

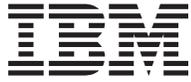


Rational Development Studio for i
ILE COBOL Programmer's Guide

7.1

SC09-2540-07





Rational Development Studio for i
ILE COBOL Programmer's Guide

7.1

SC09-2540-07

Note!

Before using this information and the product it supports, be sure to read the general information under "Notices" on page 671.

- | This edition applies to Version 7, Release 1, Modification Level 0, of IBM Rational Development Studio for i
- | (5770-WDS) and to all subsequent releases and modifications until otherwise indicated in new editions. This edition
- | applies only to reduced instruction set computer (RISC) systems.
- | This edition replaces SC09-2540-06.

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

IBM welcomes your comments. You can send your comments to:

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

You can also send your comments by facsimile (attention: RCF Coordinator), or you can send your comments electronically to IBM. See "How to Send Your Comments" for a description of the methods.

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 IBM Corporation 1993, 2010.

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

About This Guide xi

Who Should Use This Guide	xi
Prerequisite and Related Information	xii
How to Send Your Comments	xii
What's New in This Release?	xiii
# What's New in V6R1?	xiv
# What's New in V5R4?	xv
What's New in V5R3?	xv
What's New in V5R2?	xvii
What's New in V5R1?	xviii
What's New in V4R4?	xix
What's New in V4R2?	xx
What's New in V3R7?	xxi
What's New in V3R6/V3R2?	xxii
What's New in V3R1?	xxiii
Industry Standards	xxvi
An Acknowledgment	xxvii
ILE COBOL Syntax Notation	xxviii
Reading Syntax Diagrams	xxviii
Identifying Documentary Syntax	xxx
Interpreting Control Language (CL) Entry Codes	xxx

Using the application development tools in the client product. xxxi

Getting started in the Remote System Explorer perspective	xxxi
Remote Systems view	xxxvi
# System i Table view	xxxvii
Remote Systems LPEX Editor	xlili

Part 1. Compiling, Running, and Debugging ILE COBOL Programs . . . 1

Chapter 1. Introduction 3

Integrated Language Environment	3
Major Steps in Creating a Runnable ILE COBOL Program Object	3
Designing Your ILE COBOL Source Program	4
Entering Source Statements into a Source Member	6
Compiling a Source Program into Module Objects	7
Creating a Program Object	7
Running a Program Object	7
Debugging a Program	7
Other Application Development Tools	8
IBM Rational Development Studio for System i	8
WebSphere Development Studio for System i	9

Chapter 2. Entering Source Statements into a Source Member 11

Creating a Library and Source Physical File	11
Entering Source Statements Using the Source Entry Utility	12
COBOL Source File Format	12

Starting SEU	13
Using the COBOL Syntax Checker in SEU	13
Example of Entering Source Statements into a Source Member	15
Using Coded Character Set Identifiers	16
Assigning a CCSID to a Source Physical File	17
Including Copy Members with Different CCSIDs in Your Source File	17
Setting the CCSID for the COBOL Syntax Checker in SEU	18
Assigning a CCSID to a Locale	19
Runtime CCSID Considerations	19
Handling Different CCSIDs with the ILE Source Debugger	20

Chapter 3. Compiling Source Programs into Module Objects 21

Definition of a Module Object	21
Using the Create COBOL Module (CRTCBMOD) Command	24
Using Prompt Displays with the CRTCBMOD Command	24
Syntax for the CRTCBMOD Command	25
Parameters of the CRTCBMOD Command	28
Example of Compiling a Source Program into a Module Object	47
Specifying a Different Target Release	48
Specifying National Language Sort Sequence in CRTCBMOD	49
Collecting Profiling Data	49
Specifying Date, Time, and Timestamp Data Types	50
Using the PROCESS Statement to Specify Compiler Options	51
PROCESS Statement Options	59
Compiling Multiple Source Programs	60
Using COPY within the PROCESS Statement	61
Understanding Compiler Output	61
Specifying the Format of Your Listing	62
Browsing Your Compiler Listing Using SEU	63
A Sample Program and Listing	63

Chapter 4. Creating a Program Object 77

Definition of a Program Object	77
The Binding Process	77
Using the Create Program (CRTPGM) Command	79
Example of Binding Multiple Modules to Create a Program Object	81
Using the Create Bound COBOL (CRTBNDCBL) Command	81
Using Prompt Displays with the CRTBNDCBL Command	82
Syntax for the CRTBNDCBL Command	82
Parameters of the CRTBNDCBL Command	85
Invoking CRTPGM Implicitly from CRTBNDCBL	88

Example of Binding One Module Object to Create a Program Object	90
Specifying National Language Sort Sequence in CRTBNDCBL	90
Reading a Binder Listing	90
A Sample Binder Listing	91
Modifying a Module Object and Binding the Program Object Again	97
Changing the ILE COBOL Source Program	97
Changing the Optimization Levels	98
Removing Module Observability	101
Enabling Performance Collection	102
Collection Levels	102
Procedures	102

Chapter 5. Creating a Service Program 105

Definition of a Service Program	105
Using Service Programs	105
Writing the Binder Language Commands for an ILE COBOL Service Program	106
Using the Create Service Program (CRTSRVPGM) Command	106
Example of Creating a Service Program	107
Using the Retrieve Binder Source (RTVBNDSRC) Command as Input	108
Calling Exported ILE Procedures in Service Programs	108
Sharing Data with Service Programs	108
Canceling an ILE COBOL Program in a Service Program	109

Chapter 6. Running an ILE COBOL Program. 111

Running a COBOL Program Using the CL CALL Command	111
Passing Parameters to an ILE COBOL Program Through the CL CALL Command	111
Running an ILE COBOL Program Using a HLL CALL Statement	112
Running an ILE COBOL Program From a Menu-Driven Application	113
Running an ILE COBOL Program Using a User Created Command	114
Ending an ILE COBOL Program	114
Replying to Run Time Inquiry Messages	115

Chapter 7. Debugging a Program . . . 117

The ILE Source Debugger	118
Debug Commands	118
Preparing a Program Object for a Debug Session	120
Using a Listing View	120
Using a Source View	121
Using a Statement View	121
Starting the ILE Source Debugger	122
STRDBG Example	123
Setting Debug Options	124
Running a Program Object in a Debug Session	125
Adding Program Objects and Service Programs to a Debug Session	125

Removing Program Objects or Service Programs from a Debug Session	126
Viewing the Program Source	127
Changing the Module Object that is Shown	128
Changing the View of the Module Object that is Shown	128
Setting and Removing Breakpoints	129
Setting and Removing Unconditional Job Breakpoints	130
Setting and Removing Unconditional Thread Breakpoints	131
Setting and Removing Conditional Job Breakpoints	132
Setting and Removing Conditional Thread Breakpoints	134
Removing All Breakpoints	134
Setting and Removing Watch Conditions	134
Characteristics of Watches	135
Setting Watch Conditions	136
Displaying Active Watches	138
Removing Watch Conditions	138
Example of Setting a Watch Condition	139
Running a Program Object or ILE Procedure After a Breakpoint.	140
Resuming a Program Object or ILE Procedure	140
Stepping Through the Program Object or ILE Procedure	140
Displaying Variables, Constant-names, Expressions, Records, Group Items, and Arrays	143
Displaying Variables and Expressions	143
Displaying Records, Group Items, and Arrays	146
Changing the Value of Variables	148
Equating a Name with a Variable, Expression, or Command	149
National Language Support for the ILE Source Debugger.	150
Changing and Displaying Locale-Based Variables	150
Support for User-Defined Data Types	151

Part 2. ILE COBOL Programming Considerations 153

Chapter 8. Working with Data Items 155

General ILE COBOL View of Numbers (PICTURE Clause)	155
Defining Numeric Items	155
Separate Sign Position (For Portability)	156
Extra Positions for Displayable Symbols (Numeric Editing)	156
Computational Data Representation (USAGE Clause)	156
External Decimal (USAGE DISPLAY) Items	157
Internal Decimal (USAGE PACKED-DECIMAL or COMP-3)	157
Binary (USAGE BINARY or COMP-4) Items	157
Native Binary (USAGE COMP-5) Items	158
Internal Floating-Point (USAGE COMP-1 and COMP-2) Items	158
External Floating-Point (USAGE DISPLAY) Items	159

Creating User-Defined Data Types	159
Data Format Conversions	165
What Conversion Means	165
Conversion Takes Time	165
Conversions and Precision	165
Sign Representation and Processing	166
With the *CHGPOSSN Compiler Option	167
Checking for Incompatible Data (Numeric Class Test)	167
How to Do a Numeric Class Test	167
Performing Arithmetic	168
COMPUTE and Other Arithmetic Statements	168
Arithmetic Expressions	169
Numeric Intrinsic Functions	169
Converting Data Items (Intrinsic Functions)	173
Evaluating Data Items (Intrinsic Functions)	178
Formatting Dates and Times Based On Locales (LOCALE-DATE, LOCALE-TIME)	183
Fixed-Point versus Floating-Point Arithmetic	183
Floating-Point Evaluations	184
Fixed-Point Evaluations	184
Arithmetic Comparisons (Relation Conditions)	184
Examples of Fixed-Point and Floating-Point Evaluations	185
Processing Table Items	185
Processing Multiple Table Items (ALL Subscript)	186
What is the Year 2000 Problem?	186
Long-Term Solution	187
Short-Term Solution	187
Working with Date-Time Data Types	189
MOVE Considerations for Date-Time Data Items	192
Working With Locales	196
# Creating Locales on the i5/OS	197
Setting a Current Locale for Your Application	197
Identification and Scope of Locales	198
LC_MONETARY Locale Category	198
LC_TIME Category	202
LC_TOD Category	206
Manipulating null-terminated strings	208
Example: null-terminated strings	208
Chapter 9. Calling and Sharing Data Between ILE COBOL Programs.	211
Run Time Concepts	211
Activation and Activation Groups	211
COBOL Run Unit	212
Control Boundaries	212
Main Programs and Subprograms	213
Initialization of Storage	214
Transferring Control to Another Program	214
Calling an ILE COBOL Program	214
Identifying the Linkage Type of Called Programs and Procedures	215
Calling Nested Programs	217
Using Static Procedure Calls and Dynamic Program Calls	220
Using CALL identifier	223
Using CALL procedure-pointer	224
Using Recursive Calls	225
Returning from an ILE COBOL Program	225
Returning from a Main Program	226

Returning from a Subprogram	226
Maintaining OPM COBOL/400 Run Unit Defined STOP RUN Semantics	227
Examples of Returning from an ILE COBOL Program	227
Passing Return Code Information (RETURN-CODE Special Register)	232
Passing and Sharing Data Between Programs	232
Comparing Local and Global Data	233
Passing Data Using CALL...BY REFERENCE, BY VALUE, or BY CONTENT	233
Sharing EXTERNAL Data	237
Sharing EXTERNAL Files	238
Passing Data Using Pointers	244
Passing Data Using Data Areas	244
Effect of EXIT PROGRAM, STOP RUN, GOBACK, and CANCEL on Internal Files	246
Canceling an ILE COBOL Program	247
Canceling from Another ILE COBOL Program	247
Canceling from Another Language	248

Chapter 10. COBOL and the eBusiness World 249

COBOL and XML	249
COBOL and MQSeries	249
COBOL and Java Programs	250
System Requirements	250
COBOL and PCML	250
COBOL and JNI	255
COBOL and Java Data Types	268
JNI Copy Members for COBOL	269

Chapter 11. Processing XML Input 277

XML parser in COBOL	277
Accessing XML documents	279
Parsing XML documents	279
Processing XML events	280
Writing procedures to process XML	285
Understanding XML document encoding	293
Specifying the code page	294
Parsing documents in other code pages	294
Handling errors in XML documents	294
Unhandled exceptions	296
Handling exceptions	296
Terminating the parse	297
CCSID conflict exception	297

Chapter 12. Producing XML output 301

Generating XML output	301
Example: generating XML	303
Enhancing XML output	306
Example: enhancing XML output	307
Example: converting hyphens in element names to underscores	310
Controlling the encoding of generated XML output	311
Handling errors in generating XML output	311

Chapter 13. Calling and Sharing Data with Other Languages 313

Calling ILE C and VisualAge C++ Programs and Procedures	313
Passing Data to an ILE C Program or Procedure	315
Sharing External Data with an ILE C Program or Procedure	317
Returning Control from an ILE C Program or Procedure	317
Examples of an ILE C Procedure Call from an ILE COBOL Program	318
Sample Code for ILE C Procedure Call Example 1	318
Sample Code for ILE C Procedure Call Example 2	320
Creating and Running the ILE C Procedure Call Examples.	322
Example of an ILE C Program Call from an ILE COBOL Program	322
Sample Code for ILE C Program Call Example	322
Creating and Running the ILE C Program Call Example	323
Calling ILE RPG Programs and Procedures	323
Passing Data to an ILE RPG Program or Procedure	324
Returning Control from an ILE RPG Program or Procedure	326
Calling ILE CL Programs and Procedures	327
Passing Data to an ILE CL Program or Procedure	327
Returning Control from an ILE CL Program or Procedure	329
Calling OPM Languages.	329
Calling OPM COBOL/400 Programs	330
Calling EPM Languages	331
Issuing a CL Command from an ILE COBOL Program	332
Including Structured Query Language (SQL) Statements in Your ILE COBOL Program	332
Calling an ILE API to Retrieve Current Century	333
Using Intrinsic Functions or the ACCEPT Statement to Retrieve Current Century	333
Calling IFS API.	334

Chapter 14. Using Pointers in an ILE COBOL Program 335

Defining Pointers	335
Pointer Alignment.	336
Writing the File Section and Working-Storage Section for Pointer Alignment	337
Redefining Pointers	338
Initializing Pointers Using the NULL Figurative Constant	338
Reading and Writing Pointers	339
Using the LENGTH OF Special Register with Pointers	339
Setting the Address of Linkage Section Items	340
Using ADDRESS OF and the ADDRESS OF Special Register.	340
Using Pointers in a MOVE Statement	340

Using Pointers in a CALL Statement.	342
Adjusting the Value of Pointers	342
Accessing User Spaces Using Pointers and APIs	343
Processing a Chained List Using Pointers	355
Passing Pointers between Programs and Procedures	356
Check for the End of the Chained List	357
Processing the Next Record.	357
Incrementing Addresses Received from Another Program	358
Passing Entry Point Addresses with Procedure-Pointers	358

Chapter 15. Preparing ILE COBOL Programs for Multithreading 361

How Language Elements Are Interpreted in a Multithreaded Environment	362
Working with Run-Unit Scoped Elements	363
Working with Program Invocation Instance Scoped Elements	364
Choosing THREAD for Multithreading Support	364
Language Restrictions under THREAD.	364
Control Transfer within a Multithreaded Environment	365
Limitations on ILE COBOL in a Multithreaded Environment	365
Example of Using ILE COBOL in a Multithreaded Environment	365
Sample Code for the Multithreading Example	365
Creating and Running the Multithreading Example	368

Chapter 16. ILE COBOL Error and Exception Handling. 369

ILE Condition Handling.	369
Ending an ILE COBOL Program	371
Using Error Handling Bindable Application Programming Interfaces (APIs)	371
Initiating Deliberate Dumps	372
Program Status Structure	373
Handling Errors in String Operations	374
Handling Errors in Arithmetic Operations.	374
The ON SIZE ERROR Phrase	374
Handling Errors in Floating-Point Computations	375
Handling Errors in Input-Output Operations	376
Processing of Input-Output Verbs	377
Detecting End-of-File Conditions (AT END Phrase)	378
Detecting Invalid Key Conditions (INVALID KEY Phrase).	379
Using EXCEPTION/ERROR Declarative Procedures (USE Statement)	380
Determining the Type of Error Through the File Status Key	381
MAP 0010: How File Status is Set	383
Handling Errors in Sort/Merge Operations	385
Handling Exceptions on the CALL Statement.	385
User-Written Error Handling Routines	386
Common Exceptions and Some of Their Causes	386
Recovery After a Failure.	387

Recovery of Files with Commitment Control	387
TRANSACTION File Recovery	388
Handling Errors in Operations Using Null-Capable Fields	393
Handling Errors in Locale Operations	393

Part 3. ILE COBOL Input-Output Considerations 395

Chapter 17. Defining Files 397

Types of File Descriptions	397
Defining Program-Described Files	397
Defining Externally Described Files	398
Describing Files Using Data Description Specifications (DDS)	398

Chapter 18. Processing Files. 409

Associating Files with Input-Output Devices	409
Specifying Input and Output Spooling	411
Input Spooling	411
Output Spooling	411
Overriding File Attributes	412
Redirecting File Input and Output	413
Locking and Releasing Files	413
Locking and Releasing Records	414
Sharing an Open Data Path to Access a File	415
Unblocking Input Records and Blocking Output Records	415
Using File Status and Feedback Areas	416
FILE STATUS	416
OPEN-FEEDBACK Area	416
I-O-FEEDBACK Area	417
Using Commitment Control	417
Commitment Control Scoping	421
Example of Using Commitment Control	422
Sorting and Merging Files	428
Describing the Files	428
Sorting Files	430
Merging Files	430
Specifying the Sort Criteria	431
Writing the Input Procedure	432
Writing the Output Procedure	433
Restrictions on the Input Procedures and Output Procedures	433
Determining Whether the Sort or Merge Was Successful	434
Premature Ending of a Sort or Merge Operation	434
Sorting Variable Length Records	435
Example of Sorting and Merging Files	435
Declaring Data Items Using SAA Data Types	438
Variable-length Fields	438
Date, Time, and Timestamp Fields	440
Null-Capable Fields	443
DBCS-Graphic Fields	451
Variable-length DBCS-graphic Fields	452
Floating-point Fields	454

Chapter 19. Accessing Externally Attached Devices. 457

Types of Device Files	457
Accessing Printer Devices	457
Naming Printer Files	458
Describing Printer Files	458
Writing to Printer Files	460
Example of Using FORMATFILE Files in an ILE COBOL Program	461
Accessing Files Stored on Tape Devices	465
Naming Files Stored on Tape Devices	465
Describing Files Stored on Tape Devices	466
Reading and Writing Files Stored on Tape Devices	467
Accessing Files Stored on Diskette Devices	469
Naming Files Stored on Diskette Devices	469
Describing Files Stored on Diskette Devices	470
Reading and Writing Files Stored on Diskette Devices	470
Accessing Display Devices and ICF Files	471

Chapter 20. Using DISK and DATABASE Files 473

Differences between DISK and DATABASE Files	473
# File Organization and i5/OS File Access Paths	473
File Processing Methods for DISK and DATABASE Files	474
Processing Sequential Files	474
Processing Relative Files	475
Processing Indexed Files	477
Processing Files with Descending Key Sequences	487
Processing Files with Variable Length Records	487
Examples of Processing DISK and DATABASE Files	489
Sequential File Creation	489
Sequential File Updating and Extension	491
Relative File Creation	493
Relative File Updating	495
Relative File Retrieval	497
Indexed File Creation	500
Indexed File Updating	502
IBM i System Files	506
Distributed Data Management (DDM) Files	507
Using DDM Files with Non-IBM i Systems	507
DDM Programming Considerations	508
DDM Direct (Relative) File Support	509
Distributed Files	509
Open Considerations for Data Processing	510
When Distributed Data Processing is Overridden	510
When Distributed Data Processing is NOT Overridden	511
Input/Output Considerations for Distributed Files	511
SQL Statement Additions for Distributed Data Files	513
Examples of Processing Distributed Files	514
Processing Files with Constraints	515
Restrictions	516
Adding, Modifying and Removing Constraints	516

Checking that Constraints Have Been Successfully Added or Removed	517
Order of Operations	517
Handling Null Fields with Check Constraints	517
Handling Constraint Violations	517
Database Features that Support Referential or Check Constraints	518
Chapter 21. Using Transaction Files	521
Defining Transaction Files Using Data Description Specifications	521
Processing an Externally Described Transaction File	523
Writing Programs That Use Transaction Files	524
Naming a Transaction File	524
Describing a Transaction File	525
Processing a Transaction File	526
Example of a Basic Inquiry Program Using Transaction Files	529
Using Indicators with Transaction Files	536
Passing Indicators in a Separate Indicator Area	536
Passing Indicators in the Record Area	537
Examples of Using Indicators in ILE COBOL Programs	537
Using Subfile Transaction Files	549
Defining a Subfile Using Data Description Specifications	549
Using Subfiles for a Display File	550
Accessing Single Device Files and Multiple Device Files	554
Writing Programs That Use Subfile Transaction Files	563
Naming a Subfile Transaction File	564
Describing a Subfile Transaction File	565
Processing a Subfile Transaction File	565
Example of Using WRITE SUBFILE in an Order Inquiry Program	569
Example of Using READ SUBFILE...NEXT MODIFIED and REWRITE SUBFILE in a Payment Update Program	583

Part 4. Appendixes 603

Appendix A. Level of Language Support	605
COBOL Standard	605
ILE COBOL Level of Language Support	605
System Application Architecture (SAA) Common Programming Interface (CPI) Support	607

Appendix B. The Federal Information Processing Standard (FIPS) Flagger . 609

Appendix C. ILE COBOL Messages	611
COBOL Message Descriptions	611
Severity Levels	611
Compilation Messages	612
Program Listings	613
Interactive Messages	613
Responding to Messages	614

Appendix D. Supporting International Languages with Double-Byte Character Sets 617

Using DBCS Characters in Literals	617
How to Specify Literals Containing DBCS Characters	618
How the COBOL Compiler Checks DBCS Characters	619
How to Continue Mixed Literals on a New Line	619
Syntax-Checker Considerations	619
Where You Can Use DBCS Characters in a COBOL Program	620
How to Write Comments	620
Identification Division	620
Environment Division	621
Configuration Section	621
Input-Output Section	621
File Control Paragraph	621
Data Division	621
File Section	621
Working-Storage Section	621
Procedure Division	623
Intrinsic Functions	623
Conditional Expressions	623
Input/Output Statements	623
Data Manipulation Statements	625
Procedure Branching Statements	628
Table Handling—SEARCH Statement	628
SORT/MERGE	628
Compiler-Directing Statements	628
COPY Statement	628
REPLACE Statement	629
TITLE Statement	629
Communications between Programs	629
FIPS Flagger	629
COBOL Program Listings	630
Intrinsic Functions with Collating Sequence Sensitivity	630

Appendix E. Example of a COBOL Formatted Dump 631

Appendix F. XML reference material	635
XML exceptions that allow continuation	635
XML exceptions that do not allow continuation	639
XML conformance	643
XML generate exceptions	645

Appendix G. Migration and Compatibility Considerations between OPM COBOL/400 and ILE COBOL . . 647

Migration Strategy	647
Compatibility Considerations	648
General Considerations	648
CL Commands	648
Compiler-Directing Statements	651
Environment Division	652
Data Division	652
Procedure Division	653

Application Programming Interfaces (APIs)	662	Notices	671
Run Time.	662	Programming Interface Information	672
Appendix H. Glossary of		Trademarks	672
Abbreviations	665	Acknowledgments.	673
Appendix I. ILE COBOL		Bibliography.	675
Documentation.	669	Index	679
Online Information	669		
Hardcopy Information	669		

About This Guide

This guide describes how to write, compile, bind, run, debug, and maintain Integrated Language Environment (ILE) COBOL compiler programs on the IBM i. It provides programming information on how to call other ILE COBOL and non-ILE COBOL programs, share data with other programs, use pointers, and handle exceptions. It also describes how to perform input/output operations on externally attached devices, database files, display files, and ICF files.

Using this book, you will be able to:

- Design and code ILE COBOL programs
- Enter, compile, and bind ILE COBOL programs
- Run and debug ILE COBOL programs
- Study coded ILE COBOL examples.

Note: You should be familiar with Chapters 1 through 6 of this guide before proceeding to the other chapters.

This book refers to other IBM® publications. These publications are listed in the “Bibliography” on page 675 with their full title and base order number. When they are referred to in text, a shortened version of the title is used.

Who Should Use This Guide

This guide is intended for application programmers who have some experience
with the COBOL programming language and for the operators who run the
programs. It is a guide to programming in the ILE COBOL language for users of
i5/OS.

Before you use this guide you should have a basic understanding of the following:

- Data processing concepts
- The COBOL programming language
- The IBM IBM i (formerly OS/400) operating system
- Integrated Language Environment (ILE) concepts
- Application Programming Interfaces (APIs)
- Development tools, such as *Application Development ToolSet (ADTS)* for Non-Programmable Terminal (NPT) base.

Note: Use WebSphere Development Studio Client for System i. This is the
recommended method and documentation about the workstation tools
appears in that product's online help.

- How to use the controls and indicators on your display and how to use the keys on your keyboard, such as:
 - Cursor movement keys
 - Function keys
 - Field exit keys
 - Insert and Delete keys
 - Error Reset key.
- How to operate your display station when it is linked to the IBM System i and running i5/OS software. This means knowing how to use the IBM i operating system and its Control Language (CL) to do such things as:

#

- # – Sign on and sign off the display station
 - # – Interact with displays
 - # – Use Help
 - # – Enter CL commands
 - # – Use Application Development Tools
 - # – Respond to messages
 - # – Perform file management.
 - The basic concepts of IBM i CL functions.
 - How to use data management support to allow an application to work with files.
 - How to use the following *Application Development ToolSet* tools:
 - The Screen Design Aid (SDA) used to design and code displays or the DDS design utility that is a part of the client product
 - The Source Entry Utility (SEU) used to enter and update source members or the language sensitive editor that is a part of the client product
- # **Note:** Use WebSphere Development Studio Client for System i. This is the recommended method and documentation about the workstation tools appears in that product's online help.
- # See Using the application development tools in the client product for information about getting started with the client tools.
- The Structured Query Language (SQL) used to insert SQL statements into ILE COBOL programs.

Prerequisite and Related Information

- # Use the Information Center as your starting point for looking up IBM i, System i, and AS/400e technical information. You can access the Information Center in two ways:
- # • From the following Web site:
 - # <http://www.ibm.com/systems/i/infocenter/>
 - # • From CD-ROMs that ship with your IBM i order:
 - # *i5/OS Information Center CD, SK3T-4091.*
- # The i5/OS Information Center contains advisors and important topics such as CL commands, system application programming interfaces (APIs), logical partitions, clustering, Java™, TCP/IP, Web serving, and secured networks. It also includes links to related IBM Redbooks and Internet links to other IBM Web sites such as the Technical Studio and the IBM home page.
- # The manuals that are most relevant to the ILE COBOL compiler are listed in the “Bibliography” on page 675.

How to Send Your Comments

- # Your feedback is important in helping to provide the most accurate and high-quality information. IBM welcomes any comments about this book or any other i5/OS documentation.
- # • If you prefer to send comments by fax, use the following number:
 - # 1-845-491-7727
 - # • If you prefer to send comments by mail, use the the following address:

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

If you are mailing a readers' comment form from a country other than the
United States, you can give the form to the local IBM branch office or IBM
representative for postage-paid mailing.

- # • If you prefer to send comments electronically, use one of these e-mail addresses:
 - # – Comments on books:
 - | RCHCLERK@us.ibm.com
 - # – Comments on the i5/OS Information Center:
 - # RCHINFOC@us.ibm.com

Be sure to include the following:

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

| What's New in This Release?

| The following list describes the enhancements made to ILE COBOL in V7R1:

- | • COMPUTATIONAL-5 (native binary) data type
 - | COMPUTATIONAL-5 or COMP-5 is a native binary data type now supported
 - | by the USAGE clause. COMP-5 data items are represented in storage as binary
 - | data, and can contain values up to the capacity of the native binary
 - | representation (2, 4, or 8 bytes). When numeric data is moved or stored into a
 - | COMP-5 item, truncation occurs at the binary field size rather than at the
 - | COBOL picture size limit. When a COMP-5 item is referenced, the full binary
 - | field size is used in the operation. This support will enhance portability to or
 - | from COBOL on other IBM platforms and operating systems.
- | • Ability to specify a non-numeric literal on the VALUE clause for a national data
| item.
- | • XML GENERATE performance improvements and PROCESS options
 - | Performance improvements have been made for XML GENERATE when the
 - | APPEND option is specified. Users who have a large number of data records to
 - | be appended into a data structure or into a stream file will benefit from these
 - | changes. The improvements include the addition of new PROCESS statement
 - | parameter XMLGEN with option values:
 - | – NOKEEPFILEOPEN / KEEPFILEOPEN
 - | Specify KEEPFILEOPEN to indicate that the XML stream file is to be left open
 - | and not closed when the XML GENERATE statement is complete, so that
 - | subsequent XML GENERATE FILE-STREAM APPEND statements can quickly
 - | append data to the stream file.
 - | – NOASSUMEVALIDCHARS / ASSUMEVALIDCHARS
 - | Specify ASSUMEVALIDCHARS to have XML GENERATE bypass the
 - | checking for special characters (less than "<", greater than ">", ampersand "&",
 - | and the single and double quote symbols), and for characters not supported
 - | by XML that would require being generated as hexadecimal. Otherwise
 - | normal checking will be done with the default NOASSUMEVALIDCHARS.
- | • Ability to encrypt the listing debug view

A new CRTBNDCBL / CRTCBMOD parameter is added to support the encryption of the listing debug view. DBGENCKEY specifies the encryption key to be used to encrypt program source that is embedded in debug views.

- Larger program support

The CRTBNDCBL / CRTCBMOD OPTIMIZE parameter now supports a new *NEVER option value. The *NEVER value allows larger programs to compile by not generating optimization code for the program. PROCESS statement option NEVEROPTIMIZE is also added.

- Support for the teraspace storage model

The storage model for a program/module can now be specified using the new CRTBNDCBL / CRTCBMOD parameter STGMDL with option values:

- *SNGLVL specifies that the program/module is to be created with single-level storage model
- *TERASPACE specifies that the program/module is to be created with teraspace storage model
- *INHERIT specifies that the program/module is to inherit the storage model of its caller

Additionally, the activation group parameter ACTGRP on the CRTBNDCBL command now has a new default option value:

- *STGMDL: When STGMDL(*TERASPACE) is specified, the program will be activated into the QILETS activation group. For all other storage models, the program will be activated into the QILE activation group when it is called.

- New PROCESS statement options

- ACTGRP is now available as a PROCESS statement parameter with option values:

- STGMDL
- NEW
- CALLER

- NEVEROPTIMIZE is now available as a PROCESS statement option

- STGMDL is now available as a PROCESS statement parameter with option values:

- INHERIT
- SNGLVL
- TERASPACE

- XMLGEN is now available as a PROCESS statement parameter with option values:

- NOKEEPFILEOPEN / KEEPFILEOPEN
- NOASSUMEVALIDCHARS / ASSUMEVALIDCHARS

This V7R1 guide, *IBM Rational Development Studio for i: ILE COBOL Programmer's Guide*, SC09-2540-07, differs in many places from the V6R1 guide, SC09-2540-06. Most of the changes are related to the enhancements; others reflect minor technical corrections.

What's New in V6R1?

The following list describes the enhancements made to ILE COBOL in V6R1:

• National UCS-2 CCSID enhancement

The NTLCCSID parameter has been added to the CRTCBMOD and

CRTBNDCBL commands and to the PROCESS statement, to allow you to specify

the UCS-2 CCSID to be used for National data items. With this parameter you
can specify a CCSID other than the default 13488, such as CCSID 1200, to be
used for National items.

- # • PCML in module support

The PGMINFO parameter on the CRTCBMOD and CRTBNDCBL commands
has been enhanced to allow you to specify the location where you want to put
the generated PCML. When the user specifies *PCML as the first parameter for
the PGMINFO keyword, a second parameter specifying a location of *STMF,
*MODULE, or *ALL can also be specified.

*STMF will cause the PCML to be put into the streamfile specified on the
INFOSTMF parameter.

*MODULE will cause the PCML to be put into the generated module.

*ALL will cause the PCML to be put in all of these locations.

- # • Complex OCCURS DEPENDING ON (ODO) debugger support

Support has been added so the system debugger and the client debugger can
now debug complex OCCURS DEPENDING ON arrays.

- # • Large Program Support

The compiler has been enhanced so that larger programs and programs
containing a very large number of data items can now be compiled (subject to
system limitations).

What's New in V5R4?

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

- XML support has been enhanced. A new statement, XML GENERATE , converts the content of COBOL data records to XML format. XML GENERATE creates XML documents encoded in Unicode UCS-2 or in one of several single-byte EBCDIC or ASCII CCSIDs. See Chapter 12, "Producing XML output," on page 301.
- Null-terminated nonnumeric literal
Nonnumeric literals can be null-terminated. They can be used anywhere a nonnumeric literal can be specified except that null-terminated literals are not supported in "ALL literal" figurative constants.
- New CRTBNDCBL / CRTCBMOD option
*NOCOMPRESSDBG/*COMPRESSDBG specifies whether listing view compression should be performed by the compiler when DBGVIEW option *LIST or *ALL is specified.
- New intrinsic functions:
 - DISPLAY-OF
 - NATIONAL-OF
 - TRIM
 - TRIML
 - TRIMR

What's New in V5R3?

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

- Large VALUE clause support

When the *NOSTDTRUNC compiler option is in effect, data items described with usage BINARY, or COMP-4 that do not have a picture symbol P in their PICTURE clause can have a value up to the capacity of the native binary representation.

- CONSTANT data type
A CONSTANT data type is defined by specifying a level-01 entry containing the CONSTANT clause for a literal. The CONSTANT data item can then be used in place of the literal.
- XML support
XML PARSE statement provides the interface to a high-speed XML parser that is part of the COBOL run time. The XML PARSE statement parses an XML document into its individual pieces and passes each piece, one at a time, to a user-written processing procedure.
These XML special registers are used to communicate information between the XML parser and the user-written processing procedure:
 - XML-CODE
 - XML-EVENT
 - XML-NTEXT
 - XML-TEXT
- Alternate Record Key support
The ALTERNATE RECORD KEY clause lets you define alternate record keys associated with indexed files. These alternate keys allow you to access the file using a different logical ordering of the file records.
- DBCS data item names (DBCS word support)
- 63 digit support
 - The maximum length of packed decimal, zoned decimal, and numeric-edited items has been extended from 31 to 63 digits.
 - The ARITHMETIC parameter on the CRTCBMOD and CRTBNDCBL commands and on the PROCESS statement has a new EXTEND63 option.
- 7 new ANSI Intrinsic functions:
 - INTEGER
 - REM
 - ANNUITY
 - INTEGER-PART
 - MOD
 - FACTORIAL
 - RANDOM
- New CRTBNDCBL / CRTCBMOD options:
 - *NOCRTARKIDX / *CRTARKIDX Specifies whether or not to create temporary alternate record key indexes if permanent ones can not be found.
 - *STDINZHEX00 Specifies that data items without a value clause are initialized with hexadecimal zero.
 - *EXTEND63 option for the ARITHMETIC parameter increases the precision of intermediate results for fixed-point arithmetic up to 63 digits.
- New PROCESS statement options:
 - PROCESS statement option NOCOMPRESSDBG/COMPRESSDBG indicates whether listing view compression should be performed by the compiler when DBGVIEW option *LIST or *ALL is specified

- NOCRTARKIDX/CRTARKIDX
- STDINZHEX00
- EXTEND63 option for the ARITHMETIC parameter
- Program Status Structure

The program status structure is a predefined structure that contains error information when the COBOL program receives an error. The PROGRAM STATUS clause is use to specify the error information that is received.

What's New in V5R2?

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

- Recursive program support

An optional RECURSIVE clause has been added to provide support for recursive programs. These are COBOL programs that can be recursively re-entered.
- Local Storage Section support

A new data section that defines storage allocated and freed on a per-invocation basis has been added. You can specify the Local-Storage Section in both recursive and non-recursive programs.
- Java interoperability

Two new features have been added to enhance Java interoperability. These include:

 - UTF8String intrinsic function

This function provides the ability to convert strings to UTF-8 format.
 - PCML support

New parameters have been added to the CRTCBMOD and CRTBNDCBL commands to give users the ability to tell the compiler to generate PCML source for their COBOL program. When the user specifies PGMINFO(*PCML) and the name of a streamfile on the INFOSTMF parameter, the compiler will generate PCML into the specifed streamfile. The generated PCML makes it easier for Java programs to call this COBOL program, with less Java code.
- Additional intrinsic functions

Several new intrinsic functions have been added to this release. These include:

 - Max
 - Median
 - Midrange
 - Min
 - ORD-Max
 - ORD-Min
 - Present Value
 - Range
 - Standard Deviation
 - Sum
 - Variance
- IFS

ILE Cobol source stored in IFS stream files can be compiled. The SRCSTMF and INCDIR parameters have been added to the CRTCBMOD and CRTBNDCBL commands to give users the ability to tell the compiler to compile from source stored in IFS stream files.

What's New in V5R1?

The following list describes the enhancements made to ILE COBOL in V5R1:

- UCS-2 (Unicode) support
 - National data, a new type of data item, has been added to provide support for the coded character set specified in ISO/IEC 10646-1 as UCS-2. The code set is the basic set defined in the Unicode standard.
 - UCS-2 character set
 - This coded character set provides a unique code for each character appearing in the principal scripts in use around the world. Each character is represented by a 16-bit (2-byte) code.
 - National data
 - This new type of data item specifies that the item contains data coded using the UCS-2 code set. An elementary data item whose description contains a USAGE NATIONAL clause, or an elementary data item subordinate to a group item whose description contains a USAGE NATIONAL clause, is a national data item.
 - NTLPADCHAR compiler option and PROCESS statement option
 - This option allows you to specify three values: the SBCS padding character, DBCS padding character, and national padding character. The appropriate padding character is used when a value is moved into a national datatype item and does not fill the national datatype item completely.
 - ALL national literal
 - Allows the word ALL wherever a national hexadecimal literal is allowed, so that for example you could move all UCS-2 blanks into a national data item.
 - PROCESS statement option NATIONAL
 - When this option is specified, elementary data items defined using the picture symbol N will have an implied USAGE NATIONAL clause. A USAGE DISPLAY-1 clause will be implied for these items if the compiler option is not used.
 - National hexadecimal literals
 - Literals containing national data values may be specified using the syntax:
NX"hexadecimal-character-sequence..."
 - Figurative constants
 - The figurative constant SPACE/SPACES represents one or more UCS-2 single byte space characters (U+0020) when used with national data items.
- JAVA interoperability support
 - QCBLESRC.JNI file
 - This file provides the same definitions and prototypes that are provided in the JNI.h file, but written in COBOL rather than C.
 - Data mapping between Java and COBOL datatypes
- Mainframe portability support
 - NOCOMPASBIN/COMPASBIN PROCESS statement option indicates whether USAGE COMPUTATIONAL or COMP has the same meaning as USAGE COMP-3 or USAGE COMP-4.
 - NOLSPTRALIGN/LSPTRALIGN PROCESS statement option indicates whether data items with USAGE POINTER or PROCEDURE-POINTER are aligned at multiples of 16 bytes relative to the beginning of the record in the linkage section.

