR) **R:80**
 maximum allowed per program **R:448**
 operation codes **R:448**
 program error (*PSSR) **PG:233**
 program exception/error (*PSSR) **R:87**

subroutines *(continued)*

program initialization (*INZSR) **R:26**
 use within a subprocedure **R:89**

SUBST (substring) operation code R:444, R:643

substring of character or graphic literal

ILE debug built-in %SUBSTR **PG:207**
 RPG built-in %SUBST **R:391**
 SUBST operation **R:643**

subtracting date-time durations R:431, R:640

subtracting factors R:639

summary tables

calculation specifications **R:315**
 edit codes **R:204, R:205**
 entry summary by type **R:295**
 file operation codes allowed with
 DISK **PG:304**
 PRINTER **PG:316**
 sequential **PG:326**
 SPECIAL **PG:328**
 WORKSTN **PG:341**
 function key indicators and corresponding function keys **R:52**
 ILE RPG built-in functions **R:349**
 ILE RPG restrictions **R:673**
 indicators **R:61, R:62**
 input specifications **R:300**
 keyword summary by definition type **R:296**
 operation codes **R:413**
 program description record identification entries **R:300**
 sequential file processing **PG:326**
 SPECIAL file processing **PG:328**

summing array elements R:666

symbolic name

array names **R:4**
 conditional compile names **R:4**
 data structure names **R:4**
 EXCEPT names **R:4**
 field names **R:4**
 file names **R:4**
 KLIST names **R:4**
 labels **R:5**
 PLIST names **R:5**
 prototype names **R:5**
 record names **R:5**
 subfield names **R:4**
 subroutine names **R:5**
 table names **R:5**

symbolic names R:3

syntax diagrams

CRTBNDRPG command **PG:406**
 CRTRPGMOD command **PG:419**
 CVTRPGSRC command **PG:383**
 interpreting **PG:405**

system functions

spooling **PG:278**

system reply list

- adding to PG:108
- changing PG:109

T

table

- See also array
- defining R:155
- definition R:141
- differences from array R:141
- displaying while debugging PG:202
- element, specifying R:155
- example of using R:155
- file R:246
- loading R:155
- name, rules for R:5
- number of elements R:278, R:363
- size of R:384
- specifying a table element R:155
- to file name R:262

table of parameters

- CRTBNDRPG command PG:58
- CRTRPGMOD command PG:74
- CVTRPGSRC command PG:383

TAG operation code R:422, R:432, R:646

tape file PG:290

TBREAK debug command

- definition PG:165
- using PG:180, PG:187

templates, inserting specification PG:390

TEST (test date/time/timestamp) operation code R:430, R:450, R:647

test library, using PG:171

test operations

- general information R:450
- TEST (test date/time/timestamp) operation code R:450, R:647
- TESTB (test bit) operation code R:450, R:649
- TESTN (test numeric) operation code R:450, R:651
- TESTZ (test zone) operation code R:450, R:653

TESTB (test bit) operation code R:421, R:450, R:649

testing breakpoints PG:178

TESTN (test numeric) operation code R:450, R:651

TESTZ (test zone) operation code R:450, R:653

TEXT keyword R:241

TEXT parameter

- CRTBNDRPG command PG:58, PG:409
- CRTRPGMOD command PG:74
- specifying on control specifications R:241

TGTRLS parameter

- CRTBNDRPG command PG:58, PG:416
- CRTRPGMOD command PG:74

THREAD debug command

- definition PG:165
- using PG:180

threaded applications

- debugging PG:178
- overview PG:21

TIME (time of day) operation code R:435, R:654

time data field

- general discussion R:177
- moving R:439
- TIMFMT R:241, R:264, R:294
- unexpected results R:432

time data format

- *JOB RUN time separator R:178
- control specification R:241
- description R:177
- external format on definition specification R:294
- file description specification R:264
- initialization R:178
- input specification R:306
- internal format on definition specification R:272
- output specification R:335
- separators R:178
- table of R:177

time out R:595

timestamp data field

- general discussion R:179
- unexpected results R:432

timestamp data format

- description R:179
- initialization R:179
- internal format on definition specification R:272
- output specification R:335
- separators R:179