- NOADJFILLER/ADJFILLER PROCESS statement option indicates whether any implicit fillers inserted by the compiler to align a pointer data item should be inserted before or after a group that has a pointer data item as the first member of the group.
- Complex OCCURS DEPENDING ON (ODO) support
The following constitute complex ODO:
 - Entries subordinate to the subject of an OCCURS or an ODO clause can contain ODO clauses (table with variable length elements).
 - A data item described by an ODO can be followed by a non-subordinate data item described with ODO clause (variably located table).
 - Entries containing an ODO clause can be followed by non-subordinate items (variably located fields). These non-subordinate items, however, cannot be the object of an ODO clause.
 - The location of any subordinate or non-subordinate item, following an item containing an ODO clause, is affected by the value of the ODO object.
 - The INDEXED BY phrase can be specified for a table that has a subordinate item that contains an ODO clause.
- The LICOPT parameter has been added to the CRTCBMOD and CRTBNDCBL commands to allow advanced users to specify Licensed Internal Code options.
- The OPTVALUE PROCESS statement option indicates whether the generation of code to initialize data items containing a VALUE clause in the working-storage section should be optimized.

What's New in V4R4?

The following list describes the enhancements made to ILE COBOL in V4R4:

- Thread Safety Support
Support for calling ILE COBOL procedures from a threaded application, such as Domino[®] or Java. The THREAD parameter has been added to the PROCESS statement, to enable ILE COBOL modules for multithreaded environments. Access to the procedures in the module should be serialized.
- 31-digit support
 - The maximum length of packed decimal, zoned decimal, and numeric-edited items has been extended from 18 to 31 numeric digits.
 - The ARITHMETIC parameter has been added to the CRTCBMOD and CRTBNDCBL commands, and to the PROCESS statement to allow the arithmetic mode to be set for numeric data. This allows you to specify the computational behavior of numeric data.
- Euro currency support
 - The ability to specify more than one currency sign in a COBOL program to support the dual currency system that will be in effect for three years starting in January 1999 among the participating countries.
 - The ability to represent multi-character currency signs, so that the international currency signs (e.g. USD, FRF, DEM, EUR) as well as single-character currency signs (e.g. "\$") can be specified for COBOL numeric edited fields.
 - The OPTION parameter values *MONOPIC/*NOMONOPIC have been added to the CRTCBMOD and CRTBNDCBL commands, and MONOPIC/NOMONOPIC have been added to the PROCESS statement. This allows you to choose between a moncased or a case sensitive currency symbol in a PICTURE character-string.

What's New in V4R2?

The following list describes the enhancements made to ILE COBOL in V4R2:

- User-defined data types

A user-defined data type is defined by specifying a level-01 entry containing the TYPEDEF clause; all entries that are subordinate to the level-01 entry are considered part of the user-defined data type. A user-defined data type can be used to define new data items of level-01, -77, or -02 through -49, by specifying a TYPE clause for the new data item, that references the user-defined data type.

- Program profiling support

The PRFDTA parameter has been added to both the CRTCBMOD and CRTBNDCBL commands, and to the PROCESS statement, to allow a program to be profiled for optimization.

- Null-values support

Null-values support (by way of the NULL-MAP and NULL-KEY-MAP keywords) has been added to the following statements and clauses to allow the manipulation of null values in database records:

- ASSIGN clause
- COPY-DDS statement
- DELETE statement
- READ statement
- REWRITE statement
- START statement
- WRITE statement.

- Locale support

i5/OS locale objects (*LOCALE) specify certain cultural elements such as a date
format or time format. This cultural information can be associated with ILE
COBOL date, time, and numeric-edited items. The following new characters,
clauses, phrases and statements were added to support this:

- # - The LOCALE clause of the SPECIAL-NAMES paragraph
- # - Associates i5/OS locale object with a COBOL mnemonic-name
- # - The LOCALE phrase of a date, time, or numeric-edited item
- # - Allows you to specify a locale mnemonic-name, so that the data item is
associated with a i5/OS locale object
- # - Along with specific locales defined in the LOCALE clause of the
SPECIAL-NAMES paragraph, a current locale, and a default locale have been
defined. The current locale can be changed with the new SET LOCALE
statement (Format 8).
- # - A locale object is made up of locale categories, each locale category can be
changed with the SET LOCALE statement.
- # - Locale categories have names such as LC_TIME and LC_MONETARY. These
names include the underscore character. This character has been added to the
COBOL character set.
- # - The SUBSTITUTE phrase of the COPY DDS statement has been enhanced
to allow the underscore character to be brought in.

The following new intrinsic functions allow you to return culturally-specific dates and times as character strings:

- LOCALE-DATE
- LOCALE-TIME.

- Additions to Century support

The following enhancements have been made to the ILE COBOL Century support:

- A new class of data items, class date-time, has been added. Class date-time includes date, time, and timestamp categories. Date-time data items are declared with the new FORMAT clause of the Data Description Entry.
- Using COPY-DDS and the following values for the CVTOPT compiler parameter, i5/OS DDS data types date, time, and timestamp can be brought into COBOL programs as COBOL date, time, and timestamp items:
 - *DATE
 - *TIME
 - *TIMESTAMP.
- Using the CVTOPT parameter value *CVTTODATE, packed, zoned, and character i5/OS DDS data types with the DATFMT keyword can be brought into COBOL as date items.
- The following new intrinsic functions allow you to do arithmetic on items of class date-time, convert items to class date-time, test to make sure a date-time item is valid, and extract part of a date-time item:
 - ADD-DURATION
 - CONVERT-DATE-TIME
 - EXTRACT-DATE-TIME
 - FIND-DURATION
 - SUBTRACT-DURATION
 - TEST-DATE-TIME.

What's New in V3R7?

The following list describes the enhancements made to ILE COBOL in V3R7:

- Century support

The capability for users to work with a 4-digit year has been added in the following statements and functions:

- ACCEPT statement with the YYYYDDD and YYYYMMDD phrases
- The following intrinsic functions convert a 2-digit year to a 4-digit year:
 - DATE-TO-YYYYMMDD
 - DAY-TO-YYYYDDD
 - YEAR-TO-YYYY
- The following intrinsic functions return a 4-digit year:
 - CURRENT-DATE
 - DAY-OF-INTEGERS
 - DATE-OF-INTEGERS
 - WHEN-COMPILED

- Floating-point support

The *FLOAT value of the CVTOPT parameter on the CRTCBMOD and CRTBNDCBL commands allows floating-point data items to be used in ILE COBOL programs. Also, the affected statements (such as ACCEPT, DISPLAY, MOVE, COMPUTE, ADD, SUBTRACT, MULTIPLY, and DIVIDE) support floating-point.

- Data area support

New formats of the ACCEPT and DISPLAY statements have been added to provide the ability to retrieve and update the contents of i5/OS data areas.

- Intrinsic Functions

The following intrinsic functions have been added:

ACOS	LOG10
ASIN	LOWER-CASE
ATAN	MEAN
CHAR	NUMVAL
COS	NUMVAL-C
CURRENT-DATE	ORD
DATE-OF-INTEGERS	REVERSE
DAY-OF-INTEGERS	SIN
DATE-TO-YYYYMMDD	SQRT
DAY-TO-YYYYDDD	TAN
INTEGER-OF-DATE	UPPER-CASE
INTEGER-OF-DAY	WHEN-COMPILED
LENGTH	YEAR-TO-YYYY
LOG	

- Binding Directory parameter—BNDDIR
The BNDDIR parameter has been added to the CRTBNDCBL command to allow the specification of the list of binding directories that are used in symbol resolution.
- Activation Group parameter—ACTGRP
The ACTGRP parameter has been added to the CRTBNDCBL command to allow the specification of the activation group that a program is associated with when it is called.
- Library qualified program objects and data areas
The LIBRARY phrase has been added to the following ILE COBOL statements to allow OS/400[®] program objects and data areas to be qualified with an OS/400 library name:
 - CALL
 - CANCEL
 - SET
 - ACCEPT
 - DISPLAY
- Performance collection data
The ENBPFRCOL parameter has been added to the CRTCBMOD and CRTBNDCBL commands, and to the PROCESS statement to allow performance measurement code to be generated in a module or program. The data collected can be used by the system performance tool to profile an application's performance.
- New ILE debugger support
The ILE debugger now allows you to:
 - Debug most OPM programs
 - Set watch conditions, which are requests to set breakpoints when the value of a variable (or an expression that determines the address of a storage location) changes.

What's New in V3R6/V3R2?

The following list describes the enhancements made to ILE COBOL in V3R6 and V3R2:

- New EXIT PROGRAM phrase

The AND CONTINUE RUN UNIT phrase has been added to the EXIT PROGRAM statement to allow exiting of a calling program without stopping the run unit.

- New SET statement pointer format

A new format of the SET statement has been added that enables you to update pointer references.

- DBCS Data Support

You can now process Double Byte Character Set (DBCS) data in ILE COBOL. The ILE COBOL compiler supports DBCS, in which each logical character is represented by two bytes. DBCS provides support for ideographic languages, such as the IBM Japanese Graphic Character Set, Kanji.

- Support for CALL...BY VALUE and CALL...RETURNING

CALL...BY VALUE and CALL...RETURNING gives you the ability to pass arguments BY VALUE instead of BY REFERENCE and receive RETURN values. This allows for greater ease of migration, and improved interlanguage support as ILE C for i5/OS and ILE RPG for i5/OS both support CALL... BY VALUE and CALL...RETURNING.

- Support of the BY VALUE and RETURNING phrases of the PROCEDURE DIVISION Header

The BY VALUE phrase of the PROCEDURE DIVISION header allows COBOL to receive BY VALUE arguments from a calling COBOL program or other ILE language such as RPG, C, or C++. The RETURNING phrase of the PROCEDURE DIVISION header allows COBOL to return a VALUE to the calling ILE procedure.

What's New in V3R1?

The following list describes the enhancements made to ILE COBOL in V3R1:

- EXTERNAL data items

You can define data items that are available to every program in the ILE COBOL run unit by using the EXTERNAL clause. No longer do you need to pass all variables that are to be shared across programs as arguments on the CALL statement. This support encourages greater modularity of applications by allowing data to be shared without using arguments and parameters on the CALL statement.

- EXTERNAL files

You can define files that are available to every program in the run unit. You can seamlessly make I/O requests to the same file from any ILE COBOL program within the run unit that declares the file as EXTERNAL. For external files there is only one file cursor regardless of the number of programs that use the file. You can share files across programs, and thereby develop smaller, more maintainable programs. Using EXTERNAL files provides advantages over using shared open files since only one OPEN and CLOSE operation is needed for all participating programs to use the file. However, an EXTERNAL file cannot be shared among different activation groups nor with programs written in other programming languages.

- Nested Source Programs

An ILE COBOL source program can contain other ILE COBOL source programs. These contained programs may refer to some of the resources, such as data items and files, of the programs within which they are contained or define their own resources locally, which are only visible in the defining program. As the ILE COBOL programs are themselves resources, their scope is also controlled by the

nesting structure and the scope attribute attached to the program. This provides greater flexibility in controlling the set of ILE COBOL programs that can be called by an ILE COBOL program. Nested ILE COBOL programs provides a mechanism to hide resources that would otherwise be visible.

- INITIAL Clause
You have a mechanism whereby an ILE COBOL program and any programs contained within it are placed in their initial state every time they are called. This is accomplished by specifying INITIAL in the PROGRAM-ID paragraph. This provides additional flexibility in controlling the COBOL run unit.
- REPLACE statement
The REPLACE statement is useful to replace source program text during the compilation process. It operates on the entire file or until another REPLACE statement is encountered, unlike the COPY directive with the REPLACING phrase. The REPLACE statements are processed after all COPY statements have been processed. This provides greater flexibility in changing the ILE COBOL text to be compiled.
- DISPLAY WITH NO ADVANCING statement
By using the NO ADVANCING phrase on the DISPLAY statement, you have the capability to leave the cursor following the last character that is displayed. This allows you to string together items to be displayed on a single line from various points in the ILE COBOL program.
- ACCEPT FROM DAY-OF-WEEK statement
ILE COBOL now allows you to accept the day of the week (Monday = 1, Tuesday = 2 ...) and assign it to an identifier. This support complements the existing ACCEPT FROM DAY/DATE/TIME support.
- SELECT OPTIONAL clause for Relative Files
This allows for the automatic creation of relative files even when the file is opened I-O. This extends the support that is already available for sequential files.
- Support for Nested COPY statements
Copy members can contain COPY statements thereby extending the power of the COPY statement. If a COPY member contains a COPY directive, neither the containing COPY directive nor the contained COPY directive can specify the REPLACING phrase.
- Enhancements to Extended ACCEPT and DISPLAY statements
You can work with tables on the Extended ACCEPT statement. This allows you to easily and selectively update the elements of the table.
Variable length tables are also allowed on the Extended ACCEPT and DISPLAY statements.
Also, the SIZE clause is supported on the extended ACCEPT statement.
- Procedure-pointer support
Procedure-pointer is a new data type that can contain the address of an ILE COBOL program or a non-ILE COBOL program. Procedure-pointers are defined by specifying the USAGE IS PROCEDURE-POINTER clause on a data item. This new data type is useful in calling programs and or ILE procedures that are expecting this type of data item as its parameter. Procedure-pointer data items can also be used as the target of a CALL statement to call another program.
- New Special Registers
 - RETURN-CODE special register

Allows return information to be passed between ILE COBOL programs. Typically, this register is used to pass information about the success or failure of a called program.

- SORT-RETURN special register

Returns information about success of a SORT or MERGE statement. It also allows you to terminate processing of a SORT/MERGE from within an error declarative or an input-output procedure.

- New Compiler options

- *PICGGRAPHIC/*NOPICGGRAPHIC

*PICGGRAPHIC is a new parameter for the CVTOPT option which allows the user to bring DBCS data into their ILE COBOL program.

- *IMBEDERR/*NOIMBEDERR option

*IMBEDERR is a new compiler option which includes compile time errors at the point of occurrence in the compiler listing as well as at the end of the listing.

- *FLOAT/*NOFLOAT

*FLOAT is a new parameter for the CVTOPT option which allows you to bring floating-point data items into your ILE COBOL programs with their DDS names and a USAGE of COMP-1 (single-precision) or COMP-2 (double-precision).

- *NOSTDTRUNC/*STDTRUNC option

*NOSTDTRUNC is a new compiler option which suppresses the truncation of values in BINARY data items. This option is useful in migrating applications from IBM System/390®.

- *CHGPOSSGN/*NOCHGPOSSGN option

This option is useful when sharing data between the OS/400 and IBM S/390®. This option is provided for IBM System/390 compatibility. It changes the bit representation of signed packed and zoned data items when they are used in arithmetic statements or MOVE statements and the values in these data items are positive.

- Quoted system names support

Support has been added to allow literals where system-names are allowed. You can use whatever names the system supports and is no longer limited to valid COBOL names.

- There is no COBOL limit on the following functions as these are now determined by system constraints.

- Number of declared files.

- Number of parameters on the CALL statement and on the Procedure Division USING phrase. A system limit of 400 for ILE procedures and 255 for program objects does apply here.

- Number of SORT-MERGE input files and the number of SORT-MERGE keys. The maximum number of SORT-MERGE input files is 32 and the maximum length of the SORT-MERGE key is 2000 bytes.

- START with NO LOCK statement.

By using the NO LOCK phrase on the START statement, the file cursor will be positioned on the first record to be read without placing a lock on the record. This support is provided for indexed and relative files and complements the READ with NO LOCK function that is already available.

Note: START with NO LOCK is a new statement in both ILE COBOL and OPM COBOL/400.

- Static procedure call support
 You can develop your applications in smaller, better maintainable module objects, and link them together as one program object, without incurring the penalty of dynamic program call overhead. This facility, together with the common runtime environment provided by the system, also improves your ability to write mixed language applications. The ILE programming languages permits the binding of C, RPG, COBOL, and CL into a single program object regardless of the mix of source languages.
 New syntax on the CALL literal statement and a new compiler option have been added to ILE COBOL to differentiate between static procedure calls and dynamic program calls.
- Variable Length Record support (RECORD IS VARYING Clause)
 You can define and easily use different length records on the same file using standard ANSI COBOL syntax. Not only does this provide great savings in storage but it also eases the task of migrating complex applications from other systems.
- Expanded compiler limits
 ILE COBOL now offers expanded compiler limits:
 - size of group and elementary data items
 - size of fixed and variable length tables
 - number of nesting levels for conditional statements
 - number of operands in various Procedure Division statements

Industry Standards

Throughout this document, Standard COBOL refers to the COBOL programming language as defined in the document:

- American National Standard for Information Systems - Programming Language - COBOL, ANSI X3.23-1985, ISO 1989:1985 updated with the content of the following documents, in the order they are listed:
 - ANSI X3.23a-1989, American National Standard for Information Systems - Programming Language - Intrinsic Function Module for COBOL and ISO 1989:1985/Amd.1:1992, Programming Languages - COBOL, Amendment 1: Intrinsic function module
 - ANSI X3.23b-1993, American National Standard for Information Systems - Programming Language - Correction Amendment for COBOL and ISO/IEC 1989 DAM2 Programming Languages - COBOL, Amendment 2: Correction and clarification amendment for COBOL

The ILE COBOL compiler is designed to support Standard COBOL (as defined above) and

- FIPS Publication 21-4, Federal Information Processing Standard 21-4, COBOL at the intermediate subset level, as understood and interpreted by IBM as of January, 1995.

From this point on, the term Standard COBOL will be used to refer to the ANSI standard just described.

Portions of this manual are copied from Standard COBOL documents, and are reproduced with permission from these publications (copyright 1985 by the American National Standards Institute), copies of which you can purchase from the American National Standard Institute at 1430 Broadway, New York, New York, 10018.

The COBOL language is maintained by the ANSI Technical Committee X3J4.

Refer to Appendix A, “Level of Language Support,” on page 605 for more information on the industry standards supported by the ILE COBOL compiler.

An Acknowledgment

The following extract from U.S. Government Printing Office Form Number 1965-0795689 is presented for your information and guidance:

Any organization interested in reproducing the COBOL report and specifications in whole or in part, using ideas taken from this report as the basis for an instruction manual or for any other purpose is free to do so. However, all such organizations are requested to reproduce this section as part of the introduction to the document. Those using a short passage, as in a book review, are requested to mention COBOL in acknowledgment of the source, but need not quote this entire section.

COBOL is an industry language and is not the property of any company or group of companies, or of any organization or group of organizations.

No warranty, expressed or implied, is made by any contributor or by the COBOL Committee as to the accuracy and functioning of the programming system and language. Moreover, no responsibility is assumed by any contributor, or by the committee, in connection therewith.

Procedures have been established for the maintenance of COBOL. Inquiries concerning the procedures for proposing changes should be directed to the Executive Committee of the Conference on Data Systems Languages.

The authors and copyright holders of copyrighted material:

- Programming for the UNIVAC® I and II, Data Automation Systems copyrighted 1958, 1959, by Unisys Corporation;
- IBM Commercial Translator, Form No. F28-8013, copyrighted 1959 by IBM;
- FACT, DSI 27A5260-2760, copyrighted 1960 by Minneapolis-Honeywell

have specifically authorized the use of this material in whole or in part, in the COBOL specifications. Such authorization extends to the reproduction and use of COBOL specifications in programming manuals or similar publications.

ILE COBOL Syntax Notation

ILE COBOL basic formats are presented in a uniform system of syntax notation. This notation, designed to assist you in writing COBOL source statements, is explained in the following paragraphs:

- COBOL keywords and optional words appear in uppercase letters; for example:

`MOVE`

They must be spelled exactly as shown. If any keyword is missing, the compiler considers it to be an error.

- Variables representing user-supplied names or values appear in all lowercase letters; for example:

`parmx`

- For easier text reference, some words are followed by a hyphen and a digit or a letter, as in:

identifier-1

This suffix does not change the syntactical definition of the word.

- Arithmetic and logical operators (+, -, *, /, **, >, <, =, >=, and <=) that appear in syntax formats are required. For a complete listing of reserved ILE COBOL words, see the *IBM Rational Development Studio for i: ILE COBOL Reference*.
- All punctuation and other special characters appearing in the diagram are required by the syntax of the format when they are shown; if you leave them out, errors will occur in the program.
- You must write the required and optional clauses (when used) in the order shown in the diagram unless the associated rules explicitly state otherwise.

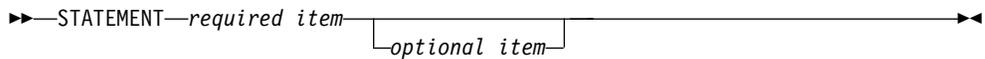
Reading Syntax Diagrams

Throughout this book, syntax is described using the structure defined below.

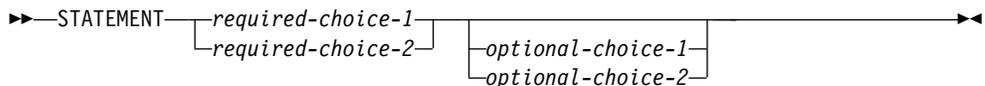
- Read the syntax diagrams from left to right, and from top to bottom, following the path of the line:
 - ▶▶— Indicates the beginning of a statement. Diagrams of syntactical units other than statements, such as clauses, phrases and paragraphs, also start with this symbol.
 - ▶ Indicates that the statement syntax is continued on the next line.
 - ▶— Indicates that a statement is continued from the previous line.
 - ▶◀ Indicates the end of a statement. Diagrams of syntactical units other than statements, such as clauses, phrases and paragraphs, also end with this symbol.

Note: Statements within a diagram of an entire paragraph do not start with ▶▶ and end with —▶◀ unless their beginning or ending coincides with that of the paragraph.

- Required items appear on the horizontal line (the main path). Optional items appear below the main path:



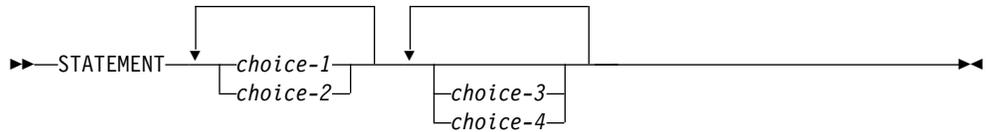
- When you can choose from two or more items, they appear vertically, in a stack. If you must choose one of the items, one item of the stack appears on the main path. If choosing an item is optional, the entire stack appears below the main path:



- An arrow returning to the left above an item indicates that the item can be repeated:

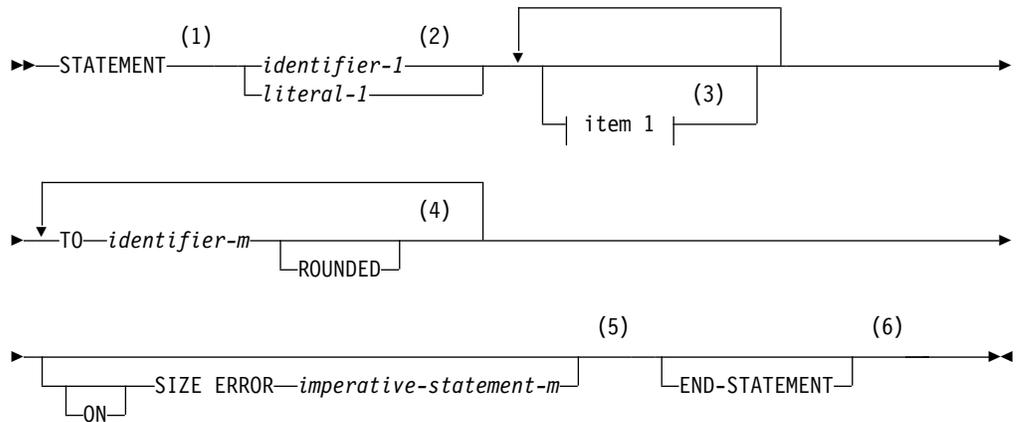


- A repeat arrow above a stack of required or optional choices indicates that you can make more than one choice from the stacked items, or repeat a single choice:

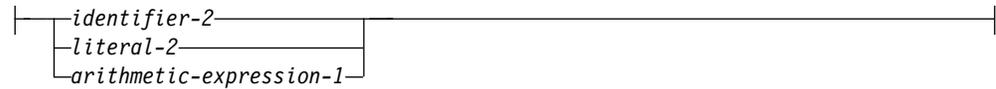


The following example shows how the syntax is used:

Format



item 1:



Notes:

- 1 The STATEMENT key word must be specified and coded as shown.
- 2 This operand is required. Either identifier-1 or literal-1 must be coded.
- 3 The item 1 fragment is optional; it can be coded or not, as required by the application. If item 1 is coded, it can be repeated with each entry separated by one or more COBOL separators. Entry selections allowed for this fragment are described at the bottom of the diagram.
- 4 The operand identifier-m and associated TO key word are required and can be repeated with one or more COBOL separators separating each entry. Each entry can be assigned the key word ROUNDED.
- 5 The ON SIZE ERROR phrase with associated imperative-statement-m are optional. If the ON SIZE ERROR phrase is coded, the key word ON is optional.
- 6 The END-STATEMENT key word can be coded to end the statement. It is not a required delimiter.

Identifying Documentary Syntax

COBOL clauses and statements illustrated within syntax diagrams that are syntax checked, but are treated as documentation by the ILE COBOL compiler, are identified by footnotes.

Interpreting Control Language (CL) Entry Codes

The code that appears in the upper right corner of each CL syntax diagram contains the entry codes that specify the environment in which the command can be entered. The codes indicate whether or not the command can be:

- Used in a batch or interactive job (outside a compiled program; Job:B or I)
- Used in a batch or interactive compiled program (Pgm:B or I)
- Used in a batch or interactive REXX procedure (REXX:B or I)
- Used as a parameter for the CALL CL command, or passed as a character string to the system program QCMDEXC (Exec).

Using the application development tools in the client product

You can accomplish the most commonly performed development tasks on your
System i[®] using the Remote Systems view, the System i Table view, and the Remote
Systems LPEX Editor. These views and their associated functions are available
through the Remote System Explorer perspective in the client product.

If you are accustomed to working with PDM, the Table view provides similar support. If you are accustomed to working with SEU, the Remote Systems LPEX Editor can operate in a similar way.

Note: References to *tutorials* throughout these topics refer to the following tutorials that are in the Tutorials Gallery in the client product:

- *Maintain an ILE COBOL application using Remote System Explorer*
- *Maintain an ILE RPG application using Remote System Explorer*

For more information, see the following topics:

Getting started in the Remote System Explorer perspective

The Remote System Explorer perspective enables you to access, edit, run,
compile, and debug all items on your system.

Remote Systems view

Use the Remote Systems view to navigate and list objects that you need to access to develop your applications.

System i Table view

The System i Table view displays the same information as the Remote Systems view, with the added ability to sort items, view descriptions, and perform more PDM-like actions.

Remote Systems LPEX Editor

The Remote Systems LPEX Editor is based on the base LPEX Editor, and contains System i specific functions.

Getting started in the Remote System Explorer perspective

The Remote System Explorer perspective enables you to access, edit, run, compile,
and debug all items on your system.

When you first open Remote System Explorer, you are not connected to any
system except your local workstation. To connect to a remote System i, you need to
define a profile and a connection.

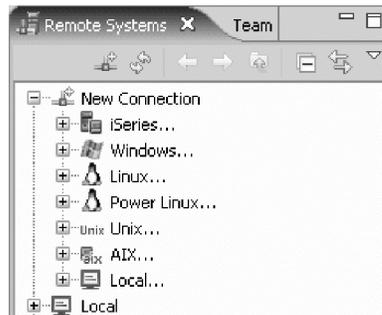
- A profile is used to group connections, share connections, or keep them private.
- A connection is a TCP/IP network connection to your System i, that enables you to access, edit, run, compile, and debug all items on the system. When you define a connection, you specify the name or IP address of the remote system and you also give the connection itself a unique name that acts as a label in your workspace so that you can easily connect and disconnect. When you connect to the System i, the workbench prompts you for your user ID and password on that system.

To start working in Remote System Explorer (RSE):

1. Start the workbench
2. When prompted, specify the workspace
3. Once the workbench opens, ensure that you are in the Remote System Explorer perspective. If the perspective is not open, you can open it by selecting **Window > Open Perspective > Remote System Explorer**
Click the X to close the Welcome view.



4. In the Remote Systems view **New Connection** shows the various remote system types you can connect to through the Remote Systems view.



5. Create a connection:

- #
- a. Expand **System i** under **New Connection** in the view, to open the Name personal profile page. Accept the default profile value to open the connection page.
 - b. Leave the **Parent profile** default value
 - c. Enter your host system name in the **Host name** field. The **Connection name** field is automatically filled with the host name.
 - d. Leave the **Verify host name** check box selected.
 - e. Click **Finish** to create your connection.

#

#

#

#

You can define multiple connections to the same System i, but in addition, you can include different configurations for the startup of your connection, such as saving different user IDs and passwords, initial library lists, for example. After you create a connection to a System i, you can easily connect and disconnect.

#

#

#

For more information see the topic *Configuring a connection to a remote system* in the online help in the client product. See also *Configuring a connection to a System i and connecting to an i5/OS* in the tutorials.

Tips:

- When creating a connection, use the default profile name. The default profile can be used to share connections with others, and to use filter pools. (For more information about filter pools, see the topic *Remote System Explorer filters, filter pools, and filter pool references* in the online help in the client product.)
- When specifying **Host name**, you can either specify the IP address, or the fully qualified name (if necessary).
- The **Verify host name** check box ensures that you can actually connect to the System i you have specified. To ensure that you have all necessary

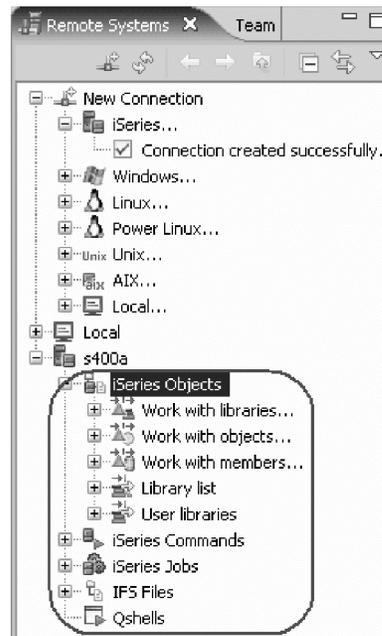
PTFs installed on your System i, right-click the connection (after it has been created) and select **Verify connection**. This ensures that all necessary ports are open, callbacks can be performed, and that the required host PTFs are applied.

- To define the startup properties for a connection, right-click on a connection and select **Properties**.

Subsystems

#

After you configure a connection to a System i, you can easily connect and expand your new connection to show the subsystems. Subsystems are represented by containers which show the libraries, command sets, and jobs on your remote system. Subsystem in this context is not related to the subsystem on the System i.



System i connections have five different subsystems:

1. System i Objects: This can be used to access libraries, objects and members.
2. System i Commands : By default, this subsystem is populated with a set of predefined commands that you can use to run against remote objects. You can also define command sets, and commands of your own. The results are logged in the Commands Log view. (For more information about the Commands Log view see the topic *Running programs and commands* in the online help in the client product.)
3. System i Jobs: Use this subsystem to list jobs. You can subset by job attributes, and perform job operations, such as hold, resume, end.
4. IFS Files: Explore files and folder structures in the Integrated File System, and perform actions on them.
5. Qshells: Access the list of active running Qshells for the connection, and use this subsystem to start a Qshell. (For more information see the topic *Running and viewing commands and shells using the Remote Shell view* in the online help in the client product.)

The view that a connection is in is called the Remote Systems view. It works much like Windows File Explorer. You drill down by clicking the “plus” (+) to gain access to desired items. For example, expand the *LIBL filter to see all the libraries in the library list, then expand a file to see all its members (much like option 12 in PDM).

Filters

Expanding a subsystem results in a list of filters for that subsystem. Filters are names of lists of items that you can specify, reuse, and share. Filters *filter out* items that you are not currently interested in. When creating filters, you can use generic values, and have as many filters as you want or need. Filters can be created for each subsystem, so you can have filters for IFS files, local files, i5/OS objects, for example.

Tips:

- You can always drill down within a filter if the item is a container (a library and a file are examples of containers)
- You can specify multiple levels of generic values, for example you could specify library BOB, file QRPG* member A*, when you create your filter.
- Pay close attention to the page where you specify the filter name. On this page you choose whether or not you want the filter to be only for the specified connection, or to appear in all connections. You can also specify a profile, if you want to share your filters with others.

Since filters are names which are stored with your connection in RSE, all filters persist between sessions.

Filter strings

When first created, a filter contains only one filter string. By modifying the properties of a filter, you can add additional filter strings. Filter strings provide the ability to generate more complex lists. By using multiple filter strings in a filter, you can list members in different files, and even in different libraries in a single named filter.

Tips:

- Filters must contain the same types of items. For example, it is not possible to list objects and members in the same filter.
- Group items into a filter by project or application. For example, add filter strings so that you can see the DDS members in the same list as your RPG and COBOL files.
- For more information about filters, see the topic *Filtering members* in the online help in the client product. See also *Introducing filters* in the tutorials.

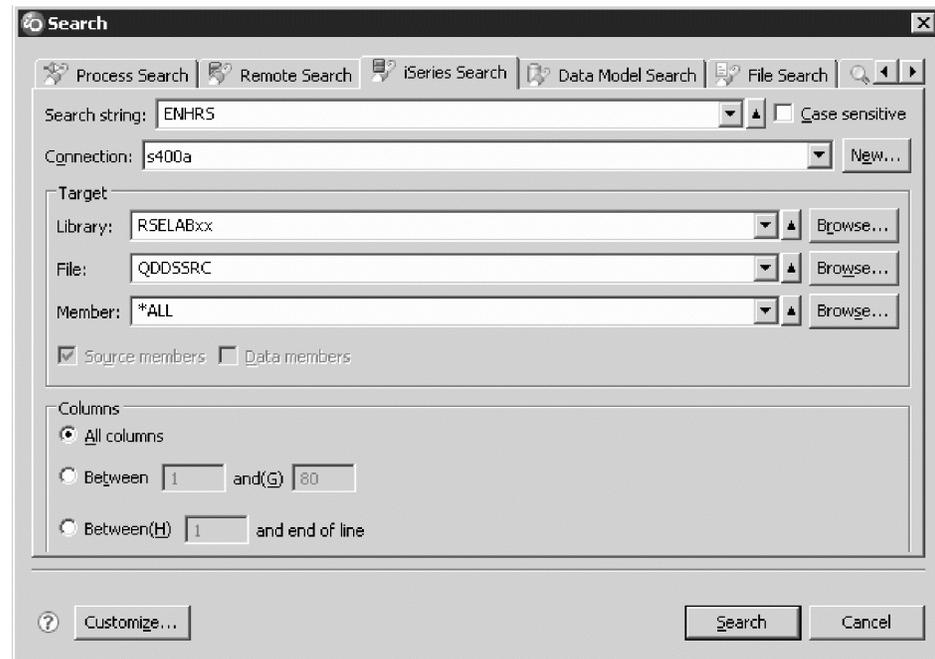
Searching

There are two ways to search in RSE:

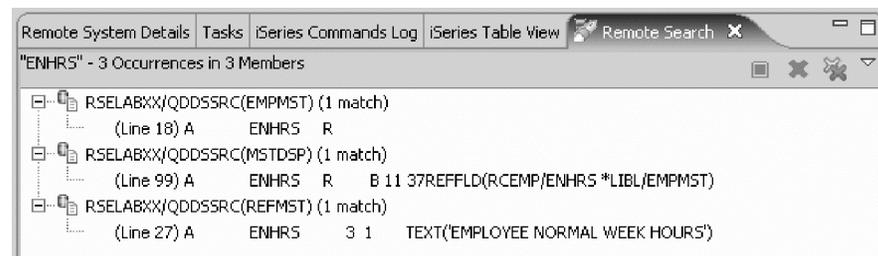
1. From the **Search** menu option (then selecting System i)
2. From the Find String action in the Remote Systems view and System i Table view

RSE allows you to search filters, not just libraries, files, and members. This means that you can search with very flexible search patterns. For example, you could

search all the members in the file QRPGLSRC in library MYLIB and the members A* in the files PRJA* in library PROJECT by invoking the Find string action on the filter that contained those filter strings.



Search results appear in the Remote Search view, and the view has a history of searches. You see the list of all the search results in one place, allowing you to open whichever member you want first, and using whichever match in the member you decide. The Remote Search view allows you to manage the resulting list, by letting you remove members and matches from the list through the pop-up menu.



Tips:

- Double-click the member name in the Remote Search view to open a member in the “Remote Systems LPEX Editor” on page xliii for editing and to be positioned to the match selected.
- The pop-up in the Remote Search view has a list of options similar to the System i Table view.
- Double-click on the Remote Search tab to maximize the view to the full workbench window. This will allow you to see more matches at one time.
- Expand or collapse matched members to quickly zero in on the matches that are important to you.

- See the topic *Searching for text strings on the System i* in the online help in the client product. See also *Searching multiple files* in the tutorials.

Comparing RSE to PDM

The following table compares the RSE features described in this topic to equivalent or similar features in PDM.

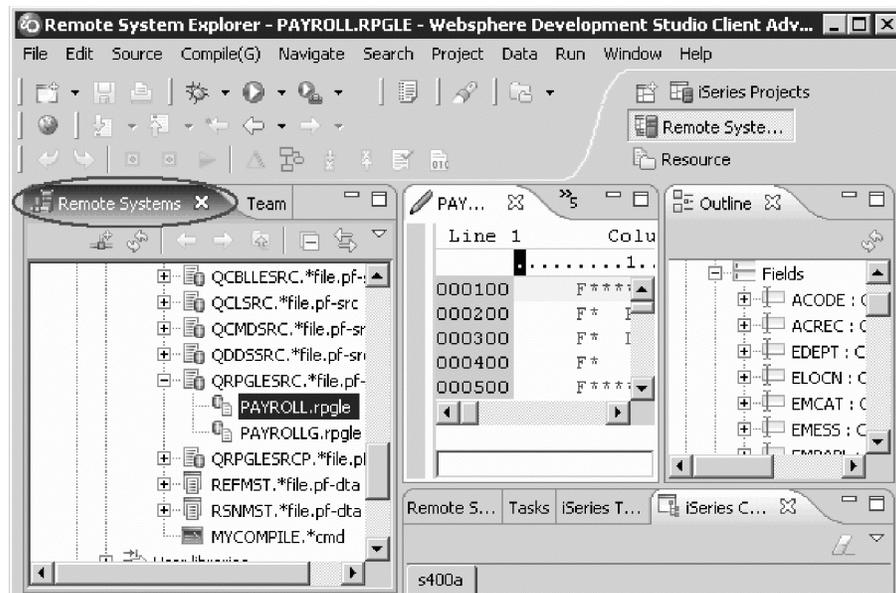
Table 1.

In RSE	In PDM
Create a connection	Start an emulator session
Create filters with generic values	Create filters with generic values
Expand a container to view additional items	Option 12
Specify multiple levels of generic items	Not available
Filters persist between sessions	Previous parameters for the WRKxxxPDM command are remembered
Create complex lists by defining multiple filter strings in a filter to list members in different files	List members in one source physical file in a single library
Flexible search patterns permit searching of filters	Single search pattern with option 25 or with FNDSTRPDM
All search results are available in the Remote Search view	Search results and members are available one at a time in the order that the matches are found

Remote Systems view

Use the Remote Systems view to navigate and list objects that you need to access to develop your applications.

Drill down, or expand items to see their children. Right-click to gain access to actions available in a pop-up menu. Standard actions such as drag and drop, copy, paste, delete, and rename are all available through the pop-up menu. These options are quite powerful in comparison to PDM. You can use copy and paste, or drag and drop, to copy or move members and even objects from one System i to another (no more SAVOBJ and FTP!). The pop-up menu also contains many other actions that can be performed on items, and allows you to create additional actions of your own.



See User actions, and the topic *Manipulating items in the Remote System Explorer* in the online help in the client product. See also *Viewing and accessing objects in the Remote System Explorer* in the tutorials.

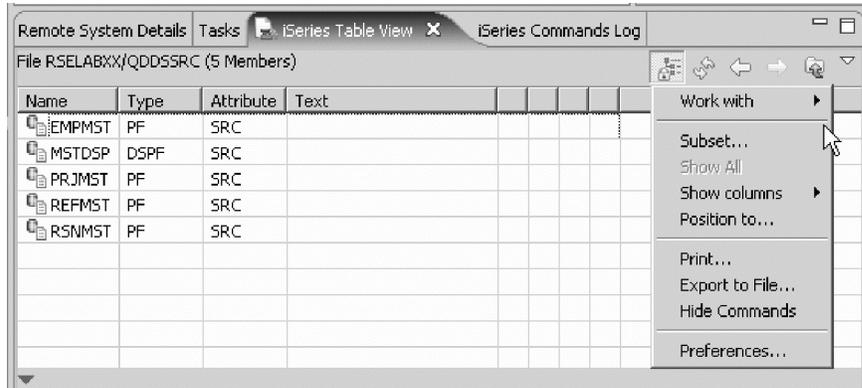
System i Table view

The System i Table view displays the same information as the Remote Systems
 # view, with the added ability to sort items, view descriptions, and perform more
 # PDM-like actions.

With the System i Table view, you can see the properties of all items at the same
 # time; they are displayed as rows across the table. The view takes the currently
 # selected file, library, or filter in the System i Objects subsystem as input, and
 # displays the contents in the table when you select the Show in Table option from
 # the pop-up menu. You can also use Work with actions from the System i Table
 # view itself to populate the view with libraries, objects, or members.

You can open the view directly by selecting the System i Table view tab at the
 # bottom of the Remote System Explorer perspective, or by selecting the **Show in
 # Table** action item in the pop-up of the Remote Systems View. You can use the
 # **Work with** menu to generate lists. The **Work with** menu keeps a small list (10) of
 # previously displayed lists in the System i Table view. The command line appears at
 # the bottom of the System i Table view, and allows you to enter commands, or
 # parameters for actions.

#



#

#

You can modify which columns appear in the System i Table view. You can choose to hide or show any individual column. You can type a character to bring up the Position To dialog. This dialog allows you to quickly scroll to your desired item.

#

Tips:

#

- Click on column headings to sort by that column.
- Use the Show in Table View to show filter contents in the System i Table view.
- Collapse the command line for a clean screen and the ability to see more items.
- Double-click on the System i Table view tab, to maximize the view to the full workbench. You will then be able to see more items in one screen.
- Use filters to generate complicated lists, use the Work with submenu to get access to infrequently used or simple lists.

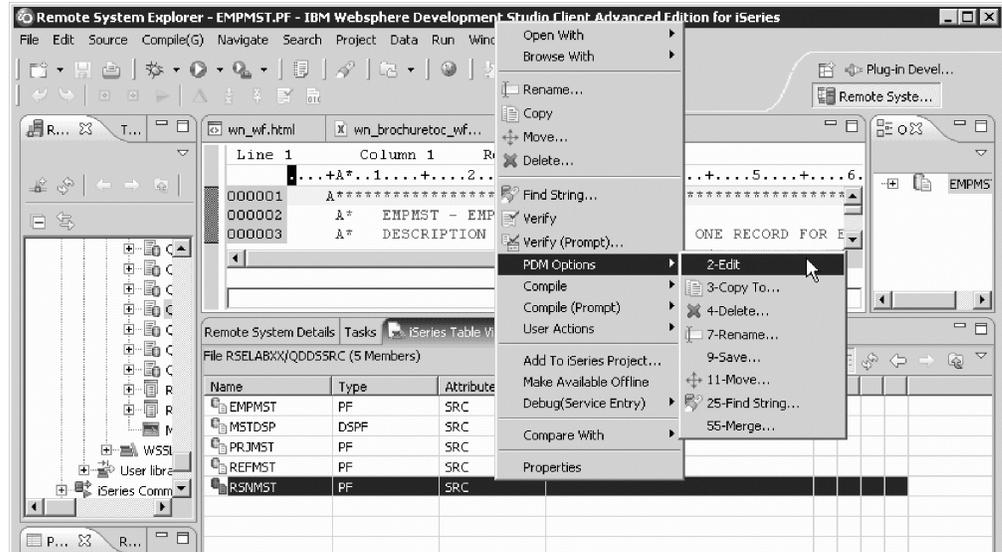
#

System i Table view actions

#

Like the Remote Systems view, the System i Table view has actions that can be invoked on items in it. As in the Remote Systems view, access to the action is provided through a pop-up menu from clicking with the right mouse button. In the pop-up from the System i Table view, you will see a list of actions with their PDM option number to make the menu feel more familiar. Use the **User actions** menu to create and add your own actions. User actions added either in the Table or Remote Systems view, appear in the **User Actions** menu in both views.

#



#

#

Tips:

- Use the Show in Table action from inside the System i Table view to go from a list of libraries to a list of objects.
- Double-click a member to open it in “Remote Systems LPEX Editor” on page xliii in edit mode.
- Click on the item name to open an edit cell in the Table view directly to quickly rename an item. The Description column, and member type column are also edit enabled, and allow you to quickly change the values in those columns.
- Access an object's properties by selecting the **Properties** menu item in the pop-up menu in the System i Table view or Remote Systems view.
- See *Managing objects in the System i Table view* in the online help in the client product. See also *Viewing and accessing objects in the Remote Systems Explorer* in the tutorials.

#

User actions

#

User actions allow you to extend the System i Table view and the Remote Systems view with the action that you use. You can create your own action, where you will be able to prompt the command to run, and to define how a command is executed.

#

#

Note: RSE has three command modes:

#

1. Normal: RSE jobs run in batch, so even though the mode is normal (meaning in this case immediate), you still cannot run interactive commands like STRPDM
2. Batch: Commands are submitted to a new batch job
3. Interactive: Commands are run interactively in a STRRSVSR job

#

#

#

#

When creating actions use the **Insert variable** button to see the list of available variables. Actions are very customizable and you can:

#

#

- Specify if the action should refresh the Remote Systems view or System i Table view after running
- Specify whether or not the command should be called once for each selected object, or once for all the objects. This gives greater flexibility than PDM. For

#

#

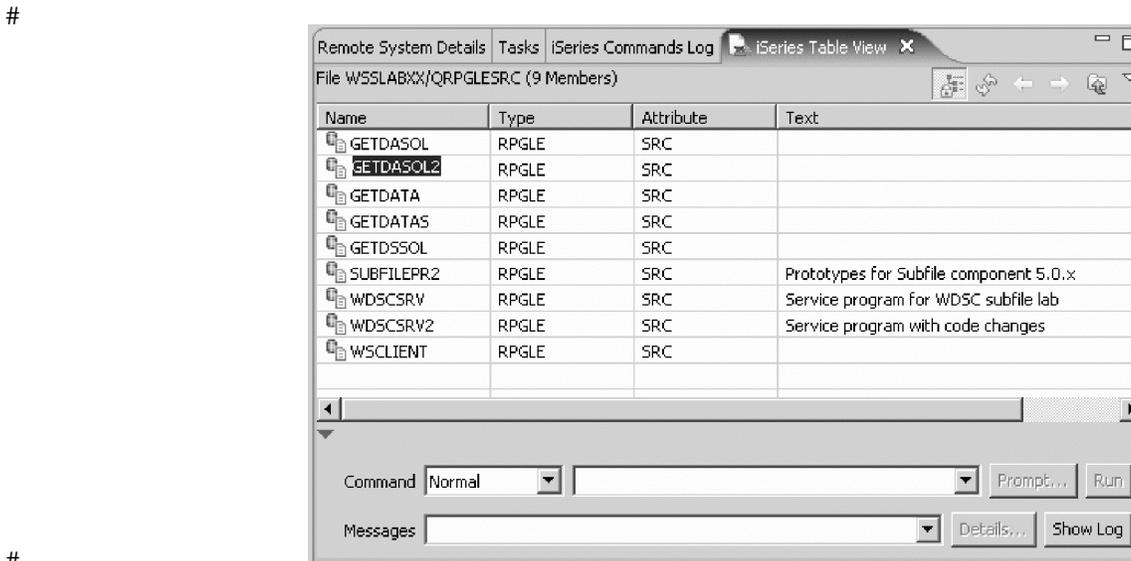
example, in RSE you could define a Save Object action, which would allow you
 # to select several objects and when invoked, would generate a single command to
 # save all of the selected objects to one save file.

- # • Actions can be refined so that they are only displayed for appropriate types.
 # There are several types predefined, but user types can be easily added to the list.
 # For example, this allows you to have an action for only *PGM objects, or only
 # CBLLE members.
- # • Selective prompts can be used when defining the CL command to run for the
 # action.

One of the advantages of RSE user actions is that they can be named, which makes
 # them easier to use and remember. (For more information, see the topic *Managing*
 # *user actions (user options)* in the online help in the client product. See also *Creating a*
 # *user action* in the tutorials.

Command line

The System i Table view contains a command line.

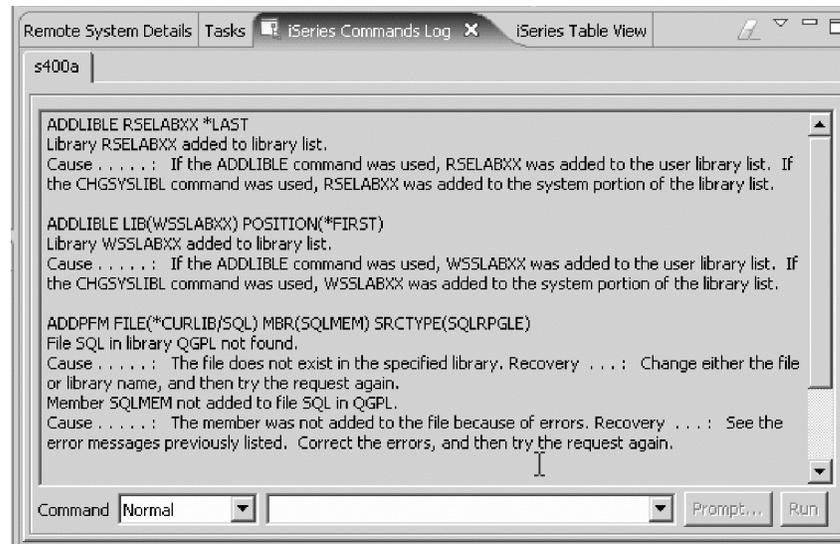


You can use the command line to run any command, or in the System i Table view,
 # to specify additional parameters for PDM options. The results of any commands
 # are displayed in the messages field. You can use familiar PDM keys:

- # • **F9** to retrieve the last command
- # • **F4** to prompt a command

The **Show Log** button can be used to view the System i Commands Log view.

#



#

#

Command execution mode can also be selected.

#

Tips:

#

- The System i Table view command line is good for quick commands
- The System i Commands Log view is better for commands where the results are more critical, and where you want to see second level help on the messages.
- The System i Commands Log also show the results of compiles, and users actions, either run through the Remote Systems view or the System i Table view.
- See the topic *Running programs and commands from the System i Table view* in the online help in the client product. See also *Submitting commands in the System i Table view* in the tutorials.

#

#

#

#

#

#

#

#

#

#

Compiling

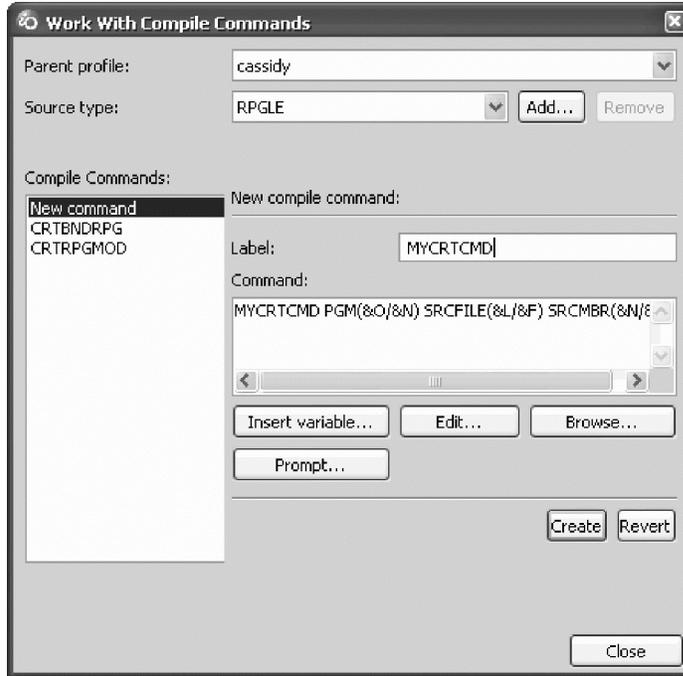
#

Compile actions are grouped into two menus: with and without prompting. You can add your own compile commands to the compile menu. This is much the same as adding a user action.

#

#

#



#

#

Compile actions are different from other actions: The results of the command itself appear in the System i Commands Log, but for commands which support event files, errors generated by the compiler are displayed in the Error List view. For more information about event files, see the topic *Events File format* in the online help in the client product.

#

Tips:

#

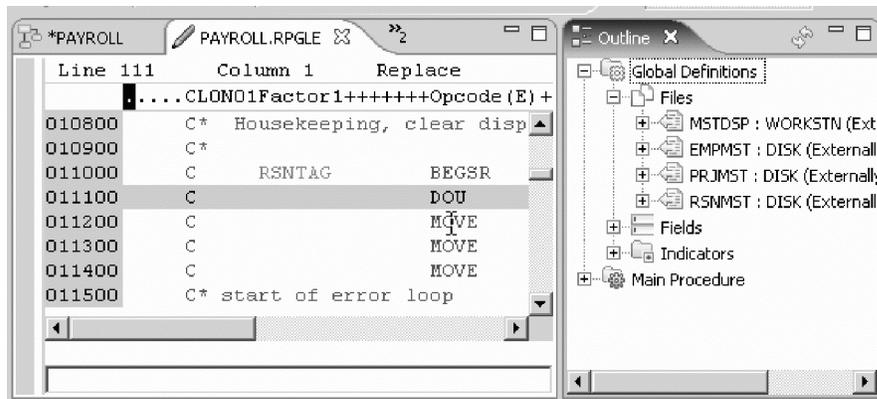
- Additional command execution preferences are available in the **Window > Preferences > Remote Systems > System i > Command Execution** preference page
- Compile commands use the same defaults as the host
- The compile actions remembers the last used compile command by member type
- Add your own compile commands and specify the source types that they should apply to, or refine existing compile commands, by modifying the properties and the CL command parameters
- The default is to compile in batch, use the Command Execution preferences to specify additional SBMJOB parameters, or to switch to normal execution (note that this is technically not interactive, because, as previously mentioned, RSE jobs run in batch)
- See the *Compiling your programs* topic in the online help in the client product. See also *Verifying and compiling source* in the tutorials.

#

Comparing the System i Table view to PDM

#

The following table compares the features of the System i Table view described in this topic to equivalent or similar features in PDM.

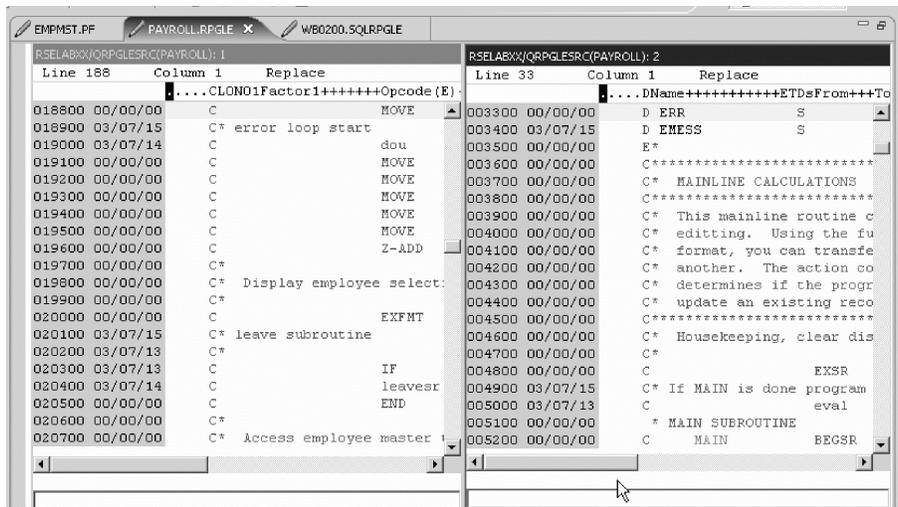


Note that the prefix area contains the sequence numbers of the member. The prefix area in Remote Systems LPEX Editor supports SEU commands (for example, CC, B, A, LL).

Also note that for many source types, an Outline view appears. The Outline view displays the outline of your source and can be used to navigate within your source.

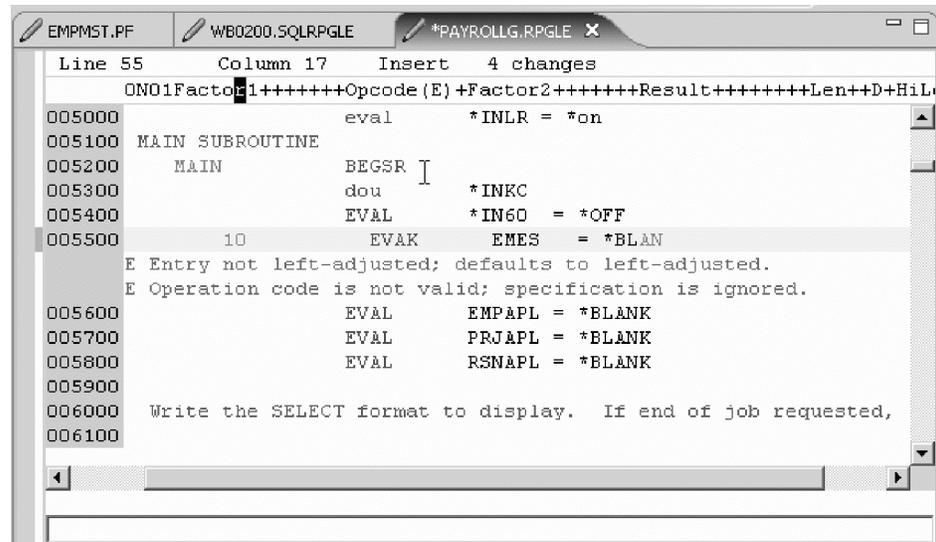
Tips:

- By default the Date area does not appear in the editor. If you want the date to appear all the time, you need to set the preference in **Window > Preferences > Remote Systems > Remote Systems LPEX Editor**. You can turn on the date area for a single session through the **Source** menu in the pop-up in the editor view.
- While you can use λ , $\lambda\lambda$, λn to exclude lines, a + appears which allows you to easily peek at and hide excluded lines.
- In addition to excluded lines, use the **Filter** menu in the pop-up menu in the editor view to show comments, control statements, for example. Each language has its own list of items that it can display in the filter menu.
- The pop-up menu in the editor view only shows appropriate items depending on source type being edited, cursor position within the source, and whether or not there is text selected.
- You can use the pop-up menu or **Ctrl+2** to split the current editor view to work with different parts of the same member. Up to 5 splits are allowed.

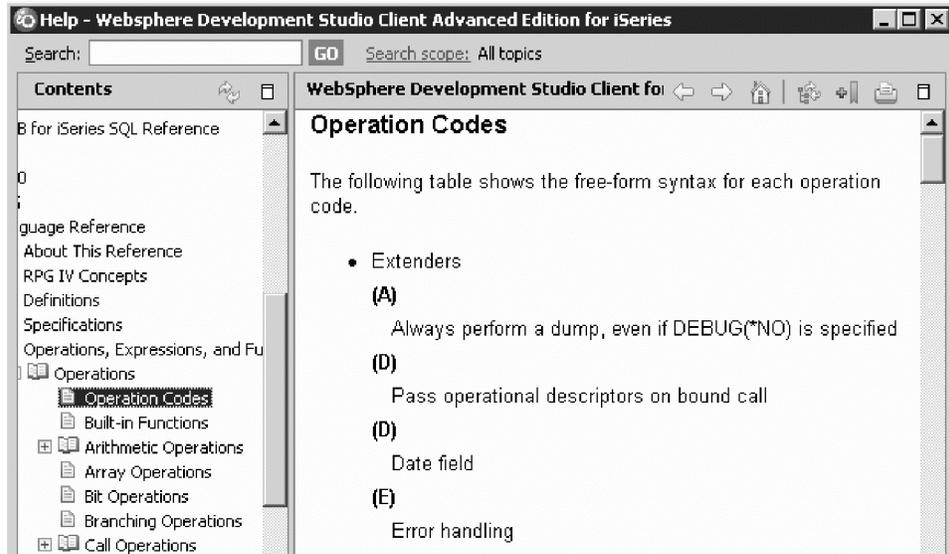


Syntax checking, prompting, and help

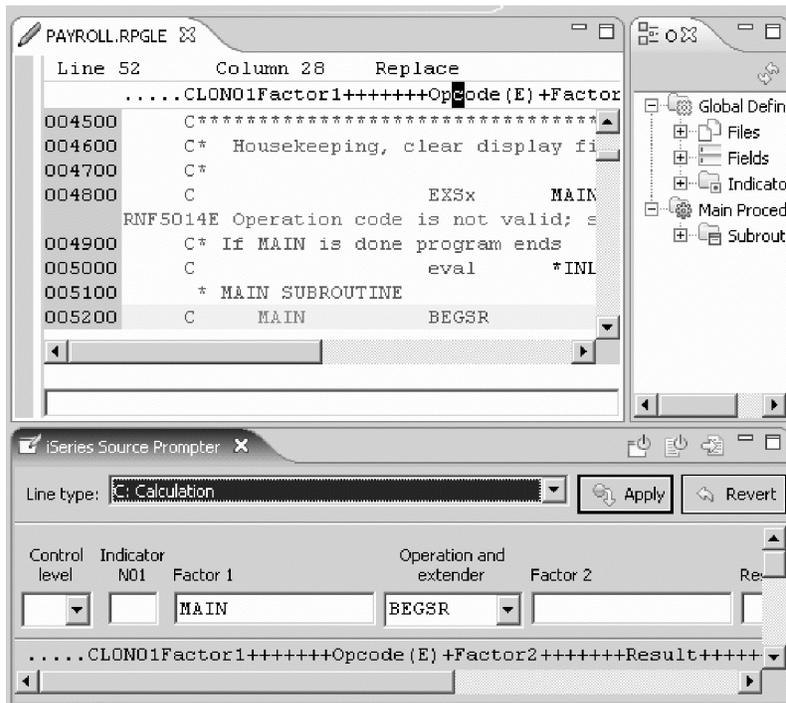
The Remote Systems LPEX Editor has automatic syntax checking. Errors are immediately visible. All of the syntax errors are embedded into the editor view and there is no need to move to the bottom of the screen to scroll through errors. The Remote Systems LPEX Editor uses the latest language syntax to check syntax for DDS, RPG and COBOL. An active connection is required for syntax checking SQL and CL. CL does cache syntax information, so syntax checking may be available when disconnected if cached information exists.



Help (F1 in the Remote Systems LPEX Editor) is not just available for errors, but for source too. Pressing F1 for context-sensitive help opens the reference manual at the topic for the code you are editing. For example, when you press F1 on an ILE COBOL statement, the help for that statement opens in the help browser. Reference manuals are also accessible through the **Source** menu in the Remote Systems LPEX Editor. This reduces the requirement for printed manuals.



F4 allows you to prompt when editing in the Remote Systems LPEX Editor. For languages other than CL, the Prompt view opens and allows you to modify your source. For CL, a window opens with your prompt. **F1** context-sensitive help is available from all prompts.



For more information, see the topic *Editing RPG, COBOL, CL, C, C++, and DDS members* in the online help in the client product. See also *Editing source* in the tutorials.

Verifiers and System i Error List view

Syntax checking ensures that there are no errors on the line that you type, but the Remote Systems LPEX Editor provides an additional check, called the Verifier. A verifier does the same syntax checking and semantic checking that the compiler does, without generating an object. This means that if you try to use an undeclared variable, the verifier will let you know.

The verifier function is available for COBOL, RPG and DDS from the Source menu, or by pressing **Ctrl+Shift+V**. Use the **Source** menu to verify with prompt and specify additional options.

Any errors detected by a verify will appear in the System i Error List view, exactly as those from a compile.

ID	Message	Severity	Line	Location	Connec...
RNF3438	LIKE keyword is expected for field EME but ...	30	34	RSELABXX/QRPGLESRC(PAYROLL)	TORAS...
RNF5178	ENDSR operation is missing for subroutine ...	30	1	RSELABXX/QRPGLESRC(PAYROLL)	TORAS...
RNF7030	The name or indicator EME is not defined.	30	34	RSELABXX/QRPGLESRC(PAYROLL)	TORAS...
<input checked="" type="checkbox"/> RNF7030	The name or indicator RSNTAX is not defined.	30	95	RSELABXX/QRPGLESRC(PAYROLL)	TORAS...
<input checked="" type="checkbox"/> RNF7018	The operand RSNTAX of EXSR is not a subr...	30	95	RSELABXX/QRPGLESRC(PAYROLL)	TORAS...
RNF5184	Factor 2 must be '0' or '1' when the Result i...	30	364	RSELABXX/QRPGLESRC(PAYROLL)	TORAS...
<input checked="" type="checkbox"/> RNF3703	The subfield or parameter definition is not ...	20	34	RSELABXX/QRPGLESRC(PAYROLL)	TORAS...
<input checked="" type="checkbox"/> RNF3602	Entry is not blank for a field definition; def...	20	34	RSELABXX/QRPGLESRC(PAYROLL)	TORAS...
<input checked="" type="checkbox"/> RNF3602	Entry is not blank for a field definition; def...	20	34	RSELABXX/QRPGLESRC(PAYROLL)	TORAS...
<input checked="" type="checkbox"/> RNF3308	Keyword name is not valid; the keyword is i...	20	34	RSELABXX/QRPGLESRC(PAYROLL)	TORAS...
<input type="checkbox"/> RNF7031	The name or indicator *IN05 is not referen...	0	1	RSELABXX/QRPGLESRC(PAYROLL)	TORAS...
<input type="checkbox"/> RNF7031	The name or indicator *IN06 is not referen...	0	1	RSELABXX/QRPGLESRC(PAYROLL)	TORAS...
<input type="checkbox"/> RNF7031	The name or indicator *IN07 is not referen...	0	1	RSELABXX/QRPGLESRC(PAYROLL)	TORAS...
<input type="checkbox"/> RNF7089	RPG provides Separate-Indicator area for f...	0	27	RSELABXX/QRPGLESRC(PAYROLL)	TORAS...
<input type="checkbox"/> RNF7031	The name or indicator RSNTAG is not refer...	0	110	RSELABXX/QRPGLESRC(PAYROLL)	TORAS...
<input type="checkbox"/> RNF7031	The name or indicator *IN03 is not referen...	0	1	RSELABXX/QRPGLESRC(PAYROLL)	TORAS...
<input type="checkbox"/> RNF7031	The name or indicator *IN04 is not referen...	0	1	RSELABXX/QRPGLESRC(PAYROLL)	TORAS...

The System i Error List view allows you to insert the errors into the editor view by double-clicking. Use the **F1** key to get help for errors, and use the **View** menu to filter errors that you don't want to see (for example, perhaps you want to ignore informational messages). The menu can also be used to dictate whether or not, and how, error messages are inserted into the editor view.

To cleanup any inserted errors, from the editor you can use **Ctrl+F5** to refresh the list. The refresh action:

- Removes any syntax, verifier, or compile errors
- Clears any excluded lines

Or you can use the **Remove messages** menu option from the **Source** menu.

The System i Error List view can also be used as a To Do list. As you modify lines, they are either marked with an X to indicate that the line has been deleted, or with a check mark, to indicate that an error has been addressed. Only another verify will ensure that the errors have truly been fixed.

Using the verifier has the following advantages:

- You can ensure a clean compile before actually compiling. This can be important for machines where compiling must only occur on off-peak times.
- If you are working offline, you can ensure that you are working with source that will compile when you reconnect to your System i.

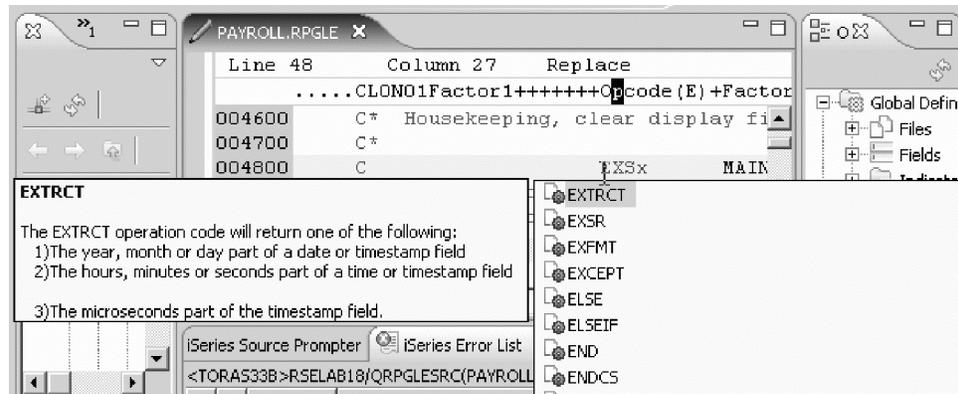
The System i Error List view (whether being used for compile results or verify) has several advantages as well:

- There is no need to switch between a spooled file and the source, or have two emulators open, as both are visible simultaneously, and you can have all the errors inserted into the source.
- When used as a To Do list, it is easy to ensure that all errors are addressed. There is no need to fix an error, recompile, fix another, and so on, until all errors are addressed.
- When an error occurs in a /COPY or a /INCLUDE member in RPG, or a Copy book in COBOL, double-clicking the error opens that member quickly for you and inserts the errors, as with the primary source member.
- F1 help for errors helps you fix the error quickly without using a reference manual.
- Using the preferences you can hide any messages you don't want to look at (such as informational or warning messages), making it easier to ensure that you address important errors quickly.
- For more information, see the topic *Verifying* and the topic *The Error List view* in the online help in the client product. See also *Verify the source* in the tutorials.

Content assist, templates, and RPG wizards

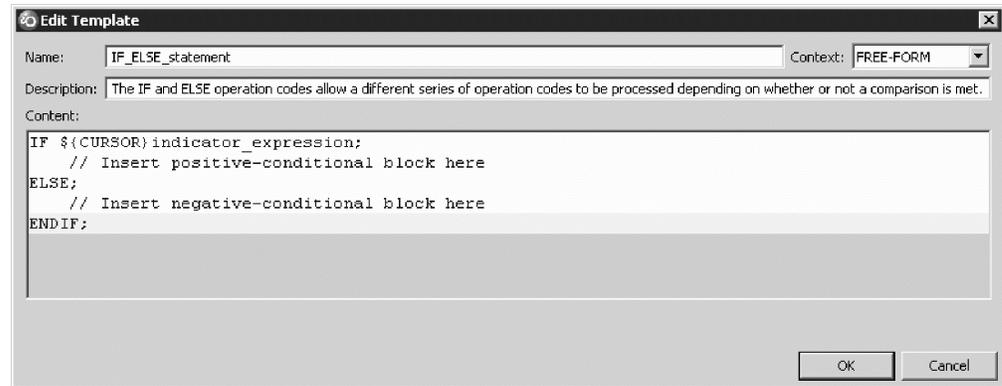
The Remote Systems LPEX Editor has several functions which help you quickly enter code.

Content assist (**Ctrl+Space**) will propose, display, and insert code completions at the cursor position. Based on the previously typed characters, or in the case of RPG, column position as well, invoking content assist presents you with possible valid completions to the code you have already entered. For example, this can be useful when you are unsure of the number of parameters required for a COBOL statement, or even the parameter types.



For more information, see the topic *Content assist* in the online help in the client product.

Templates can be used to generate frequently used blocks of code. They can be imported and exported, which means they can be shared. For example, if you have a standard header that must be added to each program, or a standard interface, you can define a template, and insert it by typing its name and then **Ctrl+Space**.



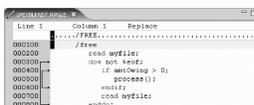
For more information, see the topic *Completing code with content assist* and the topic *Templates* in the online help in the client product. See also *Verifying the source* in the tutorials.

Tip: The content assist function and Outline view in COBOL is driven by the information generated by a verify. This is what pulls in external information like procedures found in a COPY member, or fields and records from a display file. It's important to refresh the Outline view at least once before invoking content assist, or only limited content assist will be available.

Additional Remote Systems LPEX Editor parser action and preferences

Additional preferences and actions are available for the System i languages:

- **Column Sensitive Editing:** This function is useful for column-sensitive languages like RPG and DDS. Normally in a windows application, inserting and deleting text pushes the remaining text left or right, and for these languages results in a syntax error. Enabling column sensitive editing limits insertion and deletion to the columns specified for the language.
- **Signatures:** Available for RPG and DDS, enabling this feature automatically flags each line with the specified signature. Note that modified lines in the Remote Systems LPEX Editor have the date changed as in SEU, regardless of member type.
- **Automatic-uppercasing:** Uppercases modified lines. Available for CL, DDS, RPG, COBOL members
- **Automatic-indent:** Indents the cursor when enter is pressed on the following line to help pretty-print your source. Available for CL, RPGLE.
- **Automatic-formatting:** Formats your source as you enter it, according to specified preferences. Available for CL and free-form SQLRPGLE.
- **Open/Browse /COPY member or Copy book:** For RPG and COBOL languages, you can open or browse members referred in the source through the pop-up in the Editor menu.
- **Show block nesting:** Using **Ctrl+Shift+O**, or from the **Source** menu in the pop-up menu, you can display an arrow indicating the nesting level at the cursor location.



- **Show fields:** When a file is referenced in a program, you can use this menu option from the pop-up menu to show the fields in the file in the System i Table view. Available in RPG, COBOL, and CL.
- **RPG actions:**

- Convert to free form (RPGLE)
- Convert to ILE (RPG)
- Show indentation (RPG, RPGLE)

Content assist, templates, and RPG wizards, are designed to speed up and improve your editing experience.

Additional LPEX keys

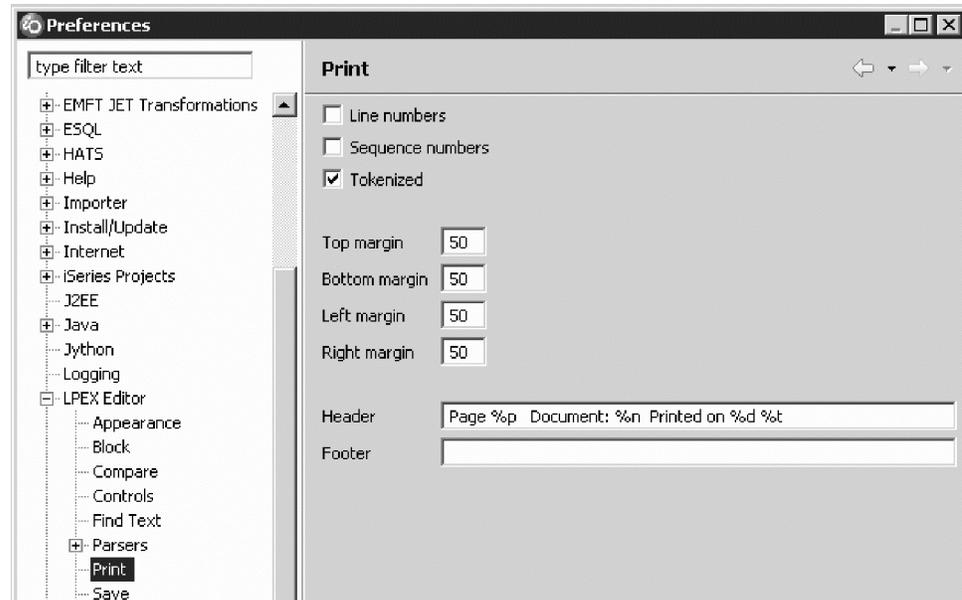
In the Remote Systems LPEX Editor, most functions are available through menus and keystrokes. Here is a list of additional keys you might find useful in LPEX:

Table 3.