TIMFMT keyword

- control specification R:241
- definition specification R:294
- file description specification R:264

tips

- See programming tips

TOFILE keyword R:294

TOFILE parameter PG:385, PG:388

TOMBR parameter PG:386, PG:388

total (T) output records R:327

TOTC PG:238

- flowchart R:22
- program exception/errors R:80

TOTL PG:238

- file exception/error subroutine (INFSR) R:80
- flowchart R:22
- program exception/errors R:83

trailing blanks, removing R:393, R:395

translate (XLATE) operation code R:667

translation table and alternate collating sequence coding sheet R:163

trial conversion, performing PG:388

TRUNCNBR keyword R:241

TRUNCNBR parameter

- CRTBNDRPG command PG:58, PG:416

TRUNCNBR parameter *(continued)*
 CRTRPGMOD command PG:74
 overflow in expressions R:399
 specifying on control specifications R:241
two-step process of program creation PG:73
type of record, output specification R:327
types of exception handlers PG:217

U

U1-U8

See external (U1-U8) indicators

UPDATE R:8

UDAY R:8

UEP

See user entry procedure (UEP)

UMONTH R:8

unary operations

- R:402

+ R:402

data types supported R:402

NOT R:402

precedence of operators R:399

unblocking/blocking records PG:287

unconditional breakpoint

definition PG:178

setting PG:179

setting and removing for job PG:179

setting and removing for thread PG:180

using statement numbers PG:185

unhandled escape message PG:224

unhandled exceptions PG:223

unhandled function check PG:225

UNLOCK (unlock a data area) operation

code R:429, R:433, R:656

Unregister ILE Condition Handler (CEEHDLU)

API PG:239

unsigned arithmetic R:420

unsigned integer format

alignment R:170

arithmetic operations R:420

considerations for using R:171

converting to R:396

definition R:170

definition specification R:272

output specification R:335

TRUNCNBR parameter PG:416

unsigned arithmetic R:420

unwanted control breaks R:38

UPDATE (modify existing record) operation

code R:433, R:658

update file R:245

Update Program (UPDPGM) command

using PG:87

updating a service program PG:100

updating data area R:586

UPDPGM command

See Update Program (UPDPGM) command

usage of indicators

See indicators

user date special words

format R:8

rules R:8

user entry procedure (UEP)

and the call stack PG:131

role in program PG:81

user-controlled file open R:257, R:265

user-created command, running an RPG program PG:108

user-defined edit codes (5-9) R:204

user-defined function

See subprocedures

USROPN keyword R:265

USRPRF keyword R:241

USRPRF parameter on CRTBNDRPG

CRTBNDRPG command PG:58, PG:415

specifying on control specifications R:241

UYEAR R:8

V

valid character set R:3

valid file operations

SPECIAL file PG:328

valid keys

for file PG:285

for records PG:285

VALUE keyword R:295

variable

based R:276, R:353

resetting R:609

scope R:91, R:112

variable-length fields R:195

variable-length format

character

description R:160

example R:161

rules R:161

definition specification R:272

graphic

description R:180

example R:181

rules R:181

input specification R:306

output specification R:337

VARYING keyword R:295

variable-length records PG:326

VARYING keyword R:295

view, debug

changing while debugging PG:176

COPY source PG:168

view, debug *(continued)*

default PG:169
definition PG:166
listing PG:168
root source PG:167
statement PG:169