Key combination	Description
Ctrl+Home	Go to the top (like TOP in SEU)
Ctrl+End	Go to the end (like BOTTOM in SEU)
Ctrl+L	Go to line number (also entering line number in prefix area like SEU works)
Alt+S	Split a line
Alt+J	Join a line
Alt+L	Select a line
Alt+R	Select a rectangle
Ctrl+W	Show all lines (useful when lines are filtered out)
Ctrl+Z	Undo
Ctrl+Y	Redo
Ctrl+S	Save
Ctrl+M	Match (selects matching brackets, and for languages like CL and RPG, control statements like DO/ENDDO, IF/ENDIF)
Ctrl+Shift+M	Find match

Printing

Like most Windows applications, printing can be done through the **File > Print** menu option or by pressing **Ctrl+P**. This can be done while editing. Printing also tokenizes the printed source, as long as you select the **Tokenized** checkbox. Printing in the Remote Systems LPEX Editor prints to your Windows printer, not the System i printer. Print options are available in **Window > Preferences > LPEX Editor > Print**.



The following substitution variables are available for use in the header and footer:

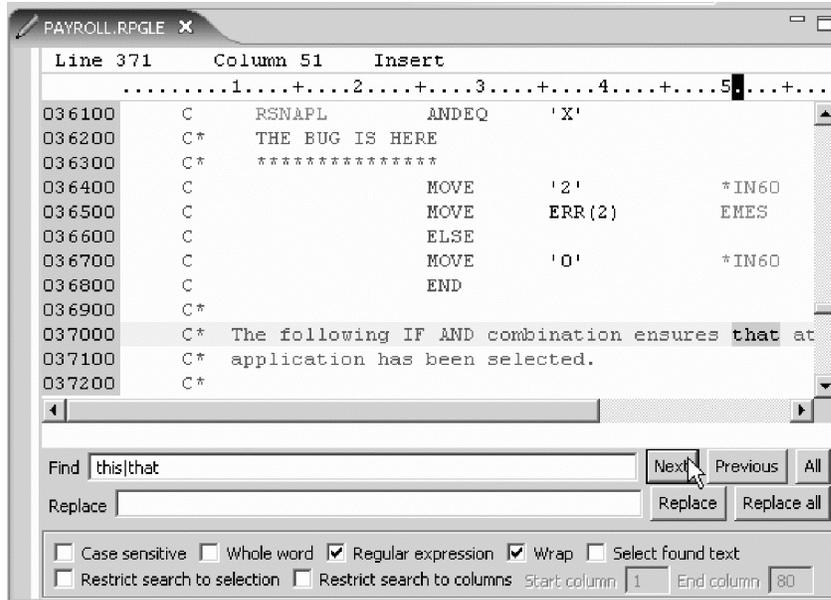
- %p: page number
- %n: source name, base file name, or document name
- %f: full path file name or document name
- %d: date
- %t: time

Tips:

- To print to a host printer, add a user action in the Remote Systems view and the System i Table view that invokes the STRSEU command with the print option.
- Printing with tokenization is best done on a color printer

Find and replace in the Remote Systems LPEX Editor

In the Remote Systems LPEX Editor you can use **Ctrl+F** to open the Find function in LPEX. The search function is very flexible. You can specify regular expressions that allow you to search for a pattern. For example, if you specify `this|that` as the search string with the **Regular expression** check box selected, then the editor searches for lines with `this` or `that` on them. You can use **Ctrl+N** or **Shift+F4** to find the next match.



Tips:

- Ensure column sensitive editing is enabled to make sure that replace does not shift text inappropriately
- Click the **All** button to have all the lines that do not contain the search string filtered out, so that you only see matching lines (they can easily be shown again by clicking the + in front of the prefix area, or by pressing **Ctrl+W**)

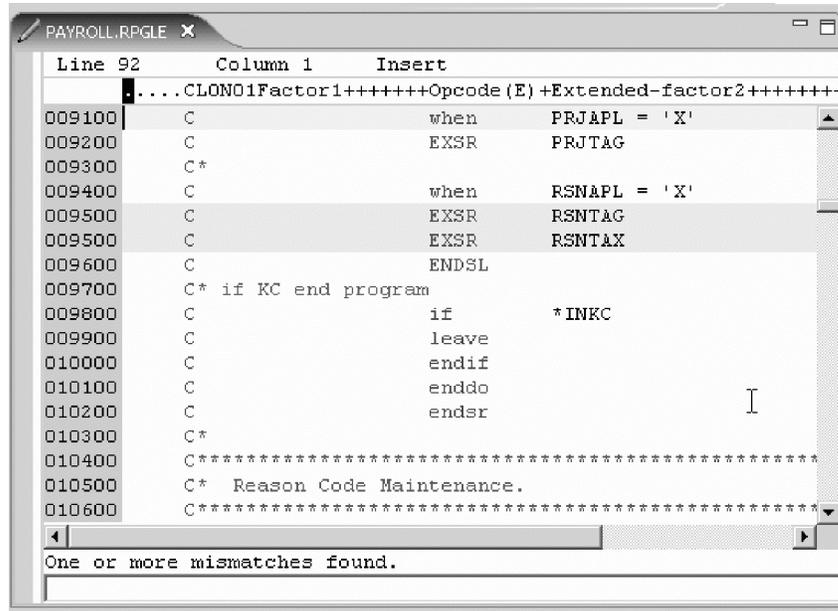
For more information, see the topic *Finding and replacing text* in the online help in the client product. See also *Finding and replacing text* in the tutorials.

Compare a file in the Remote Systems LPEX Editor

Comparing members in the Remote Systems LPEX Editor requires you to open a member in the editor. Once open, you can easily compare that member to another by selecting the **Compare** button in the toolbar or through the **Edit > Compare > Compare** file menu option.

Once the compare has been triggered, source appears merged with different lines flagged in color. Pink is the color used for the source being compared to, and yellow is the color for the opened source.

Unlike on the System i , where you have to flip between the spooled file and the source opened in SEU, comparing in the Remote Systems LPEX Editor allows you to continue to modify the member that was opened originally. Use **Ctrl+Shift+N** to navigate to the next mismatch and **Ctrl+Shift+P** for the previous mismatch. If you do modify source, you can use **Ctrl+Shift+R** to refresh the compare and finally **Edit > Compare > Clear** to end.



Tips:

- Specify additional preferences in **Window > Preferences > LPEX Editor > Compare**.
- Unlike other compare tools in eclipse, the Remote Systems LPEX Editor is sequence number aware, and will not mismatch lines just because the sequence number has been modified.
- For more information about comparing a file, see the topic *Editing RPG, COBOL, CL, C, C++, and DDS members* in the online help in the client product. See also *Comparing file differences from the Remote Systems view* in the tutorials.

Compile from the Remote Systems LPEX Editor

When you have source open in the Remote Systems LPEX Editor, it's not convenient to go to the Remote Systems view or System i Table view to issue a compile. Instead you can use one of the following:

- Toolbar button (uses the last used compile command for the member type to compile without prompt)
- **Ctrl+Shift+C** (uses the last used compile command for the member type to compile without prompt)
- Compile menu (where you can choose to compile with and without prompts, and select whichever compile command you want for the item). If you have not saved prior to compiling, you are prompted to do so.

For more information, see the topic *Compiling* in the online help in the client product. See also *Compiling source remotely* in the tutorials.

Comparing the Remote Systems LPEX Editor to SEU

The following table compares the features of the Remote Systems LPEX Editor described in this topic to equivalent or similar features in SEU.

Table 4.

In the Remote Systems LPEX Editor	In SEU
Launch the editor from the pop-up menus on a member in edit or browse modes	Launch SEU with PDM option 5
Full screen mode for both edit and browse (double-click the editor tab). However, many more lines are visible in Remote Systems LPEX Editor in full screen than in SEU.	Full screen mode when browsing only (F13)
Split screen for edit and browse: <ul style="list-style-type: none"> • Drag and drop editor tabs to view more than on member at a time • Use editor view pop-up or Ctrl+2 to split the current editor view to work with different parts of the same member (up to 5 splits are allowed) 	Split screen and browse
Language tokens are displayed in colors (tokenization)	Not available
Provides a prefix area containing sequence numbers and supports SEU edit commands	Prefix area available for edit commands
Date area appears next to the sequence numbers. It is Off by default, but can be enabled through the preference, or the pop-up menu.	Date area is at the right, and is always enabled
View or hide excluded lines by expanding (clicking +), or collapsing (clicking -) items	Excluded lines cannot be viewed
Automatic syntax checking - all errors are immediately visible	Automatic syntax checking - first error is visible
Content assist, code templates, and RPG wizards are available to assist in code creation	Prompter is available to assist in code creation
Most editing functions are available through menus and keystrokes	All editing functions are available through keystrokes
Printing is available from the File menu, or Ctrl+P	Print with STRSEU, Option 6
Printing can be done while editing	Not available
A member opened in the editor for compare can be modified	To edit the source, switch from the spooled file to the source in SEU

Part 1. Compiling, Running, and Debugging ILE COBOL Programs

Chapter 1. Introduction

Common Business Oriented Language (COBOL) is a programming language that resembles English. As its name suggests, COBOL is especially efficient for processing business problems. It emphasizes describing and handling data items and input/output records; thus, COBOL is well adapted for managing large files of data.

This chapter provides the following:

- an introduction to the Integrated Language Environment (ILE)
- an introduction to ILE COBOL
- an overview of function that has been incorporated in ILE COBOL that is not available in OPM COBOL/400
- an overview of the major steps in creating a runnable program object
- an overview of other application development tools that are available to help you develop ILE COBOL applications more effectively.

Integrated Language Environment

The **Integrated Language Environment (ILE)** is the current stage in the evolution of IBM i program models. Each stage evolved to meet the changing needs of application programmers. For a full description of the concepts and terminology pertaining to ILE, refer to the *ILE Concepts* book.

#

The programming environment provided when the i5/OS was first introduced is called the **Original Program Model (OPM)**. COBOL, RPG, CL, BASIC and PL/1 all operated in this model. In Version 1 Release 2, the **Extended Program Model (EPM)** was introduced. EPM was created to support languages like C, Pascal, and FORTRAN. For a full description of the principal characteristics of OPM and EPM, refer to the *ILE Concepts* book.

The most significant difference between the OPM COBOL/400 environment and the ILE COBOL environment is how a runnable program object is created. The ILE COBOL compiler does not produce a runnable program object. It produces one or more **module objects** that can be **bound** together in various combinations to form one or more runnable units known as **program objects**.

ILE allows you to bind module objects written in different languages. Therefore, it is possible to create runnable program objects that consist of module objects written separately in COBOL, RPG, C, C++ and CL.

Major Steps in Creating a Runnable ILE COBOL Program Object

Figure 1 on page 4 illustrates the typical steps involved in the development of a runnable program object written in ILE COBOL:

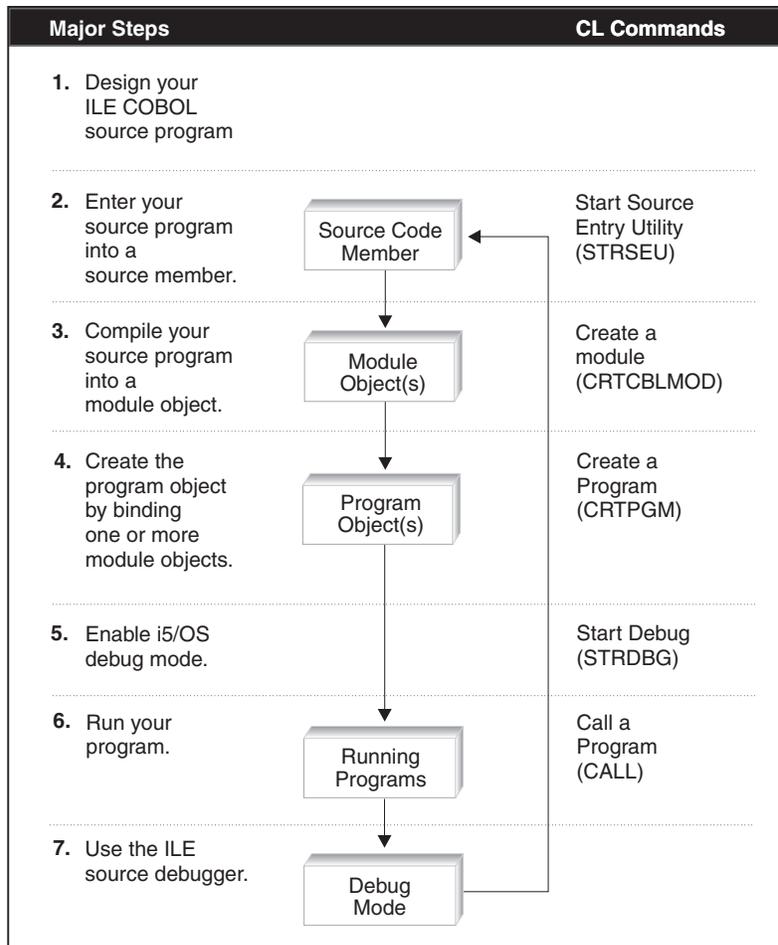


Figure 1. Major Steps in Creating a Runnable ILE COBOL Program Object

Steps 3 and 4 can be accomplished with a single command, CRTBNDCBL. This command creates temporary module objects from the ILE COBOL source program and then creates the program object. Once the program object is created, the module objects are deleted.

Designing Your ILE COBOL Source Program

The first step in creating a runnable ILE COBOL program object is to design your ILE COBOL source program.

An **ILE COBOL source program** consists of four divisions. The skeleton program in Figure 2 on page 5 shows the structure of an ILE COBOL source program. It can be used as a sample for designing ILE COBOL source programs.

ILE COBOL programs can be contained in other ILE COBOL programs. This concept is known as nesting and the contained program is known as a **nested program**. Figure 2 on page 5 shows how a nested ILE COBOL program is included in an outermost ILE COBOL program. Not all the entries provided in the skeleton program are required; most are provided for informational purposes only.

```

IDENTIFICATION DIVISION. 1
PROGRAM-ID. outermost-program-name.
AUTHOR. comment-entry.
INSTALLATION. comment-entry.
DATE-WRITTEN. comment-entry.
DATE-COMPILED. comment-entry.
SECURITY.
* The SECURITY paragraph can be used to specify
* copyright information pertaining to the
* generated module object. The first 8 lines
* of the SECURITY paragraph generate the
* copyright information that is displayed on
* the Copyright Information panel when the
* Display Module (DSPMOD) CL command is issued.
ENVIRONMENT DIVISION. 2
CONFIGURATION SECTION. 3
SOURCE-COMPUTER. IBM-ISERIES.
OBJECT-COMPUTER. IBM-ISERIES.
SPECIAL-NAMES. REQUESTOR IS CONSOLE.
INPUT-OUTPUT SECTION. 4
FILE-CONTROL.
SELECT file-name ASSIGN TO DISK-file-name
ORGANIZATION IS SEQUENTIAL
ACCESS MODE IS SEQUENTIAL
FILE STATUS IS data-name.
DATA DIVISION. 5
FILE SECTION.
FD file-name.
01 record-name PIC X(132).
WORKING-STORAGE SECTION.
77 data-name PIC XX.
LINKAGE SECTION.
PROCEDURE DIVISION. 6
DECLARATIVES
END DECLARATIVES.
main-processing SECTION.
mainline-paragraph.
ILE COBOL statements.
STOP RUN.
IDENTIFICATION DIVISION. 7
PROGRAM-ID. nested-program-name.
ENVIRONMENT DIVISION. 8
INPUT-OUTPUT SECTION.
FILE-CONTROL.
SELECT file-name ASSIGN TO DISK-file-name
ORGANIZATION IS SEQUENTIAL
ACCESS MODE IS SEQUENTIAL
FILE STATUS IS data-name.
DATA DIVISION.
FILE SECTION.
FD file-name.
01 record-name PIC X(132).
WORKING-STORAGE SECTION.
77 data-name PIC XX.
LINKAGE SECTION.
PROCEDURE DIVISION.
DECLARATIVES
END DECLARATIVES.
main-processing SECTION.
mainline-paragraph.
ILE COBOL statements.
EXIT PROGRAM.

END PROGRAM nested-program-name. 9
END PROGRAM outermost-program-name.

```

Figure 2. Example of ILE COBOL Program Structure

- The Identification Division **1** is the only division that must be included; all other divisions are optional.
- The Environment Division **2** is made up of two sections: the Configuration Section **3**, which describes the overall specifications of the source and object computers, and the Input-Output Section **4**, which defines each file, and specifies information needed for transmission of data between an external medium and the ILE COBOL program.
- The Data Division **5** describes the files to be used in the program and the records contained within the files. It also describes any internal Working-Storage or Local-Storage data items that are needed.
- The Procedure Division **6** consists of optional declaratives, and procedures that contain sections and/or paragraphs, sentences, and statements.
- This second Identification Division **7** marks the beginning of a nested ILE COBOL program which is contained within the outermost ILE COBOL program.
- Nested programs cannot have a Configuration Section **8** in the Environment Division. The outermost program must specify any Configuration Section options that may be required.
- Nested programs and the outermost program must be terminated by an END PROGRAM **9** header.

An ILE COBOL program is identified by the PROGRAM-ID in the IDENTIFICATION DIVISION. It contains a set of self-contained statements that perform a particular task.

In ILE, an ILE COBOL source program is considered to be an **ILE procedure**. If an ILE COBOL program contains nested ILE COBOL programs, each nested ILE COBOL program is an ILE procedure. The name of the nested program is only known within its containing program. If the nested program has the COMMON attribute, the name of the nested program is also known to other programs in the same compilation unit. ILE procedures are not to be confused with COBOL procedures, which are found in the Procedure Division of a COBOL program and contain sections, paragraphs, sentences, and statements.

For more information on writing your ILE COBOL program, refer to the *IBM Rational Development Studio for i: ILE COBOL Reference*.

Entering Source Statements into a Source Member

Once you have designed your ILE COBOL program, you must enter it into a source member.

```
# Use WebSphere Development Studio Client for System i. Your program editing
# tasks are simplified with the Remote Systems LPEX editor. The editor can access
# source files on your workstation or your System i. When a compilation results in
# errors, you can jump directly from the compiler messages to an editor containing
# the source. The editor opens with the cursor positioned at the offending source
# statements so that you can correct them.
```

The Source Entry Utility (SEU) command, Start Source Entry Utility (STRSEU), can also be used to enter and edit your ILE COBOL source statements. To help you enter accurate ILE COBOL statements into the system the SEU display corresponds to the standard COBOL coding form and as you enter or change a line of code, the COBOL syntax checker checks the line for errors.

A **compilation unit** is an outermost ILE COBOL program and any nested ILE COBOL programs within the outermost program. Multiple compilation units may be entered in a single source member.

For information on entering source statements, refer to Chapter 2, “Entering Source Statements into a Source Member,” on page 11.

Compiling a Source Program into Module Objects

Once you have completed entering or editing the source statements in a source member, you now need to create module objects using the Create COBOL Module (CRTCBMOD) command. This command compiles the source statements in the source member into one or more module objects. Each compilation unit in the source member creates a separate module object.

Module objects are the output of the ILE COBOL compiler. They are represented on the system by the type *MODULE. Module objects cannot be run without first being bound into program objects.

For more information on creating module objects using the CRTCBMOD command, refer to Chapter 3, “Compiling Source Programs into Module Objects,” on page 21.

Creating a Program Object

In order to create a runnable **program object**, module objects must be bound together. One or more module objects can be bound together to create a program object. Module objects need to be bound into program objects since only program objects can be run. Module objects written in various programming languages can be bound together to create a program object. For example, a program object could consist of COBOL or RPG module objects for the report, but a C module object for the calculations. A program object can be created using one of the following commands:

- Create Program (CRTPGM)
- Create Bound COBOL Program (CRTBNDCBL)

For information on these commands, refer to Chapter 4, “Creating a Program Object,” on page 77.

Running a Program Object

You run a program object by calling it. You can use one of the following ways to call a program object:

- The CALL CL command on any command line
- A high-level language CALL statement
- An application-oriented menu
- A user-created command
- The Run menu action or Run toolbar icon in the WebSphere Development Studio Client for System i workbench

For information on running a program, refer to Chapter 6, “Running an ILE COBOL Program,” on page 111.

Debugging a Program

The ILE source debugger is used to detect errors in and eliminate errors from program objects and service programs. You can use the ILE source debugger to:

- View the program source

- Set and remove conditional and unconditional breakpoints
- Step through the program
- Display the value of variables, structures, records, and arrays
- Change the value of variables
- Change the reference scope
- Equate a shorthand name to a variable, expression, or debug command.

For information on the debugger, refer to Chapter 7, “Debugging a Program,” on page 117.

Other Application Development Tools

The System i offers a full set of tools that you may find useful for programming.

IBM Rational Development Studio for System i

IBM Rational Development Studio for System i is an application development
package to help you rapidly and cost-effectively increase the number of e-business
applications for the System i. This package consolidates all of the key System i
development tools, both host and workstation, into one System i5 offering.

The host development tools have undergone major improvements. Recent compiler enhancements include new C and C++ compilers, completely refreshed from the latest AIX compilers, to replace the existing versions of these compilers. This will help customers and solution providers port e-business solutions from other platforms. ILE RPG has also made major enhancements. Improved Java interoperability and free-form C-Specs top the list of enhancements. COBOL has added z/OS migration capabilities as well as introducing some COBOL/Java interoperability capabilities. For a complete list of new features for a language, please see the What's New section in the Language Reference or Programmer's guide.

The following components are included in WebSphere Development Studio for System i.

Host components:

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

Workstation components:

- # IBM WebFacing Tool
- # System i development tools: Remote System Explorer and System i projects
- # Java development tools (with System i enhancements)
- # Web development tools (with System i enhancements)
- Struts environment support
- Database development tools
- Web services development tools
- # System development tools
- XML development tools
- CODE
- VisualAge RPG

- # • Integrated i5/OS debugger

WebSphere Development Studio for System i

WebSphere Development Studio for System i (Development Studio Client) is an
application development package of workstation tools that helps you rapidly and
cost-effectively increase the number of e-business applications for the System i.

This package consolidates all of the key System i workstation-based development
tools into one System i5 offering.

WebSphere Development Studio for System i Feature List:

The workbench-based integrated development environment

IBM WebSphere Development Studio Client for System i uses WebSphere Studio Workbench (WSWB) version 2.1.

The IBM WebFacing Tool

The IBM WebFacing Tool can convert your DDS display source files into an application that can be run in a browser.

Remote System Explorer and System i Development Tools

The Remote System Explorer, included as a part of System i development
tools, encompasses the framework, user interface, editing, and file,
command, and job actions of System i capability.

System i Java development tools

Java development tools and System i Java development tools give you the
ability to develop Java applications and write, compile, test, debug, and
edit programs written in the Java programming language for Java
applications development.

System i Web development tools

Web development tools give you the ability to create new e-business
applications that use a Web-based front end to communicate with the
business logic in an ILE and non-ILE language program residing on an
i5/OS host.

Struts environment support

Development Studio Client offers support for Struts and the Web Diagram editor.

Database development tools

Database development tools support any local or remote database that has a Java Database Connectivity (JDBC) driver.

Web services development tools

Web services development tools allow developers to create modular applications that can be invoked on the World Wide Web.

System development tools

System development tools are used to test applications in a local or
remotely installed run-time environment.

XML development tools

XML development tools support any XML-based development.

CODE (CoOperative Development Environment)

CODE is the classic set of Windows tools for System i development. It
gives you a suite of utilities for creating source and DDS files, and
managing your projects.

#

VisualAge RPG

VisualAge RPG is a visual development environment that allows you to create and maintain client/system applications on the workstation.

Integrated i5/OS debugger

The integrated i5/OS debugger helps you debug code that is running on i5/OS or on your Windows system, using a graphical user interface on your workstation.

If you want to learn more about WebSphere Development Studio for System i, see the most current information available on the World Wide Web at:
<http://www.ibm.com/systems/i/infocenter/>.

Chapter 2. Entering Source Statements into a Source Member

This chapter provides the information you need to enter your ILE COBOL source statements. It also briefly describes the tools and methodology necessary to complete this step.

See Using the application development tools in the client product for information about getting started with the client tools.

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

1. Enter source statements using the Source Entry Utility (SEU). This is the method documented in this chapter.
2. Enter the source statements from a diskette or a tape by using the IBM i CL commands, CPYFRMTAP and CPYFRMDKT.

To obtain information on how to enter source statements using the CL commands, refer to the *CL and APIs* section of the *Database and File System* category in the **i5/OS Information Center** at this Web site -<http://www.ibm.com/systems/i/infocenter/>.

3. Enter the source statements into a stream file using the Edit File (EDTF) CL command. This command is a general purpose stream file editor that can also be used to enter ILE Cobol source statements. However, this editor does not provide syntax checking capability and special format lines like SEU to aid in source entry. If you want to store your ILE Cobol source statements in a stream file but also want to use the syntax checking features of SEU, perform the following steps:
 - Enter the source statements into a file member using SEU
 - Use the Copy to Stream File (CPYTOSTMF) command to copy the contents of the file member to a stream file

Creating a Library and Source Physical File

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

A **library** is a system object that serves as a directory to other objects. A library groups related objects and allows you to find objects by name. The object type for a library is *LIB.

A **Source physical file** is a file that stores members. These members contain source statements, such as ILE COBOL source statements.

To create a library called MYLIB, use the Create Library (CRTLIB) command:
CRTLIB LIB(MYLIB)

To create a source physical file called QCBLESRC in library MYLIB, use the Create Source Physical File (CRTSRCPF) command:
CRTSRCPF FILE(MYLIB/QCBLESRC)

Note: In the above example, the library MYLIB must exist before you can create the source physical file.

The ILE COBOL compiler supports an additional record length of 102; a field of 10 characters containing supplementary information is placed at the end of the record (positions 93-102). This information is not used by the ILE COBOL compiler, but is placed on the extreme right of the compiler listing. You are responsible for placing information into this field. If you want to use this additional field, create a source file with a record length of 102.

A source file is supplied where you can store your source records if you do not want to create your own file. This file, named QCBLLSRC, is in library QGPL and has a record length of 92 characters.

Starting SEU

To enter ILE COBOL source program using SEU, enter the Start Source Entry Utility (STRSEU) command, and specify CBLLE for the TYPE parameter. Specify SQLCBLLE for the TYPE parameter if your source program contains imbedded SQL.

If you do not specify a TYPE parameter, SEU uses the same type used when the member was last edited, as the default value. If you do not specify a TYPE parameter and you are creating a new member, SEU assigns a default member type associated with the name of the source physical file. For ILE COBOL, this default member type is CBLLE. For other methods of starting SEU, refer to *ADTS for AS/400: Source Entry Utility*.

See Using the application development tools in the client product for information about getting started with the client tools.

Using the COBOL Syntax Checker in SEU

To use the COBOL syntax checker in SEU, specify the TYPE (CBLLE) parameter of the STRSEU command. The COBOL syntax checker checks each line for errors as you enter new lines or change existing lines. Incorrect source statements are identified and error messages displayed, allowing you to correct the errors before compiling the program.

See Using the application development tools in the client product for information about getting started with the client tools.

Any time a source line is entered or changed, other lines of source code can be syntax checked as part of that unit of syntax-checking. The length of a single unit of syntax-checking is determined by extending from an entered or changed line as follows:

- A unit of syntax-checking extends towards the beginning of the source member until the beginning of the first source line, or until a period that is the last entry on a line is found.
- A unit of syntax-checking extends towards the end of the source member until the end of the last source line, or until a period that is the last entry on a line is found.

Because the COBOL syntax checker checks only one sentence as it is entered or changed, independent of sentences that precede or follow it, only syntax errors within each source statement can be detected. No inter-relational errors, such as undefined names and incorrect references to names, are detected. These errors are detected by the ILE COBOL compiler when the program is compiled.

Conversely, if a change is made to a sentence that is part of a comment-entry for an optional paragraph of the Identification Division, the syntax checker is not able to recognize that the context permits any combination of characters to be entered. It may generate multiple errors as it attempts to identify the contents of the sentence as a valid COBOL statement. This will be avoided if the comment-entry is written as a single sentence that starts on the same line as the paragraph name, or if the comment-entry is replaced by a series of comment lines.

If there is an error in a unit of syntax-checking, the part of the unit identified as being in error is presented in reverse image. The message at the bottom of the display refers to the first error in the unit.

Syntax checking occurs as you enter the source code. Error messages are generated by lines consisting of incomplete statements. These disappear when the statements are completed, as in the example:

```

Columns . . . : 1 71          Edit          TESTLIB/QCBLLESRC
SEU==>                                     ADDATOB
FMT CB .....-A+++B+++++-----
***** Beginning of data *****
0000.10      IDENTIFICATION DIVISION.
0000.20      PROGRAM-ID. ADDATOB.
0000.30      ENVIRONMENT DIVISION.
0000.40      CONFIGURATION SECTION.
0000.50          SOURCE-COMPUTER. IBM-ISERIES.
0000.60          OBJECT-COMPUTER. IBM-ISERIES.
0000.70      DATA DIVISION.
0000.80      WORKING-STORAGE SECTION.
0000.90      01 A      PIC S9(8) VALUE 5.
0001.00      01 B      PIC S9(8) VALUE 10.
0001.10      PROCEDURE DIVISION.
0001.20      MAINLINE.
0001.30          MOVE A
.....
***** End of data *****

F3=Exit  F4=Prompt  F5=Refresh  F9=Retrieve  F10=Cursor  F11=Toggle
F16=Repeat find  F17=Repeat change  F24=More keys
COBOL reserved word or special character 'T0' expected. 'T0' assumed.      +

```

Figure 4. COBOL Syntax Checker error message generated for an incomplete statement

```

Columns . . . : 1 71          Edit          TESTLIB/QCBLLESRC
SEU==>          ADDATOB
FMT CB .....-A+++B+++++*****
***** Beginning of data *****
0000.40      IDENTIFICATION DIVISION.
0000.50      PROGRAM-ID. ADDATOB.
0000.60      ENVIRONMENT DIVISION.
0000.70      CONFIGURATION SECTION.
0000.80          SOURCE-COMPUTER. IBM-ISERIES.
0000.90          OBJECT-COMPUTER. IBM-ISERIES.
0000.91      DATA DIVISION.
0000.92      WORKING-STORAGE SECTION.
0000.93      01 A      PIC S9(8) VALUE 5.
0000.94      01 B      PIC S9(8) VALUE 10.
0001.00      PROCEDURE DIVISION.
0001.10      MAINLINE.
0002.00          MOVE A
0003.00          TO B.
.....
***** End of data *****
F3=Exit  F4=Prompt  F5=Refresh  F9=Retrieve  F10=Cursor  F11=Toggle
F16=Repeat find  F17=Repeat change  F24=More keys

```

Figure 5. COBOL Syntax Checker error message disappears after statement is completed

An error message is generated after the first line is entered and disappears after the second line is entered, when the statement is completed.

The following regulations apply to syntax checking for ILE COBOL source code:

- Source code on a line with an asterisk (*) or a slash (/) in column 7 is not syntax checked. An asterisk indicates a comment line; a slash indicates a comment line and page eject.
- No compiler options are honored during syntax checking.
 For example, the syntax checker accepts both quotation marks or apostrophes as nonnumeric delimiters provided they are not mixed within one unit of syntax checking. The syntax checker does not check if the delimiter is the one that will be specified in the CRTCBMOD or CRTBNDCBL commands, or in the PROCESS statement.
- Character replacement specified by the CURRENCY and DECIMAL-POINT clauses of the SPECIAL-NAMES paragraph is not honored during interactive syntax checking.
- When using the REPLACING *Identifier-1* BY *Identifier-2* clause of the COPY statement and when either identifier includes reference modification, the COBOL syntax checker in SEU checks for matching parentheses only.
- The COPY statement and REPLACE statement are checked for syntax structure.
- Imbedded SQL statements are syntax-checked.

Example of Entering Source Statements into a Source Member

See Using the application development tools in the client product for information about getting started with the client tools.

This example shows you how to create a library and source physical file, start an edit session, and enter source statements using the Create Library (CRTLIB), Create Source Physical File (CRTSRCPF) and Start SEU (STRSEU) commands.

Note: In order to perform these tasks using these commands you must first have the authority to use the commands.

1. To create a library called MYLIB, type

All valid ILE COBOL characters except \$, @, and # are included in the Syntactic/Invariant Character Set 640. Characters in this set have the same code point in all single-byte EBCDIC code pages, except Code Page 290 (where the code points used for lower-case alphabetic characters in the other code pages are assigned to Katakana characters), and certain code pages which use a different code point for the " (quotes) character.

Note: The @ and # characters support IBM extensions and conventions. The @ character can appear as a conversion specifier in a FORMAT clause literal. The @ and # characters are accepted as valid when checking a literal that defines a program name in a CALL statement.

The ILE COBOL compiler will accept source code written in any single-byte or mixed-byte EBCDIC CCSID, except those based on Code Page 290 (for example, CCSID 290 or CCSID 930). If the source code is stored in a stream file, it may have a non-EBCDIC CCSID. In this case, the compiler will convert the stream file to an EBCDIC CCSID related to the stream file's CCSID before compiling the source code.

CCSIDs can help you to maintain the integrity of character data across systems.

Character Data Representation Architecture (CDRA) defines CCSID values to identify the code points used to represent characters, and to convert these codes as needed to preserve their meanings.

The Extended ACCEPT and DISPLAY statements do not support CCSID conversion.

Assigning a CCSID to a Source Physical File

A CCSID is assigned to each source file at the time it is created on the system. You can explicitly specify the character set you want to use with the CCSID parameter of the CRTSRCPF command when you create the source physical file, or you can accept the default which is *DFTCCSID. For example, to create a source physical file with CCSID 273, type:

```
CRTSRCPF FILE(MYLIB/QCBLLESRC) CCSID(273)
```

If you accept the default, then the CCSID of the job will be assigned to the source
physical file. The CCSID assigned depends on the code page being used by the
System i machine on which the source file is created.

The default CCSID for i5/OS is CCSID 65535. If the system's CCSID is 65535, then
the CCSID assigned to the source physical file is determined by the language
identifier of the job.

Including Copy Members with Different CCSIDs in Your Source File

Your ILE COBOL source program can consist of more than one source file. You can have a primary source file and multiple secondary source files such as copy books and DDS files.

The secondary source files can have CCSIDs that are different from the CCSID of the primary source file. In this case, the contents of the secondary files are converted to the CCSID of the primary source files as they are processed by the ILE COBOL compiler.

CCSID 65535 implies that no conversion of the source file is to take place. If either the primary source file, the secondary source file, or both are assigned CCSID 65535 then no conversion takes place. A syntax error could be reported by the ILE COBOL compiler if the secondary source file contains characters that are not recognized by the character set specified by the CCSID of the primary source file.

When a Format 2 COPY statement is used to incorporate DDS file descriptions into your source program, CCSID conversion does not take place. If the DDS source has a different CCSID than the source member into which it is being copied, then the copied DDS source may contain some characters which are not valid. These characters will be flagged as syntax errors.

If the primary source file and the secondary source files have different CCSIDs and neither is CCSID 65535 then the compile time performance may be impacted. The ILE COBOL compiler must spend time converting the secondary source files from one CCSID to the CCSID of the primary source file. This time may be significant depending on the size of the source files. This is illustrated in the following figure:

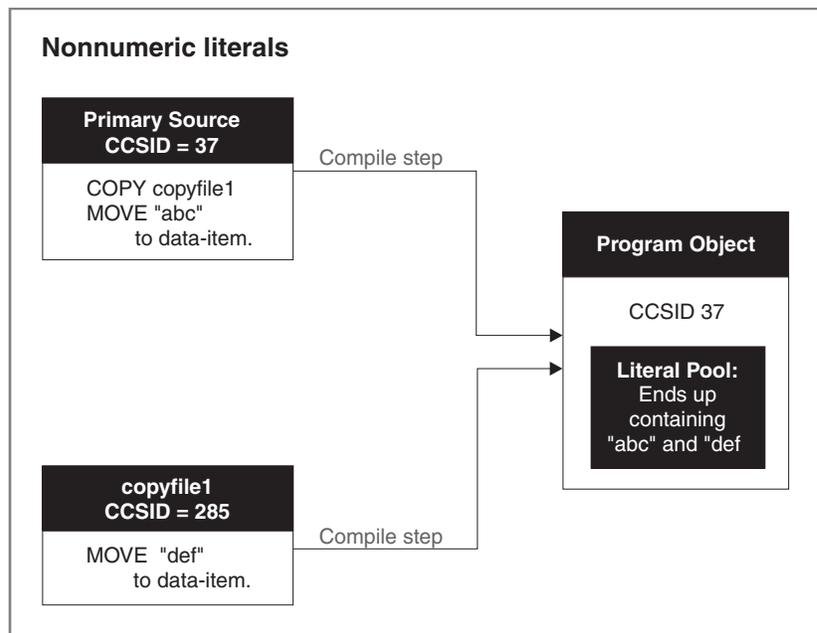


Figure 7. Converting Source Files based on CCSID

Setting the CCSID for the COBOL Syntax Checker in SEU

In order for the COBOL Syntax Checker in SEU to behave in the same manner as the ILE COBOL compiler, you must set the CCSID of the SEU job to be the same as the CCSID of the primary source file that you are editing. In most situations, they will already be the same. However, if they are different, you can change the CCSID of the job by specifying the new CCSID number in the CCSID parameter of the CHGJOB command. For example, to change the CCSID of the current job to 280, type:

```
CHGJOB CCSID(280)
```

```
# For more information on changing the attributes of a job, see the CHGJOB
# command in the CL and APIs section of the Programming category in the i5/OS
# Information Center at this Web site -http://www.ibm.com/systems/i/infocenter/.
```

Assigning a CCSID to a Locale

A CCSID is assigned to each locale when it is created on the system. Unlike a file, you have to specify a CCSID when you create the locale. You do this by specifying the CCSID parameter on the CRTLOCALE (Create Locale) command. For example, to create a locale with CCSID 273, type:

```
CRTLOCALE LOCALE('/qsys.lib/testlib.lib/en_us.locale')
SRCFILE('/qsys.lib/qsyslocale.lib/qlocalesrc.file/en_us.mbr')
CCSID(273)
```

Runtime CCSID Considerations

This section describes the runtime CCSID considerations for:

- # • i5/OS files, and their associated COBOL files
- # • i5/OS locales and their associated COBOL data items, which include
- # numeric-edited, date, and time data items.