viewing source while debugging PG:174

W

WAITRCD R:595

WATCH debug command

definition PG:165
example PG:192
setting conditions PG:188

WHEN (when true then select) operation

code R:397, R:427, R:445

When (When) operation code R:660

WHENxx (when true then select) operation

code R:427, R:661

WHxx (when true then select) operation

code R:445

Work with Reply List Entry (WRKRPLYE) command

changing a system reply list PG:109

WORKSTN file

definition PG:331
device name R:254
examples PG:345
externally described
 processing PG:335
externally-described PG:331
file operation codes allowed with PG:341
function key indicators with PG:334
multiple-device PG:342
processing PG:341
program-described
 calculation specifications PG:340
 combined file PG:341
 considerations PG:340
 general PG:338
 input file PG:341
 input specifications PG:340
 output file PG:341
 output specifications PG:339
 with format name PG:339
 without format name PG:340
sample data maintenance program PG:349
sample inquiry and search program PG:364
sample inquiry program PG:346
sample subfile processing program PG:358
subfiles
 control-record format PG:335
 examples PG:337
 for display-device file PG:335
 record format PG:335
 uses of PG:337

WORKSTN file *(continued)*

using PG:331

WRITE (create new records) operation code R:433, R:664

write/then read format (EXFMT) operation code R:517

writing a new record to a file R:664

writing records during calculation time R:515

WRKRPLYE command

See Work with Reply List Entry (WRKRPLYE) command

X

XFOOT (summing the elements of an array) operation code R:418, R:421, R:666

XLATE (translate) operation code R:444, R:667

Y

Y edit code R:233

Z

Z-ADD (zero and add) operation code R:418, R:669

Z-SUB (zero and subtract) operation code R:418, R:670

zero (blanking) fields R:334, R:340

zero suppression R:202

in body of edit word R:211

with combination edit code R:202

zoned decimal format

definition specification R:272

description R:170

Communicating Your Comments to IBM

AS/400 Advanced Series
ILE RPG for AS/400
Programmer's Guide
Version 4

Publication No. SC09-2507-01

If there is something you like—or dislike—about this book, please let us know. You can use one of the methods listed below to send your comments to IBM. If you want a reply, include your name, address, and telephone number. If you are communicating electronically, include the book title, publication number, page number, or topic you are commenting on.

The comments you send should only pertain to the information in this book and its presentation. To request additional publications or to ask questions or make comments about the functions of IBM products or systems, you should talk to your IBM representative or to your IBM authorized remarketer.

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

If you are mailing a readers' comment form (RCF) from a country other than the United States, you can give it to the local IBM branch office or IBM representative for postage-paid mailing.

- If you prefer to send comments by mail, use the RCF at the back of this book.
- If you prefer to send comments by FAX, use this number:
 - United States and Canada: 416-448-6161
 - Other countries: (+1)-416-448-6161
- If you prefer to send comments electronically, use the network ID listed below. Be sure to include your entire network address if you wish a reply.
 - Internet: torrcf@ca.ibm.com
 - IBMLink: [toribm\(torrcf\)](mailto:toribm(torrcf)@ca.ibm.com)
 - IBM/PROFS: [torolab4\(torrcf\)](mailto:torolab4(torrcf)@ca.ibm.com)
 - IBMMAIL: [ibmmail\(caibmwt9\)](mailto:ibmmail(caibmwt9)@ca.ibm.com)

Readers' Comments — We'd Like to Hear from You

AS/400 Advanced Series
ILE RPG for AS/400
Programmer's Guide
Version 4

Publication No. SC09-2507-01

Overall, how satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Overall satisfaction	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

How satisfied are you that the information in this book is:

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Accurate	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Complete	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Easy to find	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Easy to understand	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Well organized	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Applicable to your tasks	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Please tell us how we can improve this book:

Thank you for your responses. May we contact you? Yes No

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

Name

Address

Company or Organization

Phone No.



Fold and Tape

Please do not staple

Fold and Tape

PLACE
POSTAGE
STAMP
HERE

IBM Canada Ltd. Laboratory
Information Development
2G/345/1150/TOR
1150 EGLINTON AVENUE EAST
NORTH YORK ONTARIO CANADA M3C 1H7

Fold and Tape

Please do not staple

Fold and Tape



Part Number: 99H3778
Program Number: 5769-RG1



Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

99H3778



SC09-2507-01