For Locales and Files

Once you have assigned a CCSID to an i5/OS object (for example, a file or a locale), the ILE COBOL runtime checks the specified CCSID parameter of your CRTCBMOD (Create ILE COBOL Module) command to decide whether conversion is necessary. The values you can specify for the CCSID parameter are:

*JOBRUN

The runtime job's CCSID is used.

*JOB The compile job's CCSID is used.

*HEX The CCSID 65535 is used.

coded-character-set-identifier

The CCSID that you specify is used.

In the case that any of these CCSIDs is equal to 65535, no conversion is done.

When you create a locale object, you can assign a CCSID to it. For example, the locale object created in “Assigning a CCSID to a Locale” is created with a CCSID of 273. When you compile a program, you can also assign a CCSID. If the CCSID you specify at compile time is different than the CCSID you specified at the time the locale object was created, then at runtime, a conversion is made to the CCSID specified at compile time.

For Date-Time Data Items and Numeric-Edited Items

For locale objects, locales in ILE COBOL are associated with numeric-edited items and date-time items of category date and time. The following is an example of how to associate a locale with a date-time item and a numeric-edited item:

```
SPECIAL-NAMES.
  LOCALE "EN_US" IN LIBRARY "QSYSLOCALE" IS usa. 1
  :
DATA DIVISION.
WORKING-STORAGE SECTION.
01 DATE-WITH-LOCALE FORMAT DATE SIZE 10 LOCALE USA. 2
01 DATE-NO-LOCALE FORMAT DATE "@Y-%m-%d" VALUE "1997-08-09". 3
01 NUMERIC-EDITED-WITH-LOCALE PIC +$9(6).99 SIZE 15 LOCALE USA. 4
01 NUMERIC-EDITED-NO-LOCALE PIC +9(6).99 VALUE "+123456.78".
  :
PROCEDURE DIVISION.
  MOVE DATE-NO-LOCALE TO DATE-WITH-LOCALE.
  MOVE NUMERIC-EDITED-NO-LOCALE TO NUMERIC-EDITED-WITH-LOCALE.
```

```
DISPLAY "date-with-locale = " date-with-locale.  
DISPLAY "numeric-edited-with-locale = "  
    numeric-edited-with-locale.  
STOP RUN.
```

The output of the program is:

```
date-with-locale = 08/09/97  
numeric-edited-with-locale = $123,456.78
```

In the above example, line **1** defines the locale mnemonic-name `usa`, and associates that locale mnemonic-name `usa` with `EN_US` in library `QSYSLOCALE`. Although this line defines a locale object, it doesn't have to exist at compile time. However, the locale object does have to exist at runtime. For more information about creating locale objects, refer to "Creating Locales on the i5/OS" on page 197 or "Assigning a CCSID to a Locale" on page 19.

Line **2** associates the locale mnemonic-name defined in line **1** with the date data item `DATE-WITH-LOCALE`. Line **4** associates the locale mnemonic-name defined in line **1** with the numeric-edited data item `NUMERIC-EDITED-WITH-LOCALE`.

At runtime, when the data in `DATE-NO-LOCALE` is moved to `DATE-WITH-LOCALE`, the CCSID of the locale object defined in line **1** (`EN_US`) is compared to the CCSID specified at compile time. If the CCSIDs are different, then the data defined in `DATE-NO-LOCALE` (line **3**) is converted to the compile-time CCSID, and the formatted data resulting from the `MOVE` statements is based on the new CCSID.

Most statements in ILE COBOL assume the data is in CCSID 37 (in other words, single-byte EBCDIC). However some statements do support data in one or multiple CCSIDs:

- A `MOVE` statement with a receiver associated with a locale will convert the sending data to the compile time CCSID.
- A `MOVE` statement with a sender associated with a locale, or a statement that involves implicit moves, will convert the sender to CCSID 37 when:
 - A numeric-edited item is de-edited
 - A date-time item is de-edited.
- A relational condition that results in date-time comparison, non-numeric comparison, or numeric comparison.

Handling Different CCSIDs with the ILE Source Debugger

Refer to "National Language Support for the ILE Source Debugger" on page 150 for a description of how the ILE source debugger handles different CCIDs.

Chapter 3. Compiling Source Programs into Module Objects

Use WebSphere Development Studio Client for i5/OS. This is the recommended method and documentation about compiling source appears in that product's online help.

See Using the application development tools in the client product for information about getting started with the client tools.

The ILE COBOL compiler does not produce a runnable program object. It produces one or more module objects that can be bound together in various combinations to form one or more runnable units known as program objects. For more information on creating runnable program objects, refer to Chapter 4, "Creating a Program Object," on page 77.

This chapter describes:

- how to create a module object
- the CRTCBMOD command and its parameters
- how to use the PROCESS statement to specify compiler options
- how to understand the output that the ILE COBOL compiler produces.

Definition of a Module Object

Module objects are the output of all ILE compilers including the ILE COBOL compiler. They are system objects of type *MODULE. For ILE COBOL, the name of any permanently created module objects is determined by the CRTCBMOD command or the PROGRAM-ID paragraph of the outermost ILE COBOL source program. Each compilation unit in a source member creates a separate module object. The outermost ILE COBOL program in a module object can be called by another ILE COBOL program in a different module object through a bound procedure call. It can also be called using a dynamic program call after the module object has been bound into a program object. Refer to "Calling an ILE COBOL Program" on page 214 for a description of bound procedure calls and dynamic program calls.

A module object cannot be run by itself. It must first be bound into a program object. You can bind one or more module objects together to create a program object (type *PGM) or a service program (type *SRVPGM). This ability to combine module objects allows you to:

- Reuse pieces of code generally resulting in smaller programs.
- Share code between several programs therefore eliminating the chance of introducing errors to other parts of the overall program while updating a shared section.
- Mix languages to select the language that best performs the task that needs to be done.

A module object can consist of one or more ILE procedures.

The Create COBOL Module (CRTCBMOD) command creates one or more module objects from ILE COBOL source statements. These module objects remain stored in the designated library until explicitly deleted or replaced. The module objects can

then later be bound into a runnable program object using the Create Program (CRTPGM) command or into a service program using the Create Service Program (CRTSRVPGM) command. The module objects still exist in the library after the program object or service program has been created. For more information on creating a program object from one or more module objects, refer to “Using the Create Program (CRTPGM) Command” on page 79. For more information on creating a service program from one or more module objects, refer to Chapter 5, “Creating a Service Program,” on page 105.

The Create Bound COBOL Program (CRTBND CBL) command creates a program object(s) from ILE COBOL source statements in a single step. CRTBND CBL does create module objects; however, these module objects are created only temporarily and are not reusable. Once CRTBND CBL has completed creating the program object(s), the temporary module objects are deleted.

For more information on creating a program object in one step, refer to “Using the Create Bound COBOL (CRTBND CBL) Command” on page 81.

When a module object is created, it may contain the following:

- Debug data
Debug data is the data necessary for debugging a program object using the ILE source debugger. This data is generated based on the option specified in the DBGVIEW parameter of the CRTCLMOD or CRTBND CBL command.
- Program entry procedure (PEP)
A **program entry procedure** is the compiler-generated code that is the entry point for a program object on a dynamic program call. Control is passed to the PEP of a program object when it is called using a dynamic program call. It is similar to the code provided for the entry point in an OPM program. The PEP identifies the ILE procedure within a module object that is to be run first when its program object is called using a dynamic program call. When a module object is created by the ILE COBOL compiler, a PEP is generated. This PEP calls the outermost ILE COBOL program contained in the compilation unit.
When you bind multiple module objects together to create a program object, you must specify which module object will have the PEP of the program object being created. You do this by identifying the module object in the ENTMOD parameter of the CRTPGM command. The PEP of this module object then becomes the PEP for the program object. The PEPs of all other module objects are logically deleted from the program object.
- User entry procedure (UEP)
When a module object is created by the ILE COBOL compiler, the outermost ILE COBOL program contained in the compilation unit is the **user entry procedure**. During a dynamic program call, the UEP is the ILE procedure that gets control from the PEP. During a static procedure call between ILE procedures in different module objects, the UEP is given control directly.

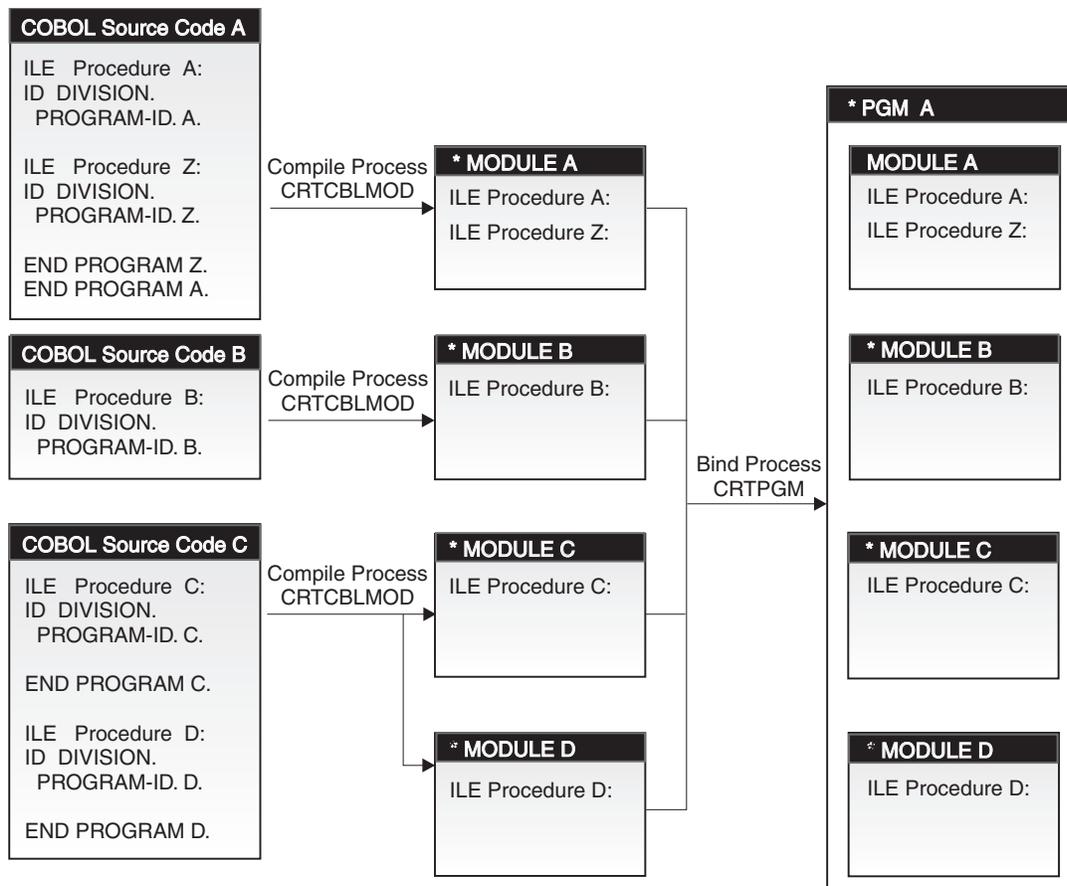


Figure 8. Creating Module Objects Using the CRTCBMOD Command

In Figure 8, *PGM A is created with *MODULE A designated the module object having the entry point for the program object. The PEP for *MODULE A calls ILE Procedure A. The PEP for *MODULE A also becomes the PEP for *PGM A so the PEP for *PGM A calls ILE Procedure A. The UEP for *PGM A is also ILE Procedure A. *MODULE B, *MODULE C, and *MODULE D also have PEPs but they are ignored by *PGM A. Also, ILE Procedure Z can only be called from ILE Procedure A. ILE Procedure Z is not visible to ILE Procedures B, C, and D as they are in separate module objects and ILE Procedure Z is not the outermost COBOL program in *MODULE A. ILE Procedures A, B, C, and D can call each other. Recursion is not allowed because all of them are non recursive procedures.

Each declarative procedure in an ILE COBOL source program generates a separate ILE procedure.

Each nested COBOL program generates a separate ILE procedure.

A module object can have module exports and module imports associated with it.

A **module export** is the name of a procedure or data item that is available for use by other ILE objects through the binding process. The module export is identified by its name and its associated type, either procedure or data. Module exports can be scoped in two ways: to the program object and to the activation group. Not all names that are exported to the program object are exported to the activation group. The ILE COBOL compiler creates module exports for each of the following COBOL programming language constructs:

- A procedure name corresponding to the outermost ILE COBOL program in a compilation unit.
- A cancel procedure name corresponding to the outermost ILE COBOL program in a compilation unit.
- A weak export of an EXTERNAL file or EXTERNAL data.

A **module import** is the use of or reference to the name of a procedure or data item not defined in a referencing module object. The module import is identified by its name and its associated type, either procedure or data. The ILE COBOL compiler creates module imports for each of the following COBOL programming language constructs:

- A procedure name corresponding to an ILE COBOL program that is called using a static procedure call.
- A cancel procedure name corresponding to an ILE COBOL program that is called using a static procedure call.
- A weak import of an EXTERNAL file or EXTERNAL data.
- A procedure name corresponding to an ILE COBOL program that is set by the SET *procedure-pointer-item* TO ENTRY *procedure-name* statement where the name of *procedure-name* is to be interpreted as an ILE procedure.

The module import is generated when the target procedure is not defined in the referencing module object. A weak import to data item is generated when the data item is referenced in the ILE COBOL program.

Using the Create COBOL Module (CRTCBMOD) Command

```
# To compile ILE COBOL source statements into one or more module objects, you
# must use the CRTCBMOD command. This command starts the ILE COBOL
# compiler that creates the module object(s) based on your ILE COBOL statements in
# the source member. You can use the CRTCBMOD command interactively, or in
# batch mode, or from a CL program on the i5/OS.
```

```
# Note: In order to create a module object with the CRTCBMOD command you
# must have authority to use the command.
```

```
# If the Format 2 COPY statement is used in the program to access externally
described files, the operating system provides information about the externally
described files to the compiled program.
```

If the ILE COBOL compiler stops, the message LNC9001
Compile failed. *module-name* not created.

You can use a control language program that can monitor for this exception by using the Monitor Message (MONMSG) CL command.

Using Prompt Displays with the CRTCBMOD Command

The CRTCBMOD command can be entered using prompt display screens. To enter command parameters in this manner, type CRTCBMOD and press F4.

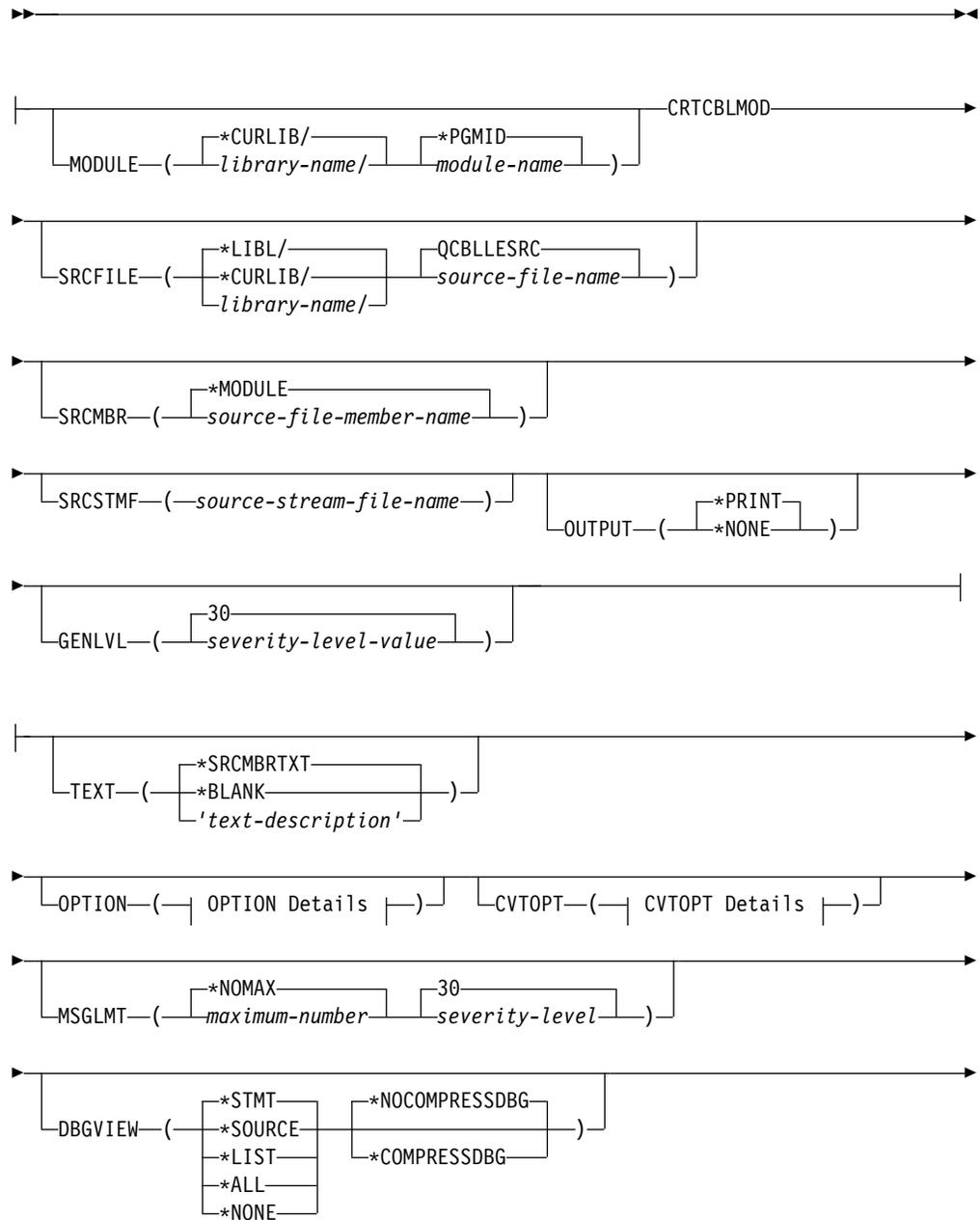
Each parameter on this display shows a default value. Type over any items to set different values or options. If you are unsure about the setting of a parameter value, type a question mark (?) in the first position of the field and press Enter, or F4 (Prompt), to receive more detailed information. The question mark must be

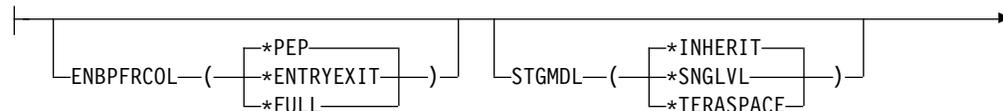
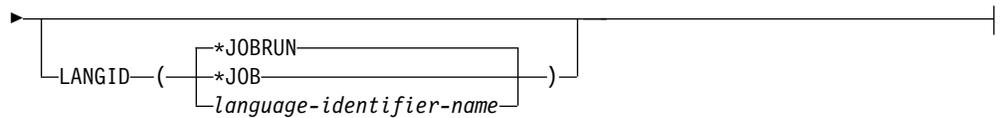
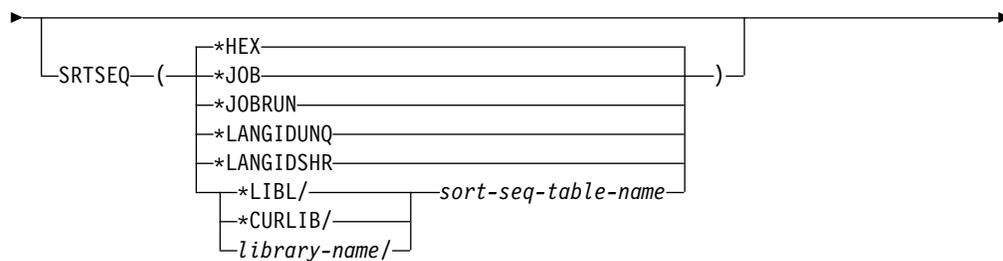
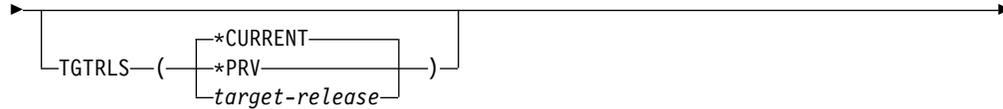
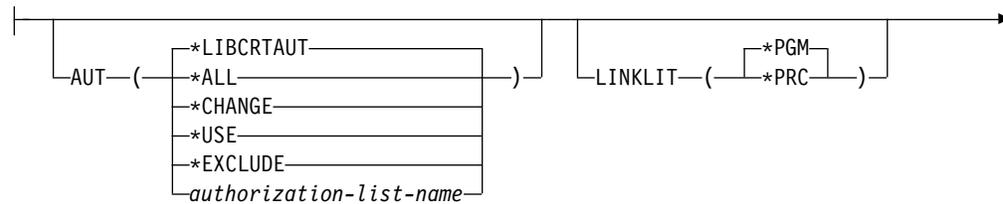
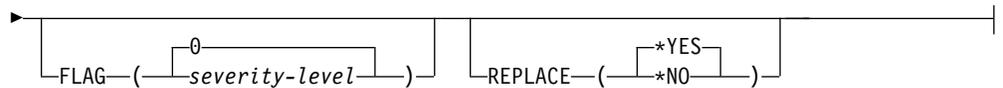
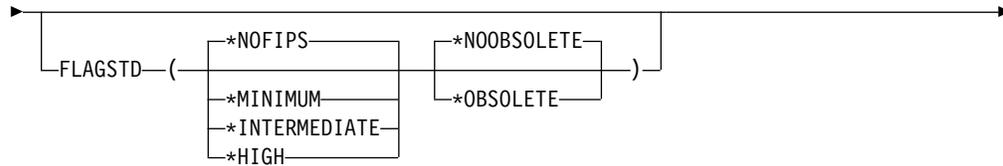
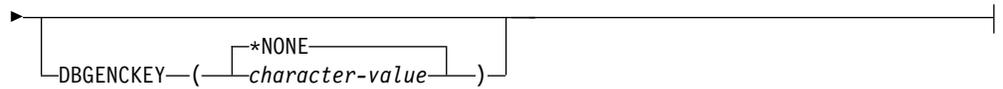
followed by a blank. If you entered some of the parameters before requesting the prompt display screen, the values that you supplied are displayed for the parameter.

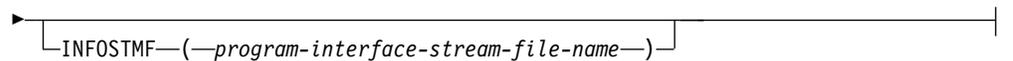
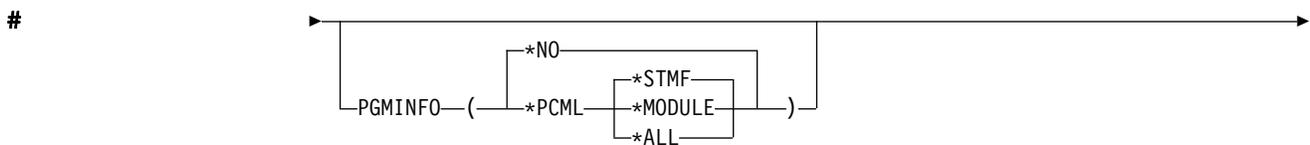
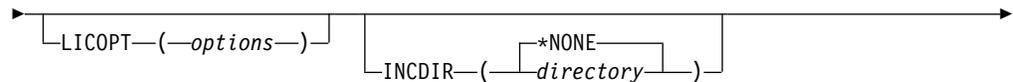
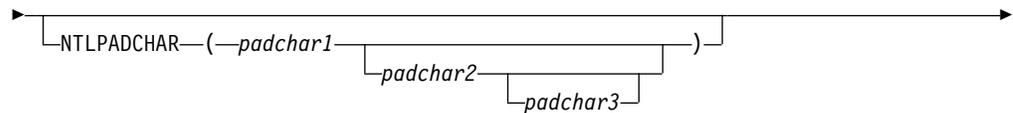
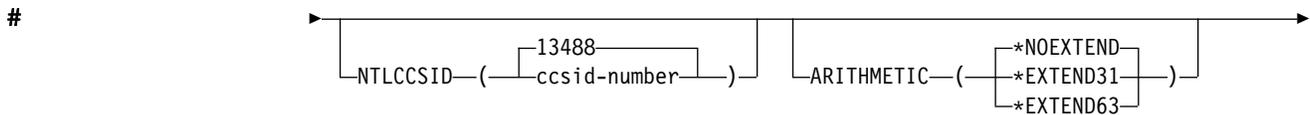
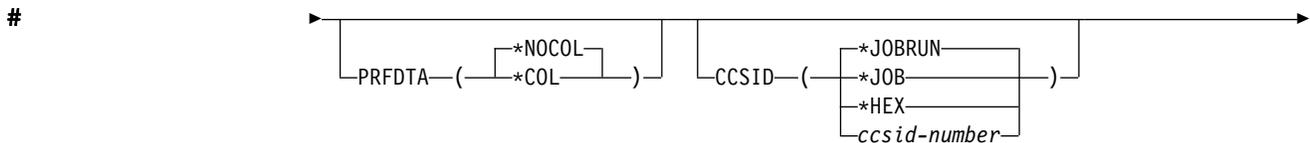
For a description of the parameters for the CRTCBMOD command refer to "Parameters of the CRTCBMOD Command" on page 28.

Syntax for the CRTCBMOD Command

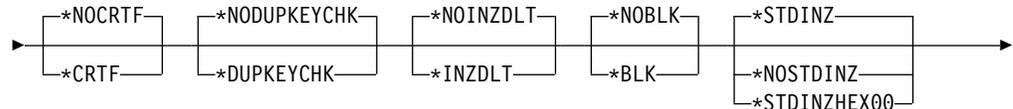
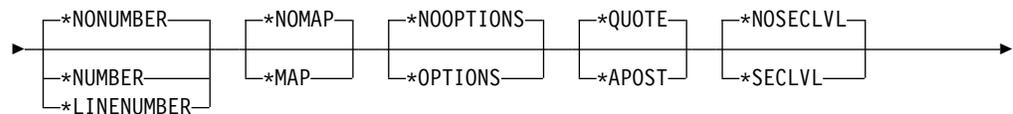
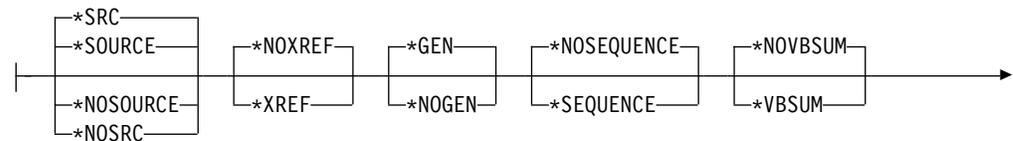
CRTCBMOD Command—Format

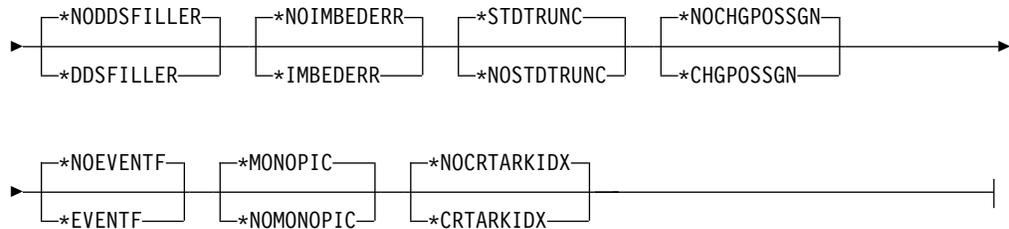




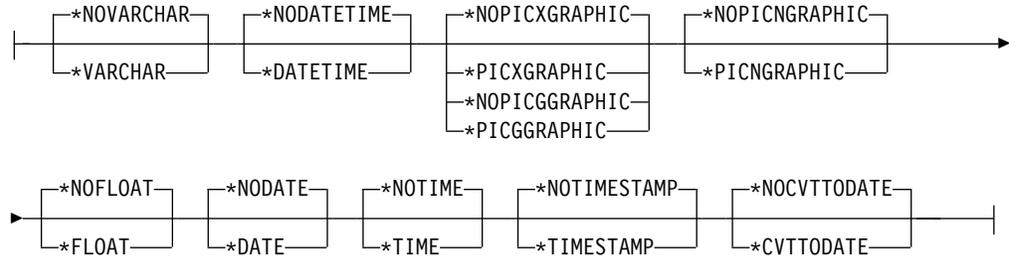


OPTION Details:





CVTOPT Details:



Parameters of the CRTCLMOD Command

A description of the parameters for the CRTCLMOD command are defined in this section. The parameters and options are described in the order they appear on the prompt displays.

The default values are displayed first, and are underscored for identification.

```
# All object names specified for the CRTCLMOD command must follow i5/OS
# naming conventions: the names may be basic names, ten characters in length,
# composed of alphanumeric characters, the first of which must be alphabetic; or the
# names may be quoted names, eight characters in length, enclosed in double quotes.
```

You can specify various compiler options by using the OPTION parameter of the CRTCLMOD command or from within the source program using the PROCESS statement. Any options specified in the PROCESS statement override the corresponding options on the CRTCLMOD command.

MODULE Parameter:

Specifies the module name and library name for the module object you are creating. The module name and library name must conform to i5/OS naming conventions. The possible values are:

***PGMID**

The name for the module is taken from the PROGRAM-ID paragraph in the outermost ILE COBOL source program of the compilation unit.

module-name

Enter a name to identify the compiled ILE COBOL module. If you specify a module name for this parameter, and compile a *sequence of source programs* (multiple compilation units in a single source file member) the first module in the sequence uses this name; any other modules use the name specified in the PROGRAM-ID paragraph in the corresponding outermost ILE COBOL source program of the compilation unit.

The possible library values are:

***CURLIB**

The created module object is stored in the current library. If you have not assigned a library as the current library, QGPL is used.

library-name

Enter the name of the library where the created module object is to be stored.

SRCFILE Parameter:

Specifies the name of the source file and library that contains the ILE COBOL source code to be compiled. This source file should have a record length of 92. The possible values are:

QCBLLSRC

Specifies that the source file, QCBLLSRC, contains the ILE COBOL source code to be compiled.

source-file-name

Enter the name of the source file that contains the ILE COBOL source code to be compiled.

The possible library values are:

***LIBL**

The library list is searched to find the library where the source file is located.

***CURLIB**

The current library is used. If you have not assigned a library as the current library, QGPL is used.

library-name

Enter the name of the library where the source file is located.

SRCMBR Parameter:

Specifies the name of the member that contains the ILE COBOL source code to be compiled. You can specify this parameter only if the source file referred to in the SRCFILE parameter is a database file. The possible values are:

***MODULE**

The source file member with the same name as the module name specified on the MODULE parameter, is used.

If you do not specify a module name for the MODULE parameter, the first member of the database source file is used.

source-file-member-name

Enter the name of the member that contains the ILE COBOL source code.

SRCSTMF Parameter:

Specifies the path name of the stream file containing the ILE COBOL source code to be compiled. The path name can be either absolutely or relatively qualified. An absolute path name starts with '/'; a relative path name starts with a character other than '/'. If absolutely-qualified, the path name is complete. If relatively-qualified, the path name is completed by appending the job's current working directory to the path name. The SRCMBR and SRCFILE parameters cannot be specified with the SRCSTMF parameter.

OUTPUT Parameter:

Specifies if the compiler listing is generated or not. The possible values are:

***PRINT**

A compiler listing is generated. If a member is being compiled, the output file has the same name as the member. If a stream file is being compiled and *PGMID is specified in the PGM parameter, the output file has the name COBOLPGM00. Otherwise, it has the same name as the program.

***NONE**

No compiler listing is generated.

GENLVL Parameter:

Specifies the severity level that determines if a module object is created. The severity level corresponds to the severity level of the messages produced during compilation. This parameter applies individually to each compilation unit in a source file member. Other compilation units in the source file member will still be compiled even if a previous compilation unit fails.

The possible values are:

30 No module object is created if errors occur with a severity level equal to or greater than 30.

severity-level

Specify a one or two-digit number, 0 through 30, which is the severity level you want to use to determine if a module object is to be created. No module object is created if errors occur with a severity level equal to or greater than this severity level.

TEXT Parameter:

Allows you to enter text that briefly describes the module and its function.

***SRCMBRTXT**

The same text that describes the database file member containing the ILE COBOL source code, is used to describe the module object. If the source comes from a device or inline file, specifying *SRCMBRTXT has the same effect as specifying *BLANK.

***BLANK**

No text is specified.

text-description

Enter text briefly describing the module and its function. The text can be a maximum of 50 SBCS characters in length and must be enclosed in single quotation marks. The single quotation marks are not part of the 50-character string.

OPTION Parameter:

Specifies the options to use when the ILE COBOL source code is compiled.

Options specified in the PROCESS statement of an ILE COBOL source program override the corresponding options of the OPTION parameter.

The possible values of the OPTION parameter are:

***SOURCE or *SRC**

The compiler produces a source listing, consisting of the ILE COBOL source program and all compile-time error messages.

***NOSOURCE or *NOSRC**

The compiler does not produce the source part of the listing. If you do not require a source listing, you should use this option because compilation may take less time.

***NOXREF**

The compiler does not produce a cross-reference listing for the ILE COBOL source program.

***XREF**

The compiler produces a cross-reference listing for the source program.

***GEN**

The compiler creates a module object after the ILE COBOL source is compiled.

***NOGEN**

The compiler does not create a module object after the ILE COBOL source program is compiled. You might specify this option if you want only error messages or listings.

***NOSEQUENCE**

The reference numbers are not checked for sequence errors.

***SEQUENCE**

The reference numbers are checked for sequence errors. Sequence errors do not occur if the *LINENUMBER option is specified.

***NOVBSUM**

Verb usage counts are not printed.

***VBSUM**

Verb usage counts are printed.

***NONUMBER**

The source-file sequence numbers are used for reference numbers.

***NUMBER**

The user-supplied sequence numbers (columns 1 through 6) are used for reference numbers.

***LINENUMBER**

The sequence numbers created by the compiler are used for reference numbers. This option combines ILE COBOL program source code and source code introduced by COPY statements into one consecutively numbered sequence. Use this option if you specify FIPS (Federal Information Processing Standards) flagging.

***NOMAP**

The compiler does not list the Data Division map.

***MAP**

The compiler lists the Data Division map.

***NOOPTIONS**

Options in effect are not listed for this compilation.

***OPTIONS**

Options in effect are listed for this compilation.

***QUOTE**

Specifies that the delimiter quotation mark (") is used for nonnumeric literals, hexadecimal literals, and Boolean literals. This option also specifies that the value of the figurative constant QUOTE has the EBCDIC value of a quotation mark.

***APOST**

Specifies that the delimiter apostrophe (') is used for nonnumeric literals,

hexadecimal literals, and Boolean literals. This option also specifies that the value of the figurative constant QUOTE has the EBCDIC value of an apostrophe.

***NOSECLVL**

Second level message text is not listed for this compilation.

***SECLVL**

Second level message text is listed for this compilation, along with the first-level error text, in the message section of the compiler listing.

***PRTCORR**

Comment lines are inserted in the compiler listing indicating which elementary items were included as a result of the use of the CORRESPONDING phrase.

***NOPRTCORR**

Comment lines are not inserted in the compiler listing when the CORRESPONDING phrase is used.

***MONOPRC**

The program-name (literal or word) found in the PROGRAM-ID paragraph, the CALL, CANCEL, or SET ENTRY statements, and the END PROGRAM header is converted to all uppercase characters (monocasing) and the rules for program-name formation are enforced.

***NOMONOPRC**

The program-name (literal or word) found in the PROGRAM-ID paragraph, the CALL, CANCEL, or SET ENTRY statements, and the END PROGRAM header is not converted to all uppercase characters (no monocasing) and the rules for program-name formation are not enforced. This option allows special characters not allowed for standard COBOL to be used in the CALL target.

***RANGE**

At run time, subscripts are verified to ensure they are within the correct ranges, but index ranges are not verified. Reference modification and compiler-generated substring operations are also checked.

The contents of date-time items are checked to make sure their format is correct, and that they represent a valid date, time, or timestamp.

***NORANGE**

Ranges are not verified at run time.

Note: The *RANGE option generates code for checking subscript ranges. For example, it ensures that you are not attempting to access the element 21 of a 20-element array.

The *NORANGE option does not generate code to check subscript ranges. As a result, the *NORANGE option produces faster running code.

***NOUNREF**

Unreferenced data items are not included in the compiled module. This reduces the amount of storage used, allowing a larger program to be compiled. You cannot look at or assign to an unreferenced data item during debugging when the *NOUNREF option is chosen. The unreferenced data items still appear in the cross-reference listings produced by specifying OPTION (*XREF).

***UNREF**

Unreferenced data items are included in the compiled module.

***NOSYNC**

The SYNCHRONIZED clause is syntax checked only.

***SYNC**

The SYNCHRONIZED clause is compiled by the compiler. The SYNCHRONIZED clause causes the position of a data item to be aligned such that the right-hand (least-significant) end is on the natural storage boundary. The natural storage boundary is the next nearest 4-byte, 8-byte, or 16-byte boundary in storage depending on the length and type of data being stored. Extra storage is reserved adjacent to the synchronized item to achieve this alignment. Each elementary data item that is described as SYNCHRONIZED is aligned to the natural storage boundary that corresponds to its data storage assignment.

***NOCRTF**

Disk files that are unavailable at the time of an OPEN operation are not created dynamically.

***CRTF**

Disk files that are unavailable at the time of an OPEN operation are created dynamically.

Note: The maximum record length for a file that will be created dynamically is 32 766. Indexed files will not be dynamically created even though the *CRTF option has been specified.

***NODUPKEYCHK**

Does not check for duplicate primary keys for INDEXED files.

***DUPKEYCHK**

Checks for duplicate primary keys for INDEXED files.

***NOINZDLT**

Relative files with sequential access are not initialized with deleted records during the CLOSE operation if the files have been opened for OUTPUT. The record boundary is determined by the number of records written at OPEN OUTPUT time. Subsequent OPEN operations allow access only up to the record boundary.

***INZDLT**

Relative files with sequential access are initialized with deleted records during the CLOSE operation if the files were opened for OUTPUT. Active records in the files are not affected. The record boundary is defined as the file size for subsequent OPEN operations.

***NOBLK**

The compiler allows blocking only of SEQUENTIAL access files with no START statement. The BLOCK CONTAINS clause, if specified, is ignored, except for tape files.

***BLK**

When *BLK is used, the compiler allows blocking for DYNAMIC access files and SEQUENTIAL access files. Blocking is not allowed for RELATIVE files opened for output operations. The BLOCK CONTAINS clause determines the number of records to be blocked if it is specified, otherwise the operating system determines the number of records to be blocked.

***STDINZ**

For those items with no VALUE clause, the compiler initializes data items to default values. The value assigned to each area of storage of the first level-01 or level-77 data item that occupies the area.

***NOSTDINZ**

For those items with no VALUE clause, the compiler does not initialize data items to system defaults.

***STDINZHEX00**

For those items with no VALUE clause, the compiler initializes data items to hexadecimal zero.

***NODDSFILLER**

If no matching fields are found by a COPY DDS statement, no field descriptions are generated.

***DDSFILLER**

If no matching fields are found by a COPY DDS statement, a single character FILLER field description, "07 FILLER PIC X", is always created.

***NOIMBEDERR**

Error messages are not included in the source listing section of the compiler listing. Error messages only appear in the error message section of the compiler listing.

***IMBEDERR**

First level error messages are included in the source listing section of the compiler listing, immediately following the line where the error occurred. Error messages also appear in the error message section of the compiler listing.

***STDTRUNC**

This option applies only to USAGE BINARY data. When *STDTRUNC is selected, USAGE BINARY data is truncated to the number of digits in the PICTURE clause of the BINARY receiving field.

***NOSTDTRUNC**

This option applies only to USAGE BINARY data. When *NOSTDTRUNC is selected, BINARY receiving fields are truncated only at half-word, full-word, or double-word boundaries. BINARY sending fields are also handled as half-words, full-words, or double-words. Thus, the full binary content of the field is significant. Also, the DISPLAY statement will convert the entire content of a BINARY field, with no truncation.

Note: *NOSTDTRUNC has no effect on the VALUE clause.

***NOCHGPOSSGN**

Hexadecimal F is used as the default positive sign for zoned and packed numeric data. Hexadecimal F is the system default for the operating system.

***CHGPOSSGN**

Hexadecimal C is used as the default positive sign for zoned and packed numeric data. This applies to all results of the MOVE, ADD, SUBTRACT, MULTIPLY, DIVIDE, COMPUTE, and INITIALIZE statements, as well as the results of the VALUE clause.

***NOEVENTF**

Do not create an Event File for use by CoOperative Development Environment/400 (the client product). The client product uses this file to

provide error feedback integrated with the client product editor. An Event File is normally created when you create a module or program from within the client product.

***EVENTF**

Create an Event File for use by the client product. 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.

The client product uses this file to provide error feedback integrated with the client product editor. An Event File is normally created when you create a module or program from within the client product.

***MONOPIC**

All alphabetic characters in a PICTURE character-string will be converted to uppercase (monocasing).

***NOMONOPIC**

The currency symbol used in the PICTURE character-string is case sensitive. That is, the lowercase letters corresponding to the uppercase letters for the PICTURE symbols A, B, E, G, N, P, S, V, X, Z, CR, and DB are equivalent to their uppercase representations in a PICTURE character-string. All other lowercase letters are not equivalent to their corresponding uppercase representations.

***NOCRTARKIDX**

Temporary alternate record key (ARK) indexes are not created if permanent ones cannot be found.

***CRTARKIDX**

Temporary alternate record key (ARK) indexes are created if permanent ones cannot be found.

CVTOPT Parameter:

Specifies how the compiler handles date, time, and timestamp field types, DBCS-graphic field type, variable-length field types, and floating-point field types passed from externally-described files to your program through COPY DDS. The possible values are:

***NOVARCHAR**

Variable-length fields are declared as FILLER fields.

***VARCHAR**

Variable-length fields are declared as group items, and are accessible to the ILE COBOL source program.

***NODATETIME**

Date, time, and timestamp data items are declared as FILLER fields.

***DATETIME**

Date, time, and timestamp DDS data items are given COBOL data item names based on their DDS names. The category of the COBOL data item is alphanumeric, unless one of the CVTOPT parameter values *DATE, *TIME, or *TIMESTAMP is specified. In this case, the category of the COBOL data item is date, time, or timestamp, respectively.

***NOPIXGRAPHIC**

DBCS-graphic data items are declared as FILLER fields.

***PICXGRAPHIC**

Fixed-length DBCS-graphic data items are declared as fixed-length alphanumeric fields, and are accessible to the ILE COBOL source program.

When the *VARCHAR option is also in use, variable-length DBCS-graphic data items are declared as fixed-length group items, and are accessible to the ILE COBOL source program.

***PICGGRAPHIC**

Fixed-length DBCS-graphic data items are declared as fixed-length G fields, and are accessible to the ILE COBOL source program.

When the *VARCHAR option is also in use, variable-length DBCS-graphic data items are declared as fixed-length group items (made of a numeric field followed by G type field), and are accessible to the ILE COBOL source program.

***NOPICGGRAPHIC**

DBCS-graphic data items are declared as FILLER fields.

*NOPICGGRAPHIC will be printed as *NOPICXGRAPHIC in the listing.

***PICNGRAPHIC**

Fixed-length graphic data items, associated with the CCSID specified in the National CCSID compiler option or in the NTLCCSID PROCESS option, are declared as fixed-length N fields, and are accessible to the ILE COBOL source program.

When the *VARCHAR option is also in use, variable-length graphic data items with the CCSID specified in the National CCSID compiler option or in the NTLCCSID PROCESS option are declared as fixed-length group items (made of a numeric field followed by N type field), and are accessible to the ILE COBOL source program.

***NOPICNGRAPHIC**

The processing of graphic fields depends upon the values specified for the PICXGRAPHIC/NOPICXGRAPHIC and PICGGRAPHIC/NOPICGGRAPHIC options.

***NOFLOAT**

Floating-point data items are declared as FILLER fields with a USAGE of binary.

***FLOAT**

Floating-point data items are brought into the program with their DDS names and a USAGE of COMP-1 (single-precision) or COMP-2 (double-precision). The fields are made accessible to the ILE COBOL source program.

***NODATE**

Date data items are declared as category alphanumeric COBOL data items, for example:

```
06 FILLER PIC X(10).
```

The COBOL data item name is determined by the *NODATETIME/*DATETIME CVTOPT parameter.

***DATE**

DDS date data items are declared as category date COBOL data items, for example:

```
06 FILLER FORMAT DATE '@Y-%m-%d'.
```


#

The COBOL data item name is determined by the *NODATETIME/
*DATETIME CVTOPT parameter.

***NOTIME**

DDS time data items are declared as category alphanumeric COBOL data items, for example:

```
06 FILLER PIC X(8).
```

The COBOL data item name is determined by the *NODATETIME/
*DATETIME CVTOPT parameter.

***TIME**

DDS time data items are declared as category time COBOL data items, for example:

```
06 FILLER FORMAT TIME '%H:%M:%S'.
```

The COBOL data item name is determined by the *NODATETIME/
*DATETIME CVTOPT parameter.

***NOTIMESTAMP**

DDS timestamp data items are declared as category alphanumeric COBOL data items, for example:

```
06 FILLER PIC X(26).
```

The COBOL data item name is determined by the *NODATETIME/
*DATETIME CVTOPT parameter.

***TIMESTAMP**

DDS timestamp data items are declared as category timestamp COBOL data items, for example:

```
06 FILLER FORMAT TIMESTAMP.
```

The COBOL data item name is determined by the *NODATETIME/
*DATETIME CVTOPT parameter.

***NOCVTTODATE**

DDS data items with the DATFMT keyword (excluding DDS date data items) are declared in ILE COBOL based on their original DDS type.

***CVTTODATE**

DDS data items with the DATFMT keyword (excluding DDS date data items) are declared in ILE COBOL as date data items. For more information about using the *CVTTODATE option, refer to “Specifying Date, Time, and Timestamp Data Types” on page 50.

MSGLMT Parameter:

Specifies the maximum number of messages of a given error severity level that can occur for each compilation unit before compilation stops. As soon as one compilation unit reaches the maximum, compilation stops for the entire source member.

For example, if you specify 3 for the maximum number of messages and 20 for the error severity level then compilation will stop if three or more errors with a severity level of 20 or higher occur. If no messages equal or exceed the given error severity level, compilation continues regardless of the number of errors encountered.

number-of-messages

Specifies the maximum number of messages. The possible values are:

***NOMAX**

Compilation continues until normal completion regardless of the number of errors encountered.

maximum-number

Specifies the maximum number of messages that can occur at or above the specified error severity level before compilation stops. The valid range is 0-9999.

message-limit-severity

Specifies the error severity level used to determine whether or not to stop compilation. The possible values are:

30 Compilation stops if the number of errors with severity level 30 or higher exceeds the maximum number of messages specified.

error-severity-level

Enter a one or two-digit number, 0 through 30, which is the error severity level you want to use to determine whether or not to stop compilation. Compilation stops if the number of errors with this severity level or higher exceeds the maximum number of messages you specified.

DBGVIEW Parameter:

Specifies options that control which views of the source program or generated listing are available for debugging the compiled module, and if the debug listing view is compressed or not.

debug-view

Specify the views to be available for debugging. The possible values are:

***STMT**

The compiled module can be debugged using symbolic names and statement numbers.

***SOURCE**

The primary source member, as well as copied source members which were included through COPY statements, will have source views available for debugging the compiled module. These views are available only if the primary source member and copied source members come from local database source files. Do not change or delete members during the time between compile and debug.

***LIST**

A listing view, which shows the source code after the processing of any COPY and REPLACE statements, will be made available for debugging the compiled module. This option increases the size of the compiled module, without affecting the runtime performance of the compiled module.

The listing view will include the cross-reference listing, Data Division map, and verb usage counts when the corresponding compiler options are requested. For example, a cross-reference listing will be included if OPTION(*XREF) is specified.

Listing views can be generated regardless of where the primary source members or copied source members come from. Listing views are not affected by changes to or deletion of the source members following the compilation.

***ALL**

Equivalent to specifying *STMT, *SOURCE, and *LIST combined.

***NONE**

The compiled module cannot be debugged. This reduces the size of the compiled program, but does not affect its runtime performance. When this option is specified, a formatted dump can not be taken.

compress-listing-view

Specifies if the listing view is compressed or not when *LIST or *ALL is specified in debug-view. The possible values are:

***NOCOMPRESSDBG**

The listing view is not compressed.

***COMPRESSDBG**

The listing view is compressed when *LIST or *ALL is specified in debug-view. By using this option, some but not all large COBOL programs will be able to compile with the *LIST debug view option.

DBGENCKEY Parameter:

Specifies the encryption key to be used to encrypt program source that is embedded in debug views.

***NONE**

No encryption key is specified.

character-value

Specify the key to be used to encrypt program source that is embedded in debug views stored in the module object. The length of the key can be between 1 and 16 bytes. A key of length 1 to 15 bytes will be padded to 16 bytes with blanks for the encryption. Specifying a key of length zero is the same as specifying *NONE.

If the key contains any characters which are not invariant over all code pages, it will be up to the user to ensure that the target system uses the same code page as the source system, otherwise the key may not match and the decryption may fail. If the encryption key must be entered on systems with differing code pages, it is recommended that the key be made of characters which are invariant for all EBCDIC code pages.

OPTIMIZE Parameter:

Specifies the level of optimization of the module. The possible values are:

***NONE**

No optimization is performed on the compiled module. Compilation time is minimized when this option is used. This option allows variables to be displayed and changed during debugging.

***BASIC**

Some optimization (only at the local block level) is performed on the compiled module. This option allows user variables to be displayed but not changed during debugging.

***FULL**

Full optimization (at the global level) is performed on the compiled module. This optimization increases compilation time but also generates the most efficient code. This option allows user variables to be displayed but not changed during debugging. The displayed values of the variables may not be their current values. Some variables may not be displayable.

***NEVER**

This option has the same effect as *NONE except the module's optimization level cannot be changed at a later time with the CHGMOD

| command. This option does not generate any optimization information.
| This enables much larger programs to be compiled without exceeding
| system storage limits.

| **Note:** The user can change the optimization level of the module object using
| the CHGMOD, CHGPGM, or CHGSRVPGM command without having
| to recompile the source program, unless the *NEVER option value was
| selected.

FLAGSTD Parameter:

Specifies the options for FIPS flagging. (Select the *LINENUMBER option to ensure that the reference numbers used in the FIPS messages are unique.) The possible values are:

***NOFIPS**

The ILE COBOL source program is not FIPS flagged.

***MINIMUM**

FIPS flag for minimum subset and higher.

***INTERMEDIATE**

FIPS flag for intermediate subset and higher.

***HIGH**

FIPS flag for high subset.

***NOOBSOLETE**

Obsolete language elements are not flagged.

***OBSOLETE**

Obsolete language elements are flagged.

EXTDSPOPT Parameter:

Specifies the options to use for extended ACCEPT and extended DISPLAY statements for workstation I/O. The possible values are:

***DFRWR**

Extended DISPLAY statements are held in a buffer until an extended ACCEPT statement is encountered, or until the buffer is filled.

The contents of the buffer are written to the display when the extended ACCEPT statement is encountered or the buffer is full.

***NODFRWR**

Each extended DISPLAY statement is performed as it is encountered.

***UNDSPCHR**

Displayable and undisplayable characters are handled by extended ACCEPT and extended DISPLAY statements.

***NOUNDSPCHR**

Only displayable characters are handled by extended ACCEPT and extended DISPLAY statements.

Although you must use this option for display stations attached to remote 3174 and 3274 controllers, you can also use it for local workstations. If you do use this option, your data must contain displayable characters only. If the data contains values less than hexadecimal 20, the results are not predictable, ranging from unexpected display formats to severe errors.

***ACCPDALL**

All types of data are predisplayed in the extended ACCEPT statements regardless of the existence of the UPDATE phrase.

***ACCUPDNE**

Only numeric edited data are predisplayed in the extended ACCEPT statements that do not contain the UPDATE phrase.

FLAG Parameter:

Specifies the minimum severity level of messages that will appear in the compiler listing. The possible values are:

0 All messages will appear in the compiler listing.

severity-level

Enter a one or two-digit number that specifies the minimum severity level of messages that you want to appear in the compiler listing. Messages that have severity levels of this specified value or higher will appear in the compiler listing.

REPLACE Parameter:

Specifies if a new module is created when a module of the same name in the specified or implied library already exists. The possible values are:

*YES

A new module is created and it replaces any existing module of the same name in the specified or implied library. The existing module of the same name in the specified or implied library is moved to library QRPLOBJ.

***NO**

A new module is not created if a module of the same name already exists in the specified or implied library. The existing module is not replaced, a message is displayed, and compilation stops.

AUT Parameter:

Specifies the authority given to users who do not have specific authority to the module object, who are not on the authorization list, or whose group has no specific authority to the module object. You can change the authority for all users, or for specific users after the module object is created by using the GRTOBJAUT (Grant Object Authority) or RVKOBJAUT (Revoke Object Authority) commands.

The possible values are:

*LIBCRTAUT

The public authority for the object is taken from the CRTAUT keyword of the target library (the library that is to contain the created module object). This value is determined when the module object is created. If the CRTAUT value for the library changes after the module object is created, the new value does NOT affect any existing objects.

***ALL**

Provides authority for all operations on the module object except those limited to the owner or controlled by authorization list management authority. The user can control the module object's existence, specify security for it, change it, and perform basic functions on it, but cannot transfer its ownership.

***CHANGE**

Provides all data authority and the authority for performing all operations on the module object except those limited to the owner or controlled by object authority and object management authority. The user can change the object and perform basic functions on it.

***USE**

Provides object operational authority and read authority; authority for

basic operations on the module object. The user can perform basic operations on the object but is prevented from changing the object.

***EXCLUDE**

The user cannot access the module object.

authorization-list-name

The name of an authorization list of users and authorities to which the module is added. The module object is secured by this authorization list, and the public authority for the module object is set to *AUTL. The authorization list must exist on the system when the CRTCBMOD command is issued. Use the Create Authorization List (CRTAUTL) command to create your own authorization list.

LINKLIT Parameter:

Specifies the linkage type for external CALL/CANCEL 'literal' target and the SET ENTRY target. You may override this option for specific external CALL/CANCEL 'literal' target and the SET ENTRY target lists by specifying the following sentence in the SPECIAL-NAMES paragraph:

```
LINKAGE TYPE IS implementer-name FOR target-list.
```

The possible values for LINKLIT are:

***PGM**

Target for CALL/CANCEL or SET ENTRY is a program object.

***PRC**

Target for CALL/CANCEL or SET ENTRY is an ILE procedure.

TGTRLS Parameter:

Specifies the release of the operating system on which you intend to use the object being created. In the examples given for the *CURRENT and *PRV values, and when specifying the *target-release* value, the format VxRxMx is used to specify the release, where Vx is the version, Rx is the release, and Mx is the modification level. For example, V2R3M0 is version 2, release 3, modification level 0.

Valid values for this parameter change every release. The possible values are:

***CURRENT**

The object is to be used on the release of the operating system currently running on the system. For example, if V2R3M5 is running on the system, *CURRENT means that you intend to use the object on a system with V2R3M5 installed. The object can also be used on a system with any subsequent release of the operating system installed.

Note: If V2R3M5 is running on the system, and the object is to be used on a system with V2R3M0 installed, specify TGTRLS(V2R3M0), not TGTRLS(*CURRENT).

***PRV**

The object is to be used on the previous release with modification level 0 of the operating system. For example, if V2R3M5 is running on the system, *PRV means that 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. The object can be used on a system with the specified release or with any subsequent release of the operating system installed.

Valid values depend on the current version, release, and modification level, and they change with each new release. If you specify a *target-release* that is earlier than the earliest release level supported by this command, an error message is sent indicating the earliest supported release.

Note: The current version of the command may support options that are not available in previous releases of the command. If the command is used to create objects that are to be used on a previous release, it will be processed by the compiler appropriate to that release, and any unsupported options will not be recognized. The compiler will not necessarily issue any warnings regarding options that it is unable to process.

SRTSEQ Parameter:

Specifies the sort sequence used when NLSSORT is associated with an alphabet-name in the ALPHABET clause. The SRTSEQ parameter is used in conjunction with the LANGID parameter to determine which system-defined or user-defined sort sequence table the module will use. The possible values are:

***HEX**

No sort sequence table will be used, and the hexadecimal values of the characters will be used to determine the sort sequence.

***JOB**

The sort sequence will be resolved and associated with the module at compile time using the sort sequence of the compile job. The sort sequence table of the compile job must exist in the system at compile time. If at run time, the CCSID of the runtime job differs from the CCSID of the compile time job, the sort sequence table loaded at compile time is converted to match the CCSID of the runtime job.

***JOBRUN**

The sort sequence of the module will be resolved and associated with the module at run time. This value allows a module to be compiled once and used with different sort sequences at run time.

***LANGIDUNQ**

Specifies that the sort sequence table being used must contain a unique weight for each character in the code page. The sort sequence table used will be the unique weighted table associated with the language specified in the LANGID parameter.

***LANGIDSHR**

Specifies that the sort sequence table being used can contain the same weight for multiple characters in the code page. The sort sequence table used will be the shared weighted table associated with the language specified in the LANGID parameter.

table-name

Enter the name of the sort sequence table to be used. The table contains weights for all characters in a given code page. A weight is associated with the character that is defined at the code point. When using a sort sequence table name, the library in which the object resides can be specified. The valid values for the library are:

***LIBL**

The library list is searched to find the library where the sort sequence table is located.

***CURLIB**

The current library is used. If you have not assigned a library as the current library, QGPL is used.

library-name

Enter the name of the library where the sort sequence table is found.

LANGID Parameter:

Specifies the language identifier which is used in conjunction with the sort sequence. The LANGID parameter is used only when the SRTSEQ value in effect is *LANGIDUNQ or *LANGIDSHR. The possible values are:

***JOB RUN**

The language identifier of the module will be resolved at run time. This value allows a module to be compiled once and used with different language identifiers at run time.

***JOB**

The language identifier of the module will be resolved at compile time by using the language identifier of the compile job.

language-identifier-name

Enter a valid 3-character language identifier.

ENBPFCOL Parameter:

Specifies whether performance measurement code should be generated in the module or program. The data collected can be used by the system performance tool to profile an application's performance. Generating the addition of the performance measurement code in a compiled module or program will result in slightly larger objects and may affect performance.

***PEP**

Performance statistics are gathered on the entry and exit of the program entry procedure only. Choose this value when you want to gather overall performance information for an application. This support is equivalent to the support formally provided with the TPST tool. This is the default.

***ENTRYEXIT**

Performance Statistics are gathered on the entry and exit of all the procedures of the program. This includes the program PEP routine.

This choice would be useful if you want to capture information on all routines. Use this option when you know that all the programs called by your application were compiled with either the *PEP, *ENTRYEXIT or *FULL option. Otherwise, if your application calls other programs that are not enabled for performance measurement, the performance tool will charge their use of resources against your application. This would make it difficult for you to determine where resources are actually being used.

***FULL**

Performance statistics are gathered on the entry and exit of all procedures. Also statistics are gathered before and after each call to an external procedure.

Use this option when you think that your application will call other programs that were not compiled with either *PEP, *ENTRYEXIT or *FULL. This option allows the performance tools to distinguish between resources that are used by your application and those used by programs it calls (even if those programs are not enabled for performance measurement). This option is the most expensive, but allows for selectively analyzing various programs in an application.

```

|
| STGM DL Parameter:
| Specifies the type of storage to be used by the module.
|
| *INHERIT
| The module is created with inherit storage model. An inherit storage
| model module can be bound into programs and service programs with a
| storage model of single-level, teraspace or inherit. The type of storage used
| for automatic and static storage for single-level and teraspace storage
| model programs matches the storage model of the object. An inherit
| storage model object will inherit the storage model of its caller.
|
| *SNG LVL
| The module is created with single-level storage model. A single level
| storage model module can only be bound into programs and service
| programs that use single level storage. These programs and service
| programs use single-level storage for automatic and static storage.
|
| *TERASPACE
| The module is created with teraspace storage model. A teraspace storage
| model module can only be bound into programs and service programs that
| use teraspace storage. These programs and service programs use teraspace
| storage for automatic and static storage.
|
| PRFDTA Parameter:
| Specifies the program profiling data attribute for the module. Program
| profiling is an advanced optimization technique used to reorder procedures
| and code within the procedures based on statistical data (profiling data). For
| more information about collecting profiling data, refer to "Collecting Profiling
| Data" on page 49.
|
| *NOCOL
| This module is not enabled to collect profiling data. This is the default.
|
| *COL
| This module is enabled to collect profiling data.
|
| Note: *COL can be specified only when the optimization level of the
| module is *FULL.
|
| CCSID Parameter:
# Specifies the coded character set identifier (CCSID) that records in files, and
# data associated with LOCALEs, are converted to at run time. Also used by
# NATIONAL-OF and DISPLAY-OF intrinsic functions as the default CCSID
# value when no CCSID is specified in the intrinsic function. Also used during
# the MOVE of a single-byte data item, such as alphabetic or alphanumeric, or a
# DBCS data item, to a National data item. See ILE COBOL Reference guide,
# MOVE statement for more information.
#
# *JOB RUN
# The CCSID of the program is resolved at run time. When the compiled
# program is run, the current job's CCSID is used.
#
# *JOB
# The current job's CCSID at compile time is used.
#
# *HEX
# The CCSID 65535 is used, which indicates that data in the fields is treated
# as bit data, and is not converted.
#
# coded-character-set-identifier
# Specifies the CCSID to be used.

```

NTLCCSID Parameter:
Specifies the coded character set identifier (CCSID) to be used for National
items.
13488
CCSID 13488 will be used for National items.
coded-character-set-identifier
The specified CCSID must be compatible with UCS-2, for example UTF-16
CCSID 1200.

ARITHMETIC Parameter:

Specifies the arithmetic mode for numeric data. The possible values are:

***NOEXTEND**

This option specifies the default arithmetic mode for numeric data. The intermediate result of a fixed-point arithmetic expression can be up to 30 digits and numeric literals may only have a maximum length of 18 digits.

***EXTEND31**

Use this option to increase the precision of intermediate results for fixed-point arithmetic. The intermediate result of a fixed-point arithmetic expression can be up to 31 digits and numeric literals may have a maximum length of 31 digits.

***EXTEND63**

Use this option to increase the precision of intermediate results for fixed-point arithmetic. The intermediate result of a fixed-point arithmetic expression can be up to 63 digits and numeric literals may have a maximum length of 63 digits.

NTLPADCHAR Parameter:

This option specifies padding characters for the MOVE statement, when a national data item receives single-byte, double-byte, or national characters. Specify the padding characters in the following order:

1. Single-byte to national

The sending item is a single-byte item, such as alphabetic or alphanumeric. Specify a national hexadecimal character. The default is NX"0020".

2. Double-byte to national

The sending item is a double-byte item. Specify a national hexadecimal character. The default is NX"3000".

3. National to national

The sending item is a national item. Specify a national hexadecimal character. The default is NX"3000".

LICOPT Parameter:

Specifies one or more Licensed Internal Code compile-time options. This parameter allows individual compile-time options to be selected, and is intended for the advanced programmer who understands the potential benefits and drawbacks of each selected type of compiler option.

INCDIR Parameter:

Specifies one or more directories to add to the search path used by the compiler to find copy files. The compiler will search the directories specified here if the copy files specified in the source code cannot be resolved.

***NONE**

No user directories are searched for copy files. By default, the current directory will still be searched.


```

CRTCBLMOD MODULE(MYLIB/XMPLE1)
SRCFILE(MYLIB/QCBLLESRC) SRCMBR(MYLIB/XMPLE1)
OUTPUT(*PRINT)
TEXT('My ILE COBOL Program on iSeries')
CVTOPT(*FLOAT)

```

The CRTCBLMOD command creates the module XMPLE1 in MYLIB, the same library which contains the source. The output option OUTPUT(*PRINT) specifies a compiler listing. The conversion option CVTOPT(*FLOAT) specifies that floating-point data types are brought into the program with their DDS names and a USAGE of COMP-1 (single-precision) or COMP-2 (double-precision).

2. Type one of the following CL commands to view the compile listing.

Note: In order to view a compiler listing you must have authority to use the commands listed below.

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

Specifying a Different Target Release

You can compile a ILE COBOL program on a System i using the current release of the IBM i operating system.

Note: The ILE COBOL compiler and the OPM COBOL/400 compiler are separate individual product options. The information contained in this section applies only to the current release of the ILE COBOL compiler.

The Target Release (TGTRLS) parameter of the CRTCBLMOD and CRTBNDCBL commands allows you to specify the release level on which you intend to use the module object. The TGTRLS parameter has three possible values: *CURRENT, *PRV, and *target-release*.

- Specify *CURRENT if the module object is to be used on the release of the operating system currently running on your system. For example, if V4R4M0 is running on the system, *CURRENT means you intend to use the program on a system with V4R4M0 installed. This value is the default.
- Specify *PRV if the object is to be used on the previous release, with modification level 0, of the operating system. For example, if V4R4M0 is running on the system, *PRV means that you intend to use the object on a system with V4R3M0 installed. You can also use the object on a system with any subsequent release of the operating system installed.
- *target-release* allows you to specify the release level on which you intend to use the module object. The values you can enter for this parameter depend on the current version, release, and modification level, and they change with each new release.

Specify the release level of the target environment in the format VxRxMx. The object can be used on a system with the specified release or with any subsequent release of the operating system installed.

For example, if you specify V4R2M0, the object can be used on a V4R2M0 system.

For more information about the TGTRLS parameter, see “TGTRLS Parameter” on page 42.

You should be aware of the following limitations:

- You can restore an object program to the current release or to a subsequent release. You cannot restore an object program on a previous release that is not allowed by the TGTRLS *target-release*.
- No product library should be in the system portion of your library list.

Specifying National Language Sort Sequence in CRTCBMOD

At the time that you compile your ILE COBOL source program, you can explicitly specify the collating sequence that the program will use when it is run, or you can specify how the collating sequence is to be determined when the program is run.

To specify the collating sequence, you first define an *alphabet-name* in the SPECIAL-NAMES paragraph using the ALPHABET clause and associate that *alphabet-name* with the NLSSORT implementor name. Then, refer to this *alphabet-name* in the PROGRAM COLLATING SEQUENCE clause in the ENVIRONMENT DIVISION, or in the COLLATING SEQUENCE phrase in the SORT/MERGE statements, to denote that the specified *alphabet-name* will determine the collating sequence to be used.

You specify the actual collating sequence used, through the options of the SRTSEQ and LANGID parameters of the CRTCBMOD and CRTBNDCBL commands. For example, if you specify SRTSEQ(*JOB RUN) and LANGID(*JOB RUN), the collating sequence of the program will be resolved at run time. This value allows the source program to be compiled once and used with different collating sequences at run time. The PROCESS statement options associated with SRTSEQ and LANGID may also be used to specify the collating sequence (see “Using the PROCESS Statement to Specify Compiler Options” on page 51).

If your source program does not have NLSSORT associated with an *alphabet-name* in its ALPHABET clause, or has an ALPHABET clause specifying NLSSORT but the associated *alphabet-name* is not referred to in any PROGRAM COLLATING SEQUENCE clause or COLLATING SEQUENCE phrase of SORT/MERGE statements, then the sort sequence identified by the SRTSEQ and LANGID parameters is not used.

The *alphabet-name* associated with NLSSORT cannot be used to determine character code set, as in the CODE-SET clause of the File Description (FD) entry. The *alphabet-name* used to determine character code set must be identified in a separate ALPHABET clause.

Refer to the *IBM Rational Development Studio for i: ILE COBOL Reference* for a full description of the ALPHABET clause, PROGRAM COLLATING SEQUENCE clause, and SORT/MERGE statements. Refer to “Parameters of the CRTCBMOD Command” on page 28 for a description of the SRTSEQ and LANGID parameters.

Collecting Profiling Data

Once profiling code has been added to a module, it must be placed in a program object or service program object in order for profiling data to be collected. The profiling data can be applied to a program object with the CHGPGM CL command and applied to a service program with the CHGSRVPGM CL command. To apply all the profiling data to a program object or service program specify the PRFDTA parameter with the Apply All (*APYALL) value. To only apply the profiling data

that reorders code within procedures specify the value *APYBLKORD. To only apply the profiling data that reorders procedures specify *APYPRCORD.

Profiling data is collected by specifying the Start Program Profiling (STRPGMPRF) CL command. All the program objects and service programs that are active on the system and that include profiling code will generate profiling data.

Once enough profiling data has been collected, the End Program Profiling (ENDPGMPRF) CL command should be entered.

Program profiling data can be removed from the modules within a program object or service program with the *CLR value of the PRFDTA parameter on the CHGPGM and CHGSRVPGM CL commands.

Enabling a module to collect profiling data causes additional code to be generated in the module object. This code is used to collect data on the number of times basic blocks within procedures have been executed, as well as the number of times procedures have been called. To enable collection of profiling data, modules must be compiled at an optimization level of 30 (*FULL), or greater.

Data that is collected for the basic blocks within procedures is used by the ILE optimizing translator to rearrange these blocks for better cache utilization. Block information is applied to procedures within a module; it does not span module boundaries.

The binder uses the procedure call data in order to package procedures that often call each other together for better page utilization. In other words, it is possible to have PROC A in module A packaged next to PROC B in module B (if PROC A makes many calls to PROC B) in the profiled program. Procedure call data is applied at the program level; it does span module boundaries.

Profiling data can only be collected if the current target release is specified. In order for an ILE program or service program to be profiled, the program *must* have a target release of V4R2M0, or later. This also means that a program enabled to collect profiling data or a profiled program cannot be saved or restored to a release earlier than V4R2M0.

For more information about the PRFDTA parameter, refer to page “PRFDTA Parameter” on page 45.

Note: The potential for inaccuracies in the collected data exists if profile data is collected for programs running in a parallel environment, for example, a multi-threaded process.

Specifying Date, Time, and Timestamp Data Types

Items of COBOL class date-time, include date, time, and timestamp items. These items are declared with the FORMAT clause of a data description entry. For example:

```
01 group-item.  
   05 date1 FORMAT DATE "%m/%d/@Y".  
   05 date2 FORMAT DATE.
```

For items of class date-time the FORMAT clause is used in place of a PICTURE clause. In the example above, after the keyword FORMAT the keyword DATE declares an item of category date. After the keyword date a format literal describes

the format of the date data item. In the case of data item date1 the %m stands for months, %d for days, and the @Y for year (including a 2-digit century). The % and @ character begin a specifier. The three specifiers shown here are part of a set of specifiers documented in the *IBM Rational Development Studio for i: ILE COBOL Reference* .

The other date data item, date2, has no format literal explicitly specified; however, a default date format can be specified in the SPECIAL-NAMES paragraph. An example is shown below:

```
SPECIAL-NAMES.  FORMAT OF DATE IS "@C:%y:%j".
```

If the above SPECIAL-NAMES paragraph had been specified in the same program as the data item, date2, its date format would have been @C:%y:%j. On the other hand, if a SPECIAL-NAMES paragraph did not exist, the format of the date item would default to ISO. An ISO date has the format @Y-%m-%d.

By default when COPY DDS declares items of class date-time it generates a PICTURE clause for an alphanumeric item. In order to change the PICTURE clause into a FORMAT clause, several new CVTOPT parameter values have been defined. These are:

- *DATE
- *TIME
- *TIMESTAMP.

When *DATE has been specified, any DDS date data types are converted to COBOL date items; in other words, a FORMAT clause is generated instead of a PICTURE clause.

In DDS to specify the format of a date field, the DATFMT keyword can be specified. The DATFMT keyword can also be specified on zoned, packed, and character fields. For these types of fields, COPY DDS would normally generate a PICTURE clause for a numeric zoned, numeric packed, and alphanumeric data item, respectively. You can force COPY DDS to generate a FORMAT clause for these items by specifying the *CVTTODATE value of the CVTOPT parameter.

For a list of the DATFMT parameters allowed for zoned, packed, and character DDS fields, and their equivalent ILE COBOL format that is generated from COPY DDS when the CVTOPT(*CVTTODATE) conversion parameter is specified, refer to "Class Date-Time" on page 440 and "Working with Date-Time Data Types" on page 189.

For moves and comparisons involving a mixture of 4-digit and 2-digit dates, ILE COBOL uses a default windowing algorithm with a base century of 1900 and a base year of 40. Because inaccuracies can result, it may be necessary to override the default window. For more information about the ILE COBOL windowing algorithm and how to override it, refer to "Conversion of 2-Digit Years to 4-Digit Years or Centuries" on page 192.

Using the PROCESS Statement to Specify Compiler Options

The PROCESS statement is an optional part of the ILE COBOL source program. You can use the PROCESS statement to specify options you would normally specify at compilation time.

Options specified in the PROCESS statement **override** the corresponding options specified in the CRTCBMOD or CRTBNDCBL CL command.

The following rules apply:

- The statement must be placed before the first source statement in the ILE COBOL source program, which starts a new compilation unit, immediately preceding the IDENTIFICATION DIVISION header.
- The statement begins with the word PROCESS. Options can appear on more than one line; however, only the first line can contain the word PROCESS.
- The word PROCESS and all options must appear within positions 8 through 72. Position 7 must be left blank. The remaining positions can be used as in ILE COBOL source statements: positions 1 through 6 for sequence numbers, positions 73 through 80 for identification purposes.
- The options must be separated by blanks and/or commas.
- Options can appear in any order. If conflicting options are specified, for example, XREF and NOXREF, the last option encountered takes precedence.
- If the option keyword is correct and the suboption is in error, the default suboption is assumed.

The following tables indicate the allowable PROCESS statement options and the equivalent CRTCBMOD or CRTBNDCBL command parameters and options. Defaults are underlined. Descriptions of the PROCESS statement options correspond to the parameter and option descriptions under "Parameters of the CRTCBMOD Command" on page 28.

Note: Not every parameter of the CRTCBMOD and CRTBNDCBL commands has a corresponding option in the PROCESS statement. In addition, several options are only available on the process statement. For descriptions of the options that are only on the PROCESS statement, see "PROCESS Statement Options" on page 59.

PROCESS Statement Options	CRTCBMOD/CRTBNDCBL
	OUTPUT Parameter Options
<u>OUTPUT</u> NOOUTPUT	* <u>PRINT</u> *NONE

PROCESS Statement Option	CRTCBMOD/CRTBNDCBL
	GENLVL Parameter Option
GENLVL(nn)	nn

PROCESS Statement Options	CRTCBMOD/CRTBNDCBL
	OPTION Parameter Options
<u>SOURCE</u> <u>SRC</u> NOSOURCE NOSRC	* <u>SOURCE</u> * <u>SRC</u> *NOSOURCE *NOSRC
<u>NOXREF</u> <u>XREF</u>	* <u>NOXREF</u> * <u>XREF</u>
<u>GEN</u> NOGEN	* <u>GEN</u> *NOGEN
<u>NOSEQUENCE</u> <u>SEQUENCE</u>	* <u>NOSEQUENCE</u> * <u>SEQUENCE</u>

PROCESS Statement Options	CRTCBLMOD/CRTBNDCBL
	OPTION Parameter Options
<u>NOVBSUM</u> VBSUM	* <u>NOVBSUM</u> *VBSUM
<u>NONUMBER</u> NUMBER LINENUMBER	* <u>NONUMBER</u> *NUMBER *LINENUMBER
<u>NOMAP</u> MAP	* <u>NOMAP</u> *MAP
<u>NOOPTIONS</u> OPTIONS	* <u>NOOPTIONS</u> *OPTIONS
<u>QUOTE</u> APOST	* <u>QUOTE</u> *APOST
<u>NOSECLVL</u> SECLVL	* <u>NOSECLVL</u> *SECLVL
<u>PRTCORR</u> NOPRTCORR	* <u>PRTCORR</u> *NOPRTCORR
<u>MONOPRC</u> NOMONOPRC	* <u>MONOPRC</u> *NOMONOPRC
<u>RANGE</u> NORANGE	* <u>RANGE</u> *NORANGE
<u>NOUNREF</u> UNREF	* <u>NOUNREF</u> *UNREF
<u>NOSYNC</u> SYNC	* <u>NOSYNC</u> *SYNC
<u>NOCRTF</u> CRTF	* <u>NOCRTF</u> *CRTF
<u>NODUPKEYCHK</u> DUPKEYCHK	* <u>NODUPKEYCHK</u> *DUPKEYCHK
<u>NOINZDLT</u> INZDLT	* <u>NOINZDLT</u> *INZDLT
<u>NOBLK</u> BLK	* <u>NOBLK</u> *BLK
<u>STDINZ</u> NOSTDINZ STDINZHEX00	* <u>STDINZ</u> *NOSTDINZ *STDINZHEX00
<u>NODDSFILLER</u> DDSFILLER	* <u>NODDSFILLER</u> *DDSFILLER
Not applicable	* <u>NOIMBEDERR</u> *IMBEDERR
<u>STDTRUNC</u> NOSTDTRUNC	* <u>STDTRUNC</u> *NOSTDTRUNC
<u>CHGPOSSGN</u> NOCHGPOSSGN	* <u>CHGPOSSGN</u> *NOCHGPOSSGN
Not applicable	* <u>NOEVENTF</u> *EVENTF
<u>MONOPIC</u> NOMONOPIC	* <u>MONOPIC</u> *NOMONOPIC

PROCESS Statement Options	CRTCBLMOD/CRTBNDCBL
	OPTION Parameter Options
<u>NOCRTARKIDX</u> CRTARKIDX	<u>*NOCRTARKIDX</u> *CRTARKIDX

PROCESS Statement Options	CRTCBLMOD/CRTBNDCBL
	CVTOPT Parameter Options
<u>NOVARCHAR</u> VARCHAR	<u>*NOVARCHAR</u> *VARCHAR
<u>NODATETIME</u> DATETIME	<u>*NODATETIME</u> *DATETIME
<u>NOCVPICXGRAPHIC</u> CVTPICXGRAPHIC CVTPICGGRAPHIC NOCVTPICGGRAPHIC	<u>*NOPICXGRAPHIC</u> *PICXGRAPHIC *PICGGRAPHIC *NOPICGGRAPHIC
<u>NOCVPICNGRAPHIC</u> CVTPICNGRAPHIC	<u>*NOPICNGRAPHIC</u> *PICNGRAPHIC
<u>NOFLOAT</u> FLOAT	<u>*NOFLOAT</u> *FLOAT
<u>NODATE</u> DATE	<u>*NODATE</u> *DATE
<u>NOTIME</u> TIME	<u>*NOTIME</u> *TIME
<u>NOTIMESTAMP</u> TIMESTAMP	<u>*NOTIMESTAMP</u> *TIMESTAMP
<u>NOCVTTODATE</u> CVTTODATE	<u>*NOCVTTODATE</u> *CVTTODATE

PROCESS Statement Options	CRTCBLMOD/CRTBNDCBL
	OPTIMIZE Parameter Options
<u>NOOPTIMIZE</u> BASICOPT FULLOPT NEVEROPTIMIZE	<u>*NONE</u> *BASIC *FULL *NEVER

PROCESS Statement Options	CRTCBLMOD/CRTBNDCBL
	FLAGSTD Parameter Options
<u>NOFIPS</u> MINIMUM INTERMEDIATE HIGH	<u>*NOFIPS</u> *MINIMUM *INTERMEDIATE *HIGH
<u>NOOBSOLETE</u> OBSOLETE	<u>*NOOBSOLETE</u> *OBSOLETE

PROCESS Statement Options EXTDSPOPT(<i>a b c</i>)	CRTCBLMOD/CRTBNDCBL
	EXTDSPOPT Parameter Options
<u>DFRWRT</u> NODFRWRT	<u>*DFRWRT</u> *NODFRWRT

PROCESS Statement Options EXTDSPOPT(<i>a b c</i>)	CRTCBLMOD/CRTBNDCBL
	EXTDSPOPT Parameter Options
<u>UNDSPCHR</u> NOUNDSPCHR	* <u>UNDSPCHR</u> *NOUNDSPCHR
<u>ACCUPDALL</u> ACCUPDNE	* <u>ACCUPDALL</u> *ACCUPDNE

PROCESS Statement Option	CRTCBLMOD/CRTBNDCBL
	FLAG Parameter Option
FLAG(nn)	nn

PROCESS Statement Options	CRTCBLMOD/CRTBNDCBL
	LINKLIT Parameter Options
<u>LINKPGM</u> LINKPRC	* <u>PGM</u> *PRC

PROCESS Statement Options SRTSEQ(<i>a</i>)	CRTCBLMOD/CRTBNDCBL
	SRTSEQ Parameter Options
<u>HEX</u> JOB JOBRUN LANGIDUNQ LANGIDSHR "LIBL/sort-seq-table-name" "CURLIB/sort-seq-table-name" "library-name/sort-seq-table-name" "sort-seq-table-name"	* <u>HEX</u> *JOB *JOBRUN *LANGIDUNQ *LANGIDSHR *LIBL/sort-seq-table-name *CURLIB/sort-seq-table-name library-name/sort-seq-table-name sort-seq-table-name

PROCESS Statement Options LANGID(<i>a</i>)	CRTCBLMOD/CRTBNDCBL
	LANGID Parameter Options
<u>JOBRUN</u> JOB "language-identifier-name"	* <u>JOBRUN</u> *JOB language-identifier-name

PROCESS Statement Options ENBPFCOL(<i>a</i>)	CRTCBLMOD/CRTBNDCBL
	ENBPFCOL Parameter Options
<u>PEP</u> ENTRYEXIT FULL	* <u>PEP</u> *ENTRYEXIT *FULL

PROCESS Statement Options PRFDTA(<i>a</i>)	CRTCBLMOD/CRTBNDCBL
	PRFDTA Parameter Options
<u>NOCOL</u> COL	* <u>NOCOL</u> *COL

PROCESS Statement Options CCSID(<i>a b c d</i>)	CRTCBLMOD/CRTBND CBL CCSID Parameter Options
<i>a</i> = Locale single-byte data CCSID	
<u>JOBRUN</u> JOB HEX <i>coded-character-set-identifier</i>	<u>*JOBRUN</u> *JOB *HEX <i>coded-character-set-identifier</i>
<i>b</i> = Non-locale single-byte data CCSID	
<u>CCSID</u> (uses CCSID specified for “ <i>a</i> ” above) JOBRUN JOB HEX <i>coded-character-set-identifier</i>	Not applicable
<i>c</i> = Non-locale double-byte data CCSID	
<u>CCSID</u> (uses CCSID specified for “ <i>a</i> ” above) JOBRUN JOB HEX <i>coded-character-set-identifier</i>	Not applicable
<i>d</i> = XML GENERATE single-byte or unicode data output CCSID	
<u>JOBRUN</u> CCSID (uses CCSID specified for “ <i>a</i> ” above) JOB HEX <i>coded-character-set-identifier</i>	Not applicable

|

#

PROCESS Statement Option NTLCCSID(<i>a</i>)	CRTCBLMOD/CRTBND CBL NTLCCSID Parameter Options
<u>13488</u> <i>coded-character-set-identifier</i>	<u>13488</u> <i>coded-character-set-identifier</i>

PROCESS Statement Options DATTIM(<i>a b</i>)	CRTCBLMOD/CRTBND CBL
<i>4-digit base century</i> (default 1900) <i>2-digit base year</i> (default 40)	Not applicable

PROCESS Statement Options THREAD(<i>a</i>)	CRTCBLMOD/CRTBND CBL
<u>NOTHREAD</u> SERIALIZE	Not applicable

PROCESS Statement Options ARITHMETIC(<i>a</i>)	CRTCBLMOD/CRTBND CBL ARITHMETIC Parameter Options
<u>NOEXTEND</u> EXTEND31 EXTEND63	<u>*NOEXTEND</u> *EXTEND31 *EXTEND63

PROCESS Statement Option	CRTCBLMOD/CRTBNDCBL
<u>NOGRAPHIC</u> GRAPHIC	Not applicable

PROCESS Statement Option	CRTCBLMOD/CRTBNDCBL
<u>NONATIONAL</u> NATIONAL	Not applicable

PROCESS Statement Option	CRTCBLMOD/CRTBNDCBL
<u>NOLSPTRALIGN</u> LSPTRALIGN	Not applicable

PROCESS Statement Option	CRTCBLMOD/CRTBNDCBL
<u>NOCOMPASBIN</u> COMPASBIN	Not applicable

PROCESS Statement Option	CRTCBLMOD/CRTBNDCBL
	DBGVIEW Parameter Options
<u>NOCOMPRESSDBG</u> COMPRESSDBG	* <u>NOCOMPRESSDBG</u> *COMPRESSDBG

PROCESS Statement Option OPTVALUE(<i>a</i>)	CRTCBLMOD/CRTBNDCBL
<u>NOOPT</u> OPT	Not applicable

PROCESS Statement Option	CRTCBLMOD/CRTBNDCBL
<u>NOADJFILLER</u> ADJFILLER	Not applicable

PROCESS Statement Option NTPADCHAR(<i>a b c</i>)	CRTCBLMOD/CRTBNDCBL
	NTPADCHAR Parameter Options
<i>a</i> = padding character for moving single-byte to national	
<u>NX"0020"</u> a national hexadecimal literal representing one national character	<u>NX"0020"</u> a national character
<i>b</i> = padding character for moving double-byte to national	
<u>NX"3000"</u> a national hexadecimal literal representing one national character	<u>NX"3000"</u> a national character
<i>c</i> = padding character for moving national to national	
<u>NX"3000"</u> a national hexadecimal literal representing one national character	<u>NX"3000"</u> a national character

PROCESS Statement Option LICOPT(a)	CRTCBLMOD/CRTBND CBL
	LICOPT Parameter Option
licensed-internal-code-option-string	licensed-internal-code-option-string

#

PROCESS Statement Option PGMINFO(a b)	CRTCBLMOD/CRTBND CBL
	PGMINFO Parameter Options
<i>a</i> = program interface information to be generated	
<u>NO</u> PCML	<u>*NO</u> <u>*PCML</u>
<i>b</i> = location for the generated program information	
MODULE	<u>*STMF</u> <u>*MODULE</u> <u>*ALL</u>

|
|
|
|
|
|
|
|
|
|

PROCESS Statement Options STGMDL(a)	CRTCBLMOD
	STGMDL Parameter Options
<u>INHERIT</u> SNGLVL TERASPACE	<u>*INHERIT</u> <u>*SNGLVL</u> <u>*TERASPACE</u>

|
|
|
|
|
|
|
|
|
|

PROCESS Statement Options STGMDL(a)	CRTBND CBL
	STGMDL Parameter Options
<u>SNGLVL</u> <u>INHERIT</u> TERASPACE	<u>*SNGLVL</u> <u>*INHERIT</u> <u>*TERASPACE</u>

|
|
|
|
|
|
|
|
|
|

PROCESS Statement Options ACTGRP(a)	CRTBND CBL
	ACTGRP Parameter Options
<u>STGMDL</u> NEW CALLER 'activation-group-name'	<u>*STGMDL</u> <u>*NEW</u> <u>*CALLER</u> activation-group-name

|
|
|
|
|
|

The EXTDSPOPT, SRTSEQ, LANGID, ENBPFCOL, PRFDTA, CCSID, DATTIM, ARITHMETIC, THREAD, NTLCCSID, STGMDL, ACTGRP, and PGMINFO options on the PROCESS statement should be coded with the associated options in brackets similar to FLAG(nn) syntax.

You can specify more than one option within the brackets for the EXTDSPOPT option. For example, to specify DFRWRT and UNDSPCHR, type
EXTDSPOPT(DFRWRT UNDSPCHR)

It is also valid to specify EXTDSPOPT or EXTDSPOPT().

When EXTDSPOPT alone is specified in the PROCESS statement, then all the default values for the additional options are in effect.

If you specify EXTDSPOPT(), it has no effect on your program.

The DATTIM, CCSID, and PGMINFO process statement options also allow more than one value within their brackets. For these options, the order of the values within their brackets is significant. For example, the DATTIM option has two values. The first is the base Century, and the second is the base year. This means that you must specify a base century in order to specify a base year.

PROCESS Statement Options

The following options are only available on the PROCESS statement and do not have equivalent CRTCLMOD or CRTBNDCBL command parameters.

NOGRAPHIC Option:

When NOGRAPHIC is specified or implied, the ILE COBOL compiler will treat nonnumeric literals containing hex 0E and hex 0F as if they only contain SBCS characters. Hex 0E and hex 0F are not treated as shift-in and shift-out characters, they are considered to be part of the SBCS character string. See Appendix D, "Supporting International Languages with Double-Byte Character Sets," on page 617 for information about DBCS support.

GRAPHIC Option:

The GRAPHIC option of the PROCESS statement is available for processing DBCS characters in mixed literals. **Mixed literals** are literals that combine SBCS characters and DBCS characters. When the GRAPHIC option is specified, mixed literals will be handled with the assumption the hex 0E and hex 0F are shift-in and shift-out characters respectively, and they enclose the DBCS characters in the mixed literal. Shift-in and shift-out characters occupy 1 byte each.

DATTIM Option:

Specifies the date window that ILE COBOL uses for its windowing algorithm. (See "Overriding the Default Date Window Using the DATTIM PROCESS Statement Option" on page 194.)

4-digit base century

This must be the first argument. Defines the base century that ILE COBOL uses for its windowing algorithm. If the DATTIM process statement option is not specified, 1900 is used.

2-digit base year

This must be the second argument. Defines the base year that ILE COBOL uses for its windowing algorithm. If the DATTIM process statement option is not specified, 40 is used.

THREAD Option:

Specifies whether or not the created module object will be enabled to run in a multithreaded environment. Refer to Chapter 15, "Preparing ILE COBOL Programs for Multithreading," on page 361 for a discussion of ILE COBOL support for multithreading. The possible values are:

NOTHREAD

The created module object will *not* be enabled to run in a multithreaded environment. This is the default.

SERIALIZE

The created module object will be enabled to run in a job with multiple threads. Access to procedures within the module(s) is serialized. That is, each thread safe module will have a recursive mutex that is locked when a procedure is entered and unlocked when the procedure is exited. Within a run unit, only one thread is allowed to be active at any one time for the same module.

NONATIONAL Option:

When NONATIONAL is specified or implied, USAGE DISPLAY-1 is implied for any item that has a picture character string consisting of only the picture symbol N and no explicit USAGE clause.

NATIONAL Option:

When NATIONAL is specified, USAGE NATIONAL is implied for any item that has a picture character string consisting of only the picture symbol N and no explicit USAGE clause.

NOLSPTRALIGN Option:

When NOLSPTRALIGN is specified or implied, data items with USAGE POINTER or PROCEDURE-POINTER are placed contiguously without any filler space in the linkage section.

LSPTRALIGN Option:

When LSPTRALIGN is specified, data items with USAGE POINTER or PROCEDURE-POINTER are aligned at multiples of 16 bytes relative to the beginning of the record in the linkage section.

NOCOMPASBIN Option:

When NOCOMPASBIN is specified or implied, USAGE COMPUTATIONAL or COMP has the same meaning as USAGE COMP-3.

COMPASBIN Option:

When COMPASBIN is specified, USAGE COMPUTATIONAL or COMP has the same meaning as USAGE COMP-4.

OPTVALUE Option:

The possible values are:

NOOPT

The generation of code to initialize data items containing a VALUE clause in the working-storage section is not optimized. This is the default.

OPT

The generation of code to initialize data items containing a VALUE clause in the working-storage section is optimized.

NOADJFILLER Option:

If a pointer data item is the first member of a group, any implicit fillers inserted by the compiler to align this pointer data item are inserted immediately after the group. This is the default.

ADJFILLER Option:

If a pointer data item is the first member of a group, any implicit fillers inserted by the compiler to align this pointer data item are inserted immediately before the group.

Compiling Multiple Source Programs

The PROCESS statement can be placed at the beginning of each compilation unit in the sequence of ILE COBOL source programs in the input source member. When compiling multiple ILE COBOL source programs, the merged results of all options specified on the CRTCLMOD or CRTBNDCBL command, plus all default options, plus the options specified on the last PROCESS statement preceding the ILE COBOL source program will be in effect for the compilation of that ILE COBOL source program. All compiler output is directed to the destinations specified by the CRTCLMOD or CRTBNDCBL command.

All module objects or program objects are stored in the library specified on the MODULE parameter or PGM parameter. If *module-name* or *program-name* is specified for the MODULE parameter or PGM parameter, the first module object or program object corresponding to the first ILE COBOL source program in the sequence of ILE COBOL source programs use that name, and all module objects or program objects corresponding to the other ILE COBOL source programs in the same input source member use the name specified in the PROGRAM-ID paragraph in the ILE COBOL source program.

Using COPY within the PROCESS Statement

A COPY statement can be used in the source program wherever a character-string or separator can be used. Each COPY statement must be preceded by a space and followed by a period and a space. For more information on the COPY statement, refer to the "COPY Statement" section of the *IBM Rational Development Studio for i: ILE COBOL Reference*.

The Format 1 COPY statement can be used within the PROCESS statement to retrieve compiler options previously stored in a source library, and include them in the PROCESS statement. COPY can be used to include options that override those specified as defaults by the compiler. Any PROCESS statement options can be retrieved with the COPY statement.

Compiler options can both precede and follow the COPY statement within the PROCESS statement. The last encountered occurrence of an option overrides all preceding occurrences of that option.

The following example shows the use of the COPY statement within the PROCESS statement. Notice also that, in this example, NOMAP overrides the corresponding option in the library member:

```

5722WDS V5R4M0 060210 LN  IBM ILE COBOL  CBLGUIDE/COPYPROC  ISERIES1  06/02/15 11:39:37  Page  2
                               S o u r c e
STMT PL SEQNBR -A 1 B...2...3...4...5...6...7..IDENTFCN  S COPYNAME  CHG DATE
      000100 PROCESS XREF
      000200 COPY PROCDFLT.
+000100      MAP, SOURCE, APOST                                PROCDFLT
+000200                                                                PROCDFLT
      000300      NOMAP, FLAG(20)
1      000400 IDENTIFICATION DIVISION.
2      000500 PROGRAM-ID. COPYPROC.
3      000600 ENVIRONMENT DIVISION.
4      000700 CONFIGURATION SECTION.
5      000800 SOURCE-COMPUTER. IBM-ISERIES
6      000900 OBJECT-COMPUTER. IBM-ISERIES
7      001000 PROCEDURE DIVISION.
      001100 MAINLINE.
8      001200      DISPLAY "HELLO WORLD".
9      001300      STOP RUN.
      001400
                               * * * * *  E N D   O F   S O U R C E  * * * * *

```

Figure 9. Using COPY within the PROCESS Statement

Understanding Compiler Output

The compiler can be directed to produce a selection of printed reports. By default, this output will be directed to the system printer file QSYSPRT.

The output can include:

- A summary of command options

- An **options listing**, which is a listing of options in effect for the compilation. Use OPTION(*OPTIONS).
- A **source listing**, which is a listing of the statements contained in the source program. Use OPTION(*SOURCE).
- A **verb usage listing**, which is a listing of the COBOL verbs and the number of times each verb is used. Use OPTION(*VBSUM).
- A **Data Division map**, which is a glossary of compiler-generated information about the data. Use OPTION(*MAP).
- **FIPS messages**, which is a list of messages for a FIPS COBOL subset, for any of the optional modules, for all of the obsolete language elements, or for a combination of a FIPS COBOL subset, optional modules and all obsolete elements. Refer to the information on the “FLAGSTD Parameter” on page “FLAGSTD Parameter” on page 40 for the specific options available for FIPS flagging.
- A **cross-reference listing**. Use OPTION(*XREF).
- An **imbedded error listing**. Use OPTION(*IMBEDERR).
- **Compiler messages** (including diagnostic statistics).
- **Compilation statistics**.
- **Module objects**. Use the CRTCBMOD command.
- **Program objects**. Use the CRTBNDCBL command.

The presence or absence of some of these types of compiler output is determined by options specified in the PROCESS statement or through the CRTCBMOD or CRTBNDCBL command. The level of diagnostic messages printed depends upon the FLAG option. The DBGVIEW option dictates what kind of debug data is contained in the generated module object or program object.

Specifying the Format of Your Listing

A slash (/) in the indicator area (column 7) of a line results in page ejection of the source program listing. You can also enter comment text after the slash (/) on this line. The slash (/) comment line prints on the first line of the next page.

If you specify the EJECT statement in your program, the next source statement prints at the top of the next page in the compiler listing. This statement may be written anywhere in Area A or Area B and must be the only statement on the line.

The SKIP1/2/3 statement causes blank lines to be inserted in the compiler listing. A SKIP1/2/3 statement can be written anywhere in Area A or B. It must be the only statement on the line.

- SKIP1 inserts a single blank line (double spacing).
- SKIP2 inserts two blank lines (triple spacing).
- SKIP3 inserts three blank lines (quadruple spacing).

Each of the above SKIP statements causes a single insertion of one, two, or three lines.

A TITLE statement places a title on each indicated page.

You can selectively list or suppress your ILE COBOL source statements by using the *CONTROL, *CBL, or COPY statements:

- *CONTROL NOSOURCE and *CBL NOSOURCE suppress the listing of source statements.

- *CONTROL SOURCE and *CBL SOURCE continue the listing of source statements.
- A COPY statement bearing the SUPPRESS phrase suppresses the listing of copied statements. For its duration, this statement overrides any *CONTROL or *CBL statement. If the copied member contains *CONTROL or *CBL statements, the last one runs once the COPY member has been processed.

Refer to the *IBM Rational Development Studio for i: ILE COBOL Reference* for additional information about the EJECT, SKIP1/2/3, *CONTROL, *CBL, COPY, and TITLE statements.

Time-Separation Characters

The TIMSEP parameter of job-related commands (such as CHGJOB) now specifies the time-separation character used in the time stamps that appear on compiler listings. In the absence of a TIMSEP value, the system value QTIMSEP is used by default.

Browsing Your Compiler Listing Using SEU

The Source Entry Utility (SEU) allows you to browse through a compiler listing in an output queue. You can review the results of a previous compilation while making the required changes to your source code.

See *Using the application development tools in the client product* for information about getting started with the client tools.

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.

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

A Sample Program and Listing

The following sample listings illustrate the compiler options and source listing produced for the program example. References to the figures are made throughout the following text. These references are indexed by the reverse printing of letters on a black background, for example (**Z**). The reverse letters in the text correspond to the letters found in the figures.

Command Summary

This summary, produced as a result of compilation, lists all options specified in the CRTCBMOD or CRTBNDCBL command. Refer to “Using the Create COBOL Module (CRTCBMOD) Command” on page 24 for more information about user-defined options.

```

5722WDS V5R4M0 060210 LN  IBM ILE COBOL          CBLGUIDE/EXTLFL  ISERIES1  06/02/15 13:11:39  Page  1
Command . . . . . : CRTCBMOD
Actual Values:
Module . . . . . : EXTLFL
Library . . . . . : CBLGUIDE
Source file . . . . . : QCBLESRC
Library . . . . . : CBLGUIDE
CCSID . . . . . : 37
Source member . . . . . : EXTLFL          02/03/05 10:50:50
Text 'description' . . . . . : *BLANK
Command Options:
Module . . . . . : EXTLFL
Library . . . . . : CBLGUIDE
Source file . . . . . : QCBLESRC
Library . . . . . : CBLGUIDE
Source member . . . . . : EXTLFL
Output . . . . . : *PRINT
Generation severity level . . . . . : 30
Text 'description' . . . . . : *SRCMBRTXT
Compiler options . . . . . : *NONE
Conversion options . . . . . : *NONE
Message limit:
Number of messages . . . . . : *NOMAX
Message limit severity . . . . . : 30
Debug view option:
Debug view . . . . . : *STMT
Compress listing view . . . . . : *NOCOMPRESSDBG
Optimize level . . . . . : *NONE
FIPS flagging . . . . . : *NOFIPS *NOOBSOLETE
Extended display options . . . . . : *NONE
Flagging severity . . . . . : 0
Replace module . . . . . : *NO
Authority . . . . . : *LIBCRTAUT
Link literal . . . . . : *PGM
Target release . . . . . : *CURRENT
Sort sequence . . . . . : *HEX
Library . . . . . :
Language identifier . . . . . : *JOB RUN
Enable performance collection:
Collection level . . . . . : *PEP
Profiling data . . . . . : *NOCOL
Coded character set ID . . . . . : *JOB RUN
Arithmetic mode . . . . . : *NOEXTEND
Padding character:
Single byte to national . . . . . : NX"0020"
Double byte to national . . . . . : NX"3000"
National to national . . . . . : NX"3000"
Include directory . . . . . : *NONE
Generate program information . . . . . : *NO
Compiler . . . . . : IBM ILE COBOL

```

Figure 10. CRTCBMOD Command Summary Listing

```

5722WDS V5R4M0 060210 LN  IBM ILE COBOL          CBLGUIDE/SAMPLE      ISERIES1  06/02/15 11:18:21      Page      1
Command . . . . . : CRTBNDCBL
Actual Values:
  Program . . . . . : SAMPLE
  Library . . . . . : CBLGUIDE
  Source file . . . . . : QCBLLSRC
  Library . . . . . : CBLGUIDE
  CCSID . . . . . : 37
  Source member . . . . . : SAMPLE          02/03/05 14:13:55
  Text 'description' . . . . . : *BLANK
Command Options:
  Program . . . . . : SAMPLE
  Library . . . . . : CBLGUIDE
  Source file . . . . . : QCBLLSRC
  Library . . . . . : CBLGUIDE
  Source member . . . . . : SAMPLE
  Output . . . . . : *PRINT
  Generation severity level . . . . . : 30
  Text 'description' . . . . . : *SRCMBRTXT
  Compiler options . . . . . : *IMBEDERR
  Conversion options . . . . . : *NONE
Message limit:
  Number of messages . . . . . : *NOMAX
  Message limit severity . . . . . : 30
  Message limit severity . . . . . : 30
Debug view option:
  Debug view . . . . . : *STMT
  Compress listing view . . . . . : *NOCOMPRESSDBG
  Optimize level . . . . . : *NONE
  FIPS flagging . . . . . : *NOFIPS *NOOBSOLETE
  Extended display options . . . . . : *NONE
  Flagging severity . . . . . : 0
  Replace program . . . . . : *YES
  Simple program . . . . . : *YES
  Authority . . . . . : *LIBCRTAUT
  Link literal . . . . . : *PGM
  Target release . . . . . : *CURRENT
  User profile . . . . . : *USER
  Sort sequence . . . . . : *HEX
  Library . . . . . :
  Language identifier . . . . . : *JOBRUN
Enable performance collection:
  Collection level . . . . . : *PEP
  Binding directory . . . . . : *NONE
  Library . . . . . :
  Activation group . . . . . : QILE
  Profiling data . . . . . : *NOCOL
  Coded character set ID . . . . . : *JOBRUN
  Arithmetic mode . . . . . : *NOEXTEND
Padding character:
  Single byte to national . . . . . : NX"0020"
  Double byte to national . . . . . : NX"3000"
  National to national . . . . . : NX"3000"
Include directory . . . . . : *NONE
Generate program information . . . . . : *NO
Compiler . . . . . : IBM ILE COBOL

```

Figure 11. CRTBNDCBL Command Summary Listing

Identifying the Compiler Options in Effect

The PROCESS statement, if specified, is printed first. Figure 12 on page 66 is a list of all options in effect for the compilation of the program example: the options specified in the CRTCBMOD command, as modified by the PROCESS statement. Compiler options are listed at the beginning of all compiler output when the OPTIONS parameter is specified.

```

Source
STMT PL SEQNBR -A 1 B.+....2....+....3....+....4....+....5....+....6....+....7..IDENTFCN S COPYNAME CHG DATE
000100 PROCESS OPTIONS, SOURCE, VBSUM, MAP,
000200 FLAG(00), MINIMUM, OBSOLETE, XREF
COBOL Compiler Options in Effect
SOURCE
XREF
GEN
NOSEQUENCE
VBSUM
NONUMBER
MAP
OPTIONS
QUOTE
NOSECLVL
PRTCORR
MONOPRC
RANGE
NOUNREF
NOSYNC
NOCRTF
NODUPKEYCHK
NOINZDLT
NOBLK
STDINZ
NODDSFILLER
IMBEDERR
STDTRUNC
NOCHGPOSSGN
NOEVENTF
MONOPIC
NONATIONAL
NOLSPTRALIGN
NOCOMPASBIN
OUTPUT
GENLVL(30)
NOOPTIMIZE
MINIMUM
OBSOLETE
DFRWRT
UNDSPCHR
ACCPDALL
FLAG(0)
LINKPGM
SRTSEQ(*HEX )
LANGID(*JOB RUN )
ENBPFCOL(PEP)
PRFDTA(NOCOL)
CCSID(JOBRUN CCSID CCSID)
DATTIM(1900 40)
THREAD(NOTHREAD)
ARITHMETIC(NOEXTEND)
NTLPADCHAR(NX"0020" NX"3000" NX"3000")
OPTVALUE(NOOPT)
NOGRAPHIC
COBOL Conversion Options in Effect
NOVARCHAR
NODATETIME
NOCVTPICXGRAPHIC
NOFLOAT
NODATE
NOTIME
NOTIMESTAMP
NOCVTTODATE
NOCVTPICNGRAPHIC

```

Figure 12. List of Options in Effect

Source Listing

Figure 13 illustrates a source listing. The statements in the source program are listed exactly as submitted except for program source text that is identified in the REPLACE statement. The replacement text will appear in the source listing. After the page in which the PROGRAM-ID paragraph is listed, all compiler output pages have the program-id name listed in the heading before the system name.

```

5722WDS V5R4M0 060210 LN IBM ILE COBOL CBLGUIDE/SAMPLE ISERIES1 06/02/15 11:18:21 Page 4
STMT PL SEQNBR -A 1 B...2...3...4...5...6...7..IDENTFCN S COPYNAME CHG DATE
  A  B  C  D  E  F
1  000300 IDENTIFICATION DIVISION.
2  000500 PROGRAM-ID. SAMPLE.
3  000600 AUTHOR. PROGRAMMER NAME.
4  000700 INSTALLATION. COBOL DEVELOPMENT CENTRE.
5  000800 DATE-WRITTEN. 02/24/94.
6  000900 DATE-COMPILED. 02/02/05 11:18:21
7  001100 ENVIRONMENT DIVISION.
8  001300 CONFIGURATION SECTION.
9  001400 SOURCE-COMPUTER. IBM-ISERIES
10 001500 OBJECT-COMPUTER. IBM-ISERIES
11 001700 INPUT-OUTPUT SECTION.
12 001800 FILE-CONTROL.
13 001900 SELECT FILE-1 ASSIGN TO DISK-SAMPLE.
15 002100 DATA DIVISION.
16 002300 FILE SECTION.
17 002400 FD FILE-1
    002500 LABEL RECORDS ARE STANDARD
    a
*==>
*a> LNC0848 0 The LABEL clause is syntax checked and ignored. G
    002600 RECORD CONTAINS 20 CHARACTERS
    002700 DATA RECORD IS RECORD-1.
*==>
*a> LNC0848 0 The DATA RECORDS clause is syntax checked and ignored.
18 002800 01 RECORD-1.
19 002900 02 FIELD-A PIC X(20).
20 003100 WORKING-STORAGE SECTION.
21 003200 01 SUBSCRIPT-TYPE TYPEDEF PIC S9(2) COMP-3.
22 003300 01 FILLER.
23 003400 05 KOUNT TYPE SUBSCRIPT-TYPE.
24 003500 05 LETTERS PIC X(26) VALUE "ABCDEFGHJKLMNOPQRSTUVWXYZ".
25 003600 05 ALPHA REDEFINES LETTERS
    003700 PIC X(1) OCCURS 26 TIMES.
26 003800 05 NUMBR TYPE SUBSCRIPT-TYPE.
27 003900 05 DEPENDENTS PIC X(26) VALUE "01234012340123401234012340".
28 004000 05 DEPEND REDEFINES DEPENDENTS
    004100 PIC X(1) OCCURS 26 TIMES.
    004200 COPY WRKRCD.
29 +000100 01 WORK-RECORD. WRKRCD
30 +000200 05 NAME-FIELD PIC X(1). WRKRCD
31 +000300 05 FILLER PIC X(1) VALUE SPACE. WRKRCD
32 +000400 05 RECORD-NO PIC S9(3). WRKRCD
33 +000500 05 FILLER PIC X(1) VALUE SPACE. WRKRCD
34 +000600 05 LOCATION PIC A(3) VALUE "NYC". WRKRCD
35 +000700 05 FILLER PIC X(1) VALUE SPACE. WRKRCD

```

Figure 13. An Example of an ILE COBOL Source Listing (Part 1 of 2)

```

5722WDS V5R4M0 060210 LN  IBM ILE COBOL          CBLGUIDE/SAMPLE      ISERIES1  06/02/15 11:18:21      Page    5
STMT PL SEQNBR -A 1 B.+...2...+...3...+...4...+...5...+...6...+...7..IDENTFCN  S COPYNAME  CHG DATE
36  +000800  05 NO-OF-DEPENDENTS                    WRKRCD
   +000900                    PIC X(2).              WRKRCD
37  +001000  05 FILLER          PIC X(7)  VALUE SPACES.  WRKRCD
38  004300  77  WORKPTR USAGE POINTER.
   004500*****
   004600* THE FOLLOWING PARAGRAPH OPENS THE OUTPUT FILE TO *
   004700* BE CREATED AND INITIALIZES COUNTERS              *
   004800*****
39  004900  PROCEDURE DIVISION.
   005100  STEP-1.
40  005200  OPEN OUTPUT FILE-1.
41  005300  MOVE ZERO TO KOUNT, NUMBR.
   005500*****
   005600* THE FOLLOWING 3 PARAGRAPHS CREATE INTERNALLY THE *
   005700* RECORDS TO BE CONTAINED IN THE FILE, WRITES THEM *
   005800* ON THE DISK, AND DISPLAYS THEM                    *
   005900*****
42  006000  STEP-2.
   006100  ADD 1 TO KOUNT, NUMBR.
43  006200  MOVE ALPHA (KOUNT) TO NAME-FIELD.
44  006300  MOVE DEPEND (KOUNT) TO NO-OF-DEPENDENTS.
45  006400  MOVE NUMBR          TO RECORD-NO.
   006600  STEP-3.
46  006700  DISPLAY WORK-RECORD.
47  006800  WRITE RECORD-1 FROM WORK-RECORD.
   007000  STEP-4.
48  007100  PERFORM STEP-2 THRU STEP-3 UNTIL KOUNT IS EQUAL TO 26.
   007300*****
   007400* THE FOLLOWING PARAGRAPH CLOSES FILE OPENED FOR   *
   007500* OUTPUT AND RE-OPENS IT FOR INPUT                  *
   007600*****
   007700  STEP-5.
49  007800  CLOSE FILE-1.
50  007900  OPEN INPUT FILE-1.
   008100*****
   008200* THE FOLLOWING PARAGRAPHS READ BACK THE FILE AND  *
   008300* SINGLE OUT EMPLOYEES WITH NO DEPENDENTS          *
   008400*****
   008500  STEP-6.
51  008600  READ FILE-1 RECORD INTO WORK-RECORD
52  008700  AT END GO TO STEP-8.
   008900  STEP-7.
53  009000  IF NO-OF-DEPENDENTS IS EQUAL TO "0"
54  009100  MOVE "Z" TO NO-OF-DEPENDENTS.
55  009200  GO TO STEP-6.
   009400  STEP-8.
56  009500  CLOSE FILE-1.
57  009600  STOP RUN.
*==>          a
*=a> LNC0650  0  Blocking/Deblocking for file 'FILE-1' will be performed by compiler-generated code.
          * * * * *  E N D   O F   S O U R C E   * * * * *

```

Figure 13. An Example of an ILE COBOL Source Listing (Part 2 of 2)

Figure 13 on page 67 displays the following fields:

- A** *Compiler-generated statement number:* The numbers appear to the left of the source program listing. These numbers are referenced in all compiler output listings except for FIPS listings. A statement can span several lines, and a line can contain more than one statement. When a sequence of ILE COBOL source programs exist in the input source member, the statement number is reset to 1 at each new compilation unit. The statement number is not reset in a single compilation unit that may contain one or more nested COBOL programs.
- B** *Program nesting level:* The number that appears in this field indicates the degree of nesting of the program.
- C** *Reference number:* The numbers appear to the left of the source statements. The numbers that appear in this field and the column heading (shown as SEQNBR in this listing) are determined by an option specified in the CRTCBMOD or CRTBNDCBL command or in the PROCESS statement, as shown in the following table:

Option	Heading	Origin
NONUMBER	SEQNBR	Source-file sequence numbers
NUMBER	NUMBER	User-supplied sequence numbers
LINENUMBER	LINNBR	Compiler-generated sequence numbers

- D** *Sequence error indicator column:* An S in this column indicates that the line is out of sequence. Sequence checking is performed on the reference number field only if the SEQUENCE option is specified.
- E** *Copyname:* The copyname, as specified in the ILE COBOL COPY statement, is listed here for all records included in the source program by that COPY statement. If the DDS-ALL-FORMATS phrase is used, the name <--ALL-FMTS appears under COPYNAME.
- F** *Change/date field:* The date the line was last modified is listed here.
- G** *Imbedded error:* The first level error message is imbedding in the listing after the line on which the error occurred. The clause, statement, or phrase causing the error is identified.

Verb Usage by Count Listing

Figure 14 shows the alphabetic list that is produced of all verbs used in the source program. A count of how many times each verb was used is also included. This listing is produced when the VBSUM option is specified.

5722WDS V5R4M0 060210 LN IBM ILE COBOL		CBLGUIDE/SAMPLE	ISERIES1	06/02/15 11:18:21	Page	7
Verb Usage By Count						
VERB	COUNT					
ADD	1					
CLOSE	2					
DISPLAY	1					
GOTO	2					
IF	1					
MOVE	5					
OPEN	2					
PERFORM	1					
READ	1					
STOP	1					
WRITE	1					

Figure 14. Verb Usage by Count Listing

Data Division Map

The Data Division map is listed when the MAP option is specified. It contains information about names in the ILE COBOL source program. The minimum number of bytes required for the File Section and Working-Storage Section is given at the end of the Data Division map.

```

5722WDS V5R4M0 060210 LN IBM ILE COBOL          CBLGUIDE/SAMPLE      ISERIES1 06/02/15 11:18:21      Page      8
Data Division Map
  STMT LVL SOURCE NAME          SECTION  DISP  LENGTH  TYPE  ATTRIBUTES
  H   I   J   K   L   M   N   O
  17  FD  FILE-1                FS
                                     FS 00000000 20  GROUP
                                     FS 00000000 20  AN
  18 01  RECORD-1                FS 00000000 20  AN
  19 02  FIELD-A                  WS 00000000 2  PACKED  TYPEDEF
  21 01  SUBSCRIPT-TYPE            WS 00000000 56  GROUP
  22 01  FILLER                    WS 00000000 2  PACKED  TYPE SUBSCRIPT-TYPE
  23 05  KOUNT                      WS 00000002 26  AN  VALUE
  24 05  LETTERS                    WS 00000002 1  AN  REDEFINES LETTERS, DIMENSION(26)
  25 05  ALPHA                      WS 00000028 2  PACKED  TYPE SUBSCRIPT-TYPE
  26 05  NUMBR                      WS 00000030 26  AN  VALUE
  27 05  DEPENDENTS                 WS 00000030 1  AN  REDEFINES DEPENDENTS, DIMENSION(26)
  28 05  DEPEND                    WS 00000000 19  GROUP
  29 01  WORK-RECORD                WS 00000000 1  AN
  30 05  NAME-FIELD                 WS 00000001 1  AN  VALUE
  31 05  FILLER                      WS 00000002 3  ZONED
  32 05  RECORD-NO                  WS 00000005 1  AN  VALUE
  33 05  LOCATION                    WS 00000006 3  A  VALUE
  34 05  FILLER                      WS 00000009 1  AN  VALUE
  35 05  NO-OF-DEPENDENTS            WS 00000010 2  AN
  36 05  FILLER                      WS 00000012 7  AN  VALUE
  37 77  WORKPTR                    WS 00000000 16  POINTR
FILE SECTION uses at least 20 bytes of storage
WORKING-STORAGE SECTION uses at least 91 bytes of storage
***** END OF DATA DIVISION MAP *****

```

Figure 15. Data Division Map

The Data Division map displays the following fields:

- H** *Statement number:* The compiler-generated statement number where the data item was defined is listed for each data item in the Data Division map.
- I** *Level of data item:* The level number of the data item, as specified in the source program, is listed here. Index-names are identified by an *IX* in the level-number and blank fields in the SECTION, DISP, LENGTH, and TYPE fields.
- J** *Source name:* The data name, as specified in the source program, is listed here.
- K** *Section:* The section where the item was defined is shown here through the use of the following codes:
 - FS File Section
 - WS Working-Storage Section
 - LO Local-Storage Section
 - LS Linkage Section
 - SM Sort/Merge Section
 - SR Special Register.
- L** *Displacement:* The offset, in bytes, of the item from the level-01 group item is given here.
- M** *Length:* The decimal length, in bytes, of the item is listed here.
- N** *Type:* The data class type for the item is shown here through the use of the following codes:

Code	Data Class Type
GROUP	Group item
A	Alphabetic
AN	Alphanumeric

Code	Data Class Type
ANE	Alphanumeric edited
DT	Date
TM	Time
TMS	Timestamp
INDEX	Index data item (USAGE INDEX)
BOOLN	Boolean
ZONED	Zoned decimal (external decimal)
PACKED	Packed decimal (internal decimal) (USAGE COMP, COMP-3 or PACKED-DECIMAL)
BINARY	Binary (USAGE COMP-4 or BINARY) Native binary (USAGE COMP-5)
FLOAT	Internal floating-point (USAGE COMP-1 or COMP-2)
EFLOAT	External floating-point (USAGE DISPLAY)
NE	Numeric edited
POINTR	Pointer data item (USAGE POINTER)
PRCPTR	Procedure-pointer data item (USAGE PROCEDURE-POINTER)
G	DBCS
GE	DBCS-edited

0 *Attributes:* The attributes of the item are listed here as follows:

- For files, the following information can be given:
 - DEVICE type
 - ORGANIZATION information
 - ACCESS mode
 - BLOCK CONTAINS information
 - RECORD CONTAINS information
 - LABEL information
 - RERUN is indicated
 - SAME AREA is indicated
 - CODE-SET is indicated
 - SAME RECORD AREA is indicated
 - LINAGE is indicated
 - NULL CAPABLE is indicated.
- For data items, the attributes indicate if the following information was specified for the item:
 - REDEFINES information
 - VALUE
 - JUSTIFIED
 - SYNCHRONIZED
 - BLANK WHEN ZERO
 - SIGN IS LEADING
 - SIGN IS LEADING SEPARATE CHARACTER
 - SIGN IS SEPARATE CHARACTER
 - INDICATORS
 - SIZE
 - TYPEDEF
 - TYPE clause information
 - LOCALE information.

- For table items, the dimensions for the item are listed here in the form DIMENSION (nn). For each dimension, a maximum OCCURS value is given. When a dimension is a variable, it is listed as such, giving the lowest and highest OCCURS values.

FIPS Messages

The FIPS messages, Figure 16, are listed when the FLAGSTD parameter is specified. See “FLAGSTD Parameter” on page 40 for more information about specifying the option for FIPS flagging. Only messages for the requested FIPS subset, optional modules and/or obsolete elements are listed.

Note: The sequence number and column number are given for each time the message is issued.

```

5722WDS V5R4M0 060210 LN  IBM ILE COBOL          CBLGUIDE/SAMPLE      ISERIES1  06/02/15 11:18:21    Page    9
      C O B O L   F I P S   M e s s a g e s
FIPS-ID  DESCRIPTION AND SEQUENCE NUMBERS FLAGGED Q
P
LNC8100  Following items are obsolete language elements.
LNC8102  AUTHOR paragraph.
          000600  10
LNC8103  DATE-COMPILED paragraph.
          000900  10
LNC8104  INSTALLATION paragraph.
          000700  10
LNC8105  DATE-WRITTEN paragraph.
          000800  10
LNC8117  LABEL RECORDS clause.
          002500  12
LNC8177  DATA RECORDS clause.
          002700  12
LNC8200  Following nonconforming standard items valid only at FIPS intermediate level or higher. R
LNC8201  COPY statement.
          004200  8
LNC8500  Following nonconforming nonstandard items are IBM-defined or are IBM extensions.
LNC8504  Assignment-name in ASSIGN clause.
          001900  36
LNC8518  USAGE IS COMPUTATIONAL-3.
          003200  49
LNC8520  USAGE IS POINTER or PROCEDURE-POINTER.
          004300  26
LNC8561  Default library assumed for COPY statement.
          004200  8
LNC8572  SKIP1/2/3 statement.
          000400  13
          001000  13
          001200  13
          001600  13
          002000  13
          002200  13
          003000  13
          004400  13
          005000  13
          005400  13
          006500  13
          006900  13
          007200  13
          008000  13
          008800  13
          009300  13
LNC8616  TYPEDEF clause.
          003200  29
LNC8617  TYPE clause.
          003400  26
          003800  26
30 FIPS violations flagged. S
***** END OF COBOL FIPS MESSAGES *****

```

Figure 16. FIPS Messages

The FIPS messages consist of the following fields:

P *FIPS-ID:* This field lists the FIPS message number.

Q *Description and reference numbers flagged:* This field lists a description of the

condition flagged, followed by a list of the reference numbers from the source program where this condition is found.

The type of reference numbers used, and their names in the heading (shown as SEQUENCE NUMBERS in this listing) are determined by an option specified in the CRTCBMOD or CRTBNDCBL command or in the PROCESS statement, as shown in the following table:

Option	Heading
NONUMBER	DESCRIPTION AND SEQUENCE NUMBERS FLAGGED
NUMBER	DESCRIPTION AND USER-SUPPLIED NUMBERS FLAGGED
LINENUMBER	DESCRIPTION AND LINE NUMBERS FLAGGED

R *Items grouped by level:* These headings subdivide the FIPS messages by level and category.

S *FIPS violations flagged:* The total number of FIPS violations flagged is included at the end of the FIPS listing.

Cross-Reference Listing

Figure 17 shows the cross-reference listing, which is produced when the XREF option is specified. It provides a list of all data references, procedure-name references, and program-name references, by statement number, within the source program.

```

5722WDS V5R4M0 060210 LN  IBM ILE COBOL          CBLGUIDE/SAMPLE      ISERIES1  06/02/15 11:18:21      Page    10
                          C r o s s   R e f e r e n c e

Data References:
Data-type is indicated by the letter following a data-name definition.
These letters and their meanings are:
  E = EXTERNAL
  G = GLOBAL
  X = EXTERNAL and GLOBAL
DATA NAMES          DEFINED  REFERENCES (* = CHANGED)
T                U          V
ALPHA                25      43
DEPEND               28      44
DEPENDENTS           27      28
FIELD-A              19
FILE-1               17      40      49      50      51      56
KOUNT                23      41*     42*     43      44      48
LETTERS              24      25
LOCATION               34
NAME-FIELD           30      43*
NO-OF-DEPENDENTS    36      44*     53      54*
NUMBR                26      41*     42*     45
RECORD-NO            32      45*
RECORD-1             18      47*
WORK-RECORD          29      46      47      51*
WORKPTR              38
TYPE NAMES          DEFINED  REFERENCES (* = CHANGED)
SUBSCRIPT-TYPE      21      23      26

Procedure References:
Context usage is indicated by the letter following a procedure-name reference.
These letters and their meanings are:
  A = ALTER (procedure-name)
  D = GO TO (procedure-name) DEPENDING ON
  E = End of range of (PERFORM) through (procedure-name)
  G = GO TO (procedure-name)
  P = PERFORM (procedure-name)
  T = (ALTER) TO PROCEED TO (procedure-name)
PROCEDURE NAMES     DEFINED  REFERENCES
STEP-1              39
STEP-2              41      48P
STEP-3              45      48E
STEP-4              47
STEP-5              48
STEP-6              50      55G
STEP-7              52
STEP-8              55      52G

Program References:
Program-type of the external program is indicated by the word in a program-name definition.
These words and their meanings are:
  EPGM = a program object that is to be dynamically linked
  BPRC = a COBOL program or a C function or an RPG program that is to be bound
  SYS  = a system program
PROGRAM NAMES       DEFINED  REFERENCES
SAMPLE              2
***** END OF CROSS REFERENCE *****

```

Figure 17. Cross-Reference Listing

The cross-reference listing displays the following fields:

- T** *Names field:* The data name, type name, procedure name, or program name referenced is listed here. The names are listed alphabetically. Program names may be qualified by a library name.
- U** *Defined field:* The statement number where the name was defined within the source program is listed here.
- V** *References field:* All statement numbers are listed in the same sequence as the name is referenced in the source program. An * following a statement number indicates that the item was modified in that statement.

Messages

Figure 18 shows the messages that are generated during program compilation.

```

5722WDS V5R4M0 060210 LN  IBM ILE COBOL          CBLGUIDE/SAMPLE      ISERIES1  06/02/15 11:18:21      Page    12
                                     M e s s a g e s
      STMT
      W  Y  X
*  17 MSGID: LNC0848 SEVERITY: 0 SEQNBR: 002500
      Message . . . . : The LABEL clause is syntax checked and ignored.  Z
*  17 MSGID: LNC0848 SEVERITY: 0 SEQNBR: 002700
      Message . . . . : The DATA RECORDS clause is syntax checked and
      ignored.
*  57 MSGID: LNC0650 SEVERITY: 0 SEQNBR: 009600
      Message . . . . : Blocking/Deblocking for file 'FILE-1' will be
      performed by compiler-generated code.
                                     Message Summary
Message totals: AA
Information (00-04) . . . . . : 3
Warning (05-19) . . . . . : 0
Error (20-29) . . . . . : 0
Severe (30-39) . . . . . : 0
Terminal (40-99) . . . . . : 0
-----
Total . . . . . : 3
* * * * *  E N D   O F   M E S S A G E S   * * * * *

Statistics: BB
Source records read . . . . . : 96
Copy records read . . . . . : 10
Copy members processed . . . . . : 1
Sequence errors . . . . . : 0
Highest severity message issued . . : 0
LNC0901 0 Program SAMPLE created in library CBLGUIDE on 00/08/17 at 11:18:23.
* * * * *  E N D   O F   C O M P I L A T I O N   * * * * *

```

Figure 18. Diagnostic Messages

The fields displayed are:

- W** *Statement number:* This field lists the compiler-generated statement number associated with the statement in the source program for which the message was issued.¹
- X** *Reference number:* The reference number is issued here.¹ The numbers that appear in this field and the column heading (shown here as SEQNBR) are determined by an option specified in the CRTCBLMOD or CRTBNDCBL command or in the PROCESS statement, as shown in the following table:

	Heading	Origin
NONUMBER	SEQNBR	Source-file sequence numbers
NUMBER	NUMBER	User-supplied sequence numbers
LINENUMBER	LINNBR	Compiler-generated sequence numbers

When a message is issued for a record from a copy file, the number is preceded by a +.

- Y** *MSGID and Severity Level:* These fields contain the message number and its associated severity level. Severity levels are defined as follows:
 - 00 Informational
 - 10 Warning
 - 20 Error
 - 30 Severe Error
 - 40 Unrecoverable (usually a user error)
 - 50 Unrecoverable (usually a compiler error)
- Z** *Message:* The message identifies the condition and indicates the action taken by the compiler.

1. The statement number and the reference number do not appear on certain messages that reference missing items. For example, if the PROGRAM-ID paragraph is missing, message LNC0031 appears on the listing with no statement or reference number listed.

AA *Message statistics:* This field lists the total number of messages and the number of messages by severity level.

The totals listed are the number of messages generated for each severity by the compiler and are not always the number listed. For example, if FLAG(10) is specified, no messages of severity less than 10 are listed. The counts, however, do indicate the number that would have been printed if they had not been suppressed.

BB *Compiler statistics:* This field lists the total number of source records read, copy records read, copy members processed, sequence errors encountered, and the highest severity message issued.

Chapter 4. Creating a Program Object

This section provides you with the information on:

- How to create a program object by binding one or more module objects together
- How to create a program object from ILE COBOL source statements
- The CRTBNDCBL command and its parameters
- How to read a binder listing
- How to create program objects using one or more module objects, service programs, and ILE binding directories.

Use WebSphere Development Studio for i5/OS. This is the recommended method and documentation about creating a program object appears in that product's online help.

See Using the application development tools in the client product for information about getting started with the client tools.

Definition of a Program Object

A **program object** is a runnable system object of type *PGM. For ILE COBOL, the name of the program object is determined by the CRTBNDCBL command, CRTPGM command, or the PROGRAM-ID paragraph of the outermost COBOL source program. The process that creates a program object from one or more module objects and referenced service programs is known as **binding**. One or more module objects are created by the CRTCBLMOD command, or are temporarily created by the CRTBNDCBL command before it creates one or more bound program objects. Binding is a process that takes module objects produced by the CRTCBLMOD or CRTBNDCBL command and combines them to create a runnable bound program object or service program.

When a program object is created, only ILE procedures in those module objects containing debug data can be debugged by the ILE source debugger. The debug data does not affect the performance of the running program object. Debug data does increase the size of the generated program object.

A program object is run by using a dynamic program call. The entry point to the program object is the PEP.

The Binding Process

The binding process improves runtime performance as program objects are able to use static procedure calls to routines already bound as part of a program object. Dynamic program calls are not needed to access the routines. Individual module objects created by different ILE HLL compilers can be bound together in the same program object allowing for a routine to be coded in the most appropriate language and then bound to a program object that requires it.

Previously compiled module objects can be bound in various sequences to create new runnable program objects. The previously compiled module objects can be re-used to create new runnable program objects without having to recompile the original source program. This allows a module object to be re-used as needed.

Instead of re-creating programs each time a module object changes, service programs may be used. Common routines can be created as service programs. If the routine changes but its interface does not, or if only upward compatible changes are made to the interface, then the change can be incorporated by re-creating the service program. The program objects and service programs that use these common routines do not have to be re-created.

There are two paths that allow you to create a program object using the binding process. The diagram below shows the two available paths:

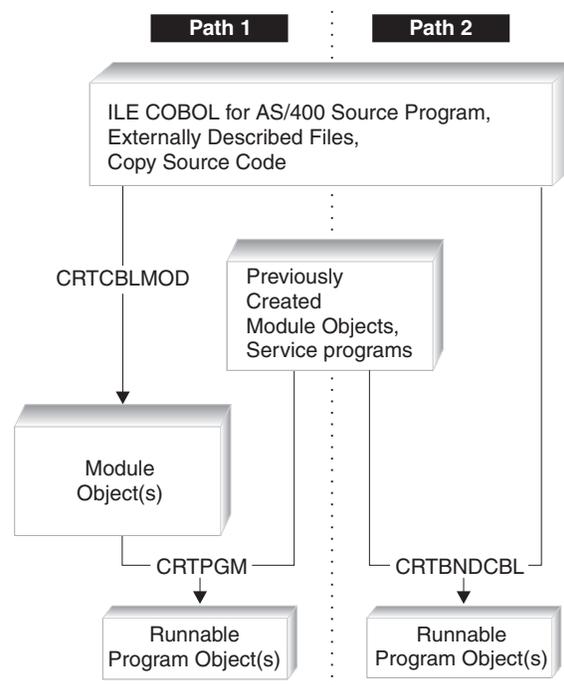


Figure 19. Two paths for creating a Program Object

These two paths both use the binding process. The Create Program (CRTPGM) command creates a program object from module objects created from ILE COBOL source programs using the Create COBOL Module (CRTCBMOD) command, and zero or more service programs.

Note: Module objects created using the Create RPG Module (CRTRPGMOD), Create C Module (CRTCMOD), and Create CL Module (CRTCLMOD) commands can also be bound using Create Program (CRTPGM).

The Create Bound COBOL (CRTBND CBL) command creates one or more temporary module objects from one or more ILE COBOL compilation units, and then creates one or more program objects. Once the program object is created, CRTBND CBL deletes the module object(s) it created. This command performs the combined tasks of the Create COBOL Module (CRTCBMOD) and Create Program (CRTPGM) commands in a single step. Previously created module objects and service programs can be bound using a binding directory. However, the input source member bound using this step must be the PEP module.

Note: Every program object only recognizes one PEP (and one UEP). If you bind several module objects together to create a program object using CRTPGM and each of these module objects has a PEP, you must specify in the

ENTMOD parameter, which module object's PEP is to be used as the PEP for the program object. Also, the order in which module objects and service programs are specified in the CRTPGM command affects the way symbols are resolved during the binding process. Therefore, it is important that you understand how binding is performed. For more information on the binding process, refer to the *ILE Concepts* book.

A **binding directory** contains the names of modules and service programs that you may need when creating an ILE program or service program. Modules or service programs listed in a binding directory are used when they provide an export that can satisfy any currently unresolved import requests. A binding directory is a system object that is identified to the system by the symbol *BNDDIR.

Binding directories are optional. The reasons for using binding directories are convenience and program size.

- They offer a convenient method of packaging the modules or service programs that you may need when creating your own ILE program or service program. For example, one binding directory may contain all the modules and service programs that provide math functions. If you want to use some of those functions, you specify only the one binding directory, not each module or service program you use.
- Binding directories can reduce program size because you do not specify modules or service programs that do not get used.

Very few restrictions are placed on the entries in a binding directory. The name of a module or service program can be added to a binding directory even if that object does not yet exist.

For a list of CL commands used with binding directories, see the *ILE Concepts* manual. Characteristics of a *BNDDIR object are:

- Convenient method of grouping the names of service programs and modules that may be needed to create an ILE program or service program.
- Because binding directory entries are just names, the objects list does not have to exist yet on the system.
- The only valid library names are *LIBL or a specific library.
- The objects in the list are optional. The named objects are used only if any unresolved imports exist, and if the named object provides an export to satisfy the unresolved import request.

Using the Create Program (CRTPGM) Command

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

Note: In order to use the CRTPGM command, you must have authority to use the command and the modules required must first have been created using the CRTCLMOD, CRTCMOD, CRTRPGMOD, or CRTCLMOD commands.

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

1. Establish a program name.

2. Identify the module object(s), and if required, service program(s) you want to bind into a program object.
3. Identify any binding directories you intend to use. Implicit references to binding directories, for ILE COBOL runtime service programs and ILE bindable APIs, are made from module objects created by CRTCBMOD and CRTBNDCBL.
4. Identify which module object's PEP will be used as the PEP for the program object being created. Specify this module object in the ENTMOD parameter of CRTPGM.

If you Specify ENTMOD(*FIRST) instead of explicitly identifying a module object in the ENTMOD parameter, then the order in which the binding occurs is important in deciding which module object has the PEP for the program object being created. The module objects that you list in the MODULE parameter, or those located through a binding directory, may contain one or more PEPs when only one can be used. The order in which binding occurs is also important for other reasons such as the resolution of symbols. For further information on binding, refer to the *ILE Concepts* book.

5. Establish whether the command will allow duplicate procedures and or variable names.
You may be binding module objects into a program object that each define the same variable names and procedure names in multiple different ways.
6. Identify the activation group in which the program is run. Refer to "Activation and Activation Groups" on page 211 for description of activation groups.

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 parameters.

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 and APIs*
section of the *Programming* category in the **i5/OS Information Center** at this Web
site -<http://www.ibm.com/systems/i/infocenter/>.

Table 5. Parameters for CRTPGM Command and their Default Values

Parameter Group	Parameter(Default Value)
Identification	PGM(<i>library name/program name</i>) MODULE(*PGM)
Program access	ENTMOD(*FIRST)
Binding	BNDSRVPGM(*NONE) BNDDIR(*NONE)
Run time	ACTGRP(*NEW)
Miscellaneous	OPTION(*GEN *NODUPPROC *NODUPVAR *WARN *RSLVREF) DETAIL(*NONE) ALWUPD(*YES) ALWRINZ(*NO) REPLACE(*YES) AUT(*LIBCRTAUT) TEXT(*ENTMODTXT) TGTRLS(*CURRENT) USRPRF(*USER) ALWLIBUPD(*NO)

Once you have entered the CRTPGM command, the program object is created as follows:

1. Listed module objects are copied into what will become the program object.
2. The module objects containing the PEP is identified and the first import in this module is located.
3. The module objects are checked in the order in which they are listed and the first import is matched with a module export.
4. The first module object is returned to and the next import is located.
5. All imports in the first module object are resolved.
6. The next module object is continued to and all imports are resolved.
7. All imports in each subsequent module object are resolved until all of the imports have been resolved.
8. If OPTION(*RSLVREF) is specified and any imports cannot be resolved with an export, the binding process terminates without creating a program object. If OPTION(*UNRSLVREF) is specified then all imports do not need to be resolved to exports for the program object to be created. If the program object uses one of these unresolved imports at run time, a MCH4439 exception message is issued.
9. Once all the imports have been resolved, the binding process completes and the program object is created.

Example of Binding Multiple Modules to Create a Program Object

This example shows you how to use the CRTPGM command to bind the module objects A, B, and C into the program object named ABC. The module object, which has the PEP and UEP for the program object, is the module object named on the ENTMOD parameter.

All external references should be resolved for the CRTPGM command to bind multiple modules into a program object. References to the ILE COBOL runtime functions are resolved as they are automatically bound into any program object containing ILE COBOL module objects.

1. To bind several module objects to create a program object, type:
CRTPGM PGM(ABC) MODULE(A B C) ENTMOD(*FIRST) DETAIL(*FULL)

and press Enter.

Using the Create Bound COBOL (CRTBNDCBL) Command

The Create Bound COBOL (CRTBNDCBL) command creates one or more program objects from a single ILE COBOL source file member in a single step. This command starts the ILE COBOL compiler and creates temporary module objects which it then binds into one or more program objects of type *PGM.

Unlike the CRTPGM command, when you use the CRTBNDCBL command, you do not need a separate preceding step of creating one or more module objects using the CRTCBMOD command. The CRTBNDCBL command performs the combined tasks of the Create COBOL Module (CRTCBMOD) and Create Program (CRTPGM) commands by creating temporary module objects from the source file member, and then creating one or more program objects. Once the program object(s) is created, CRTBNDCBL deletes the module objects it created.

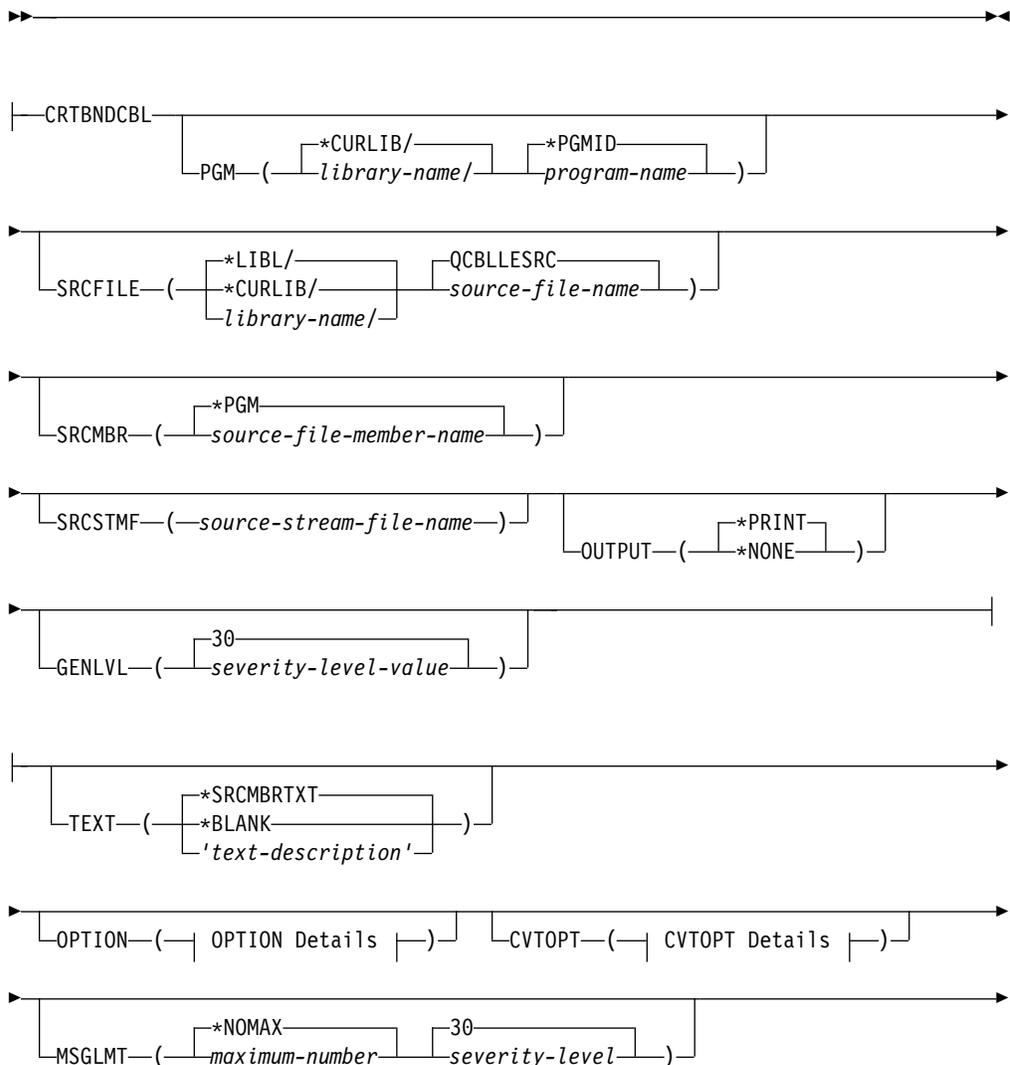
Note: If you want to retain the module objects, use CRTCBMOD instead of CRTBNDCBL. If you use CRTCBMOD, you will have to use CRTPGM to bind the module objects into one or more program objects. Also, if you want to use options of CRTPGM other than those implied by CRTBNDCBL, use CRTCBMOD and CRTPGM.

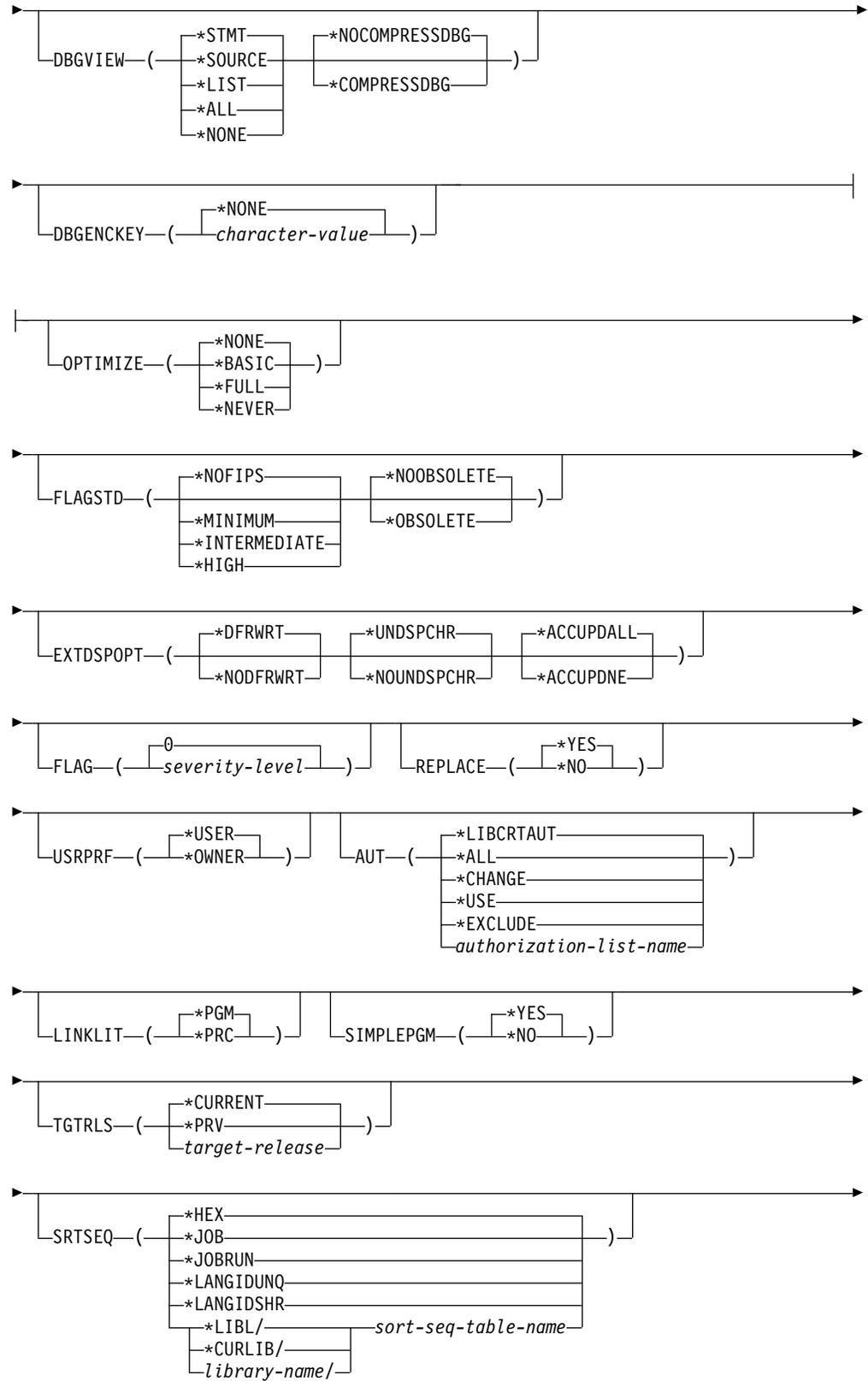
Using Prompt Displays with the CRTBNDCBL Command

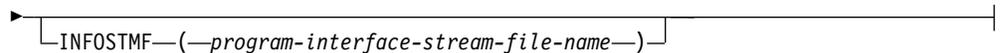
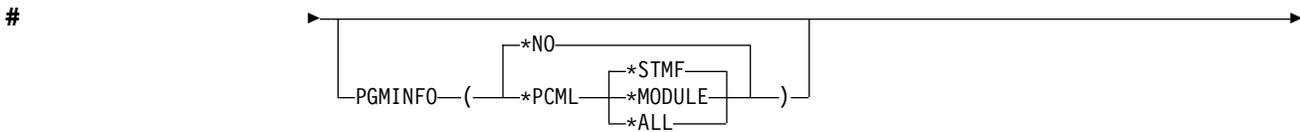
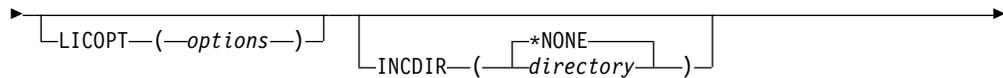
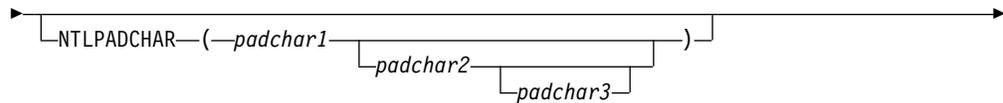
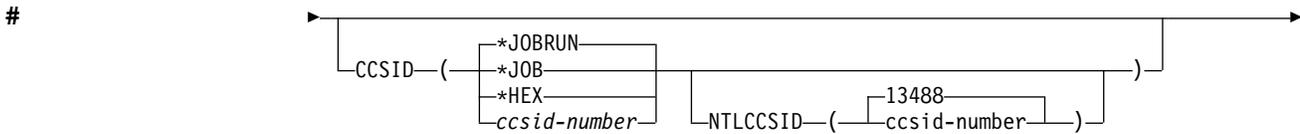
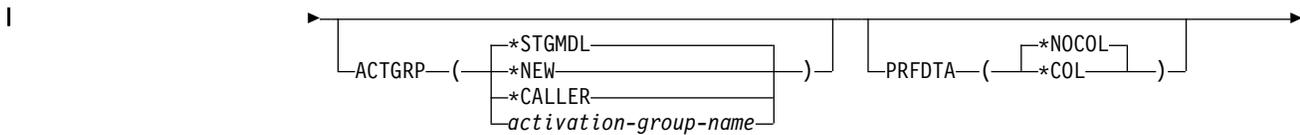
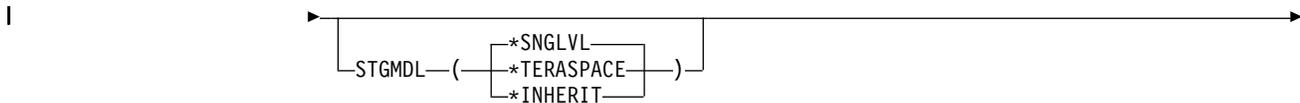
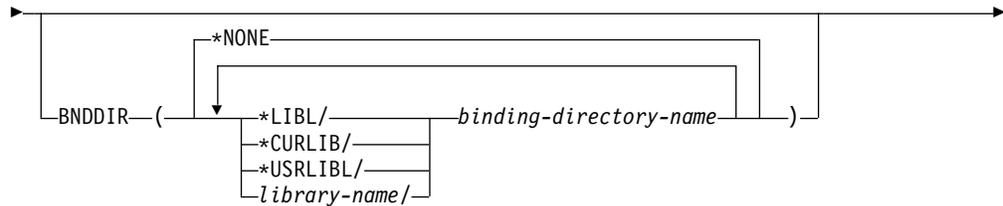
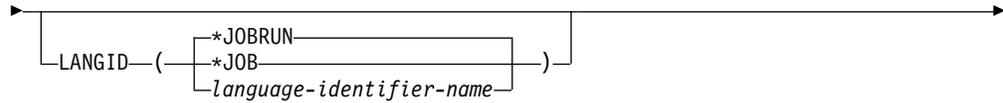
If you require prompting, type CRTBNDCBL and press F4 (Prompt). The CRTBNDCBL display which lists parameters and supplies default values appears. If you have already supplied parameters before you request prompting, the display appears with parameters values filled in. If you require help, type CRTBNDCBL and press F1 (Help).

Syntax for the CRTBNDCBL Command

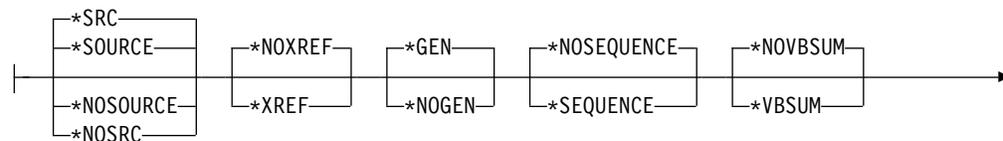
CRTBNDCBL Command - Format

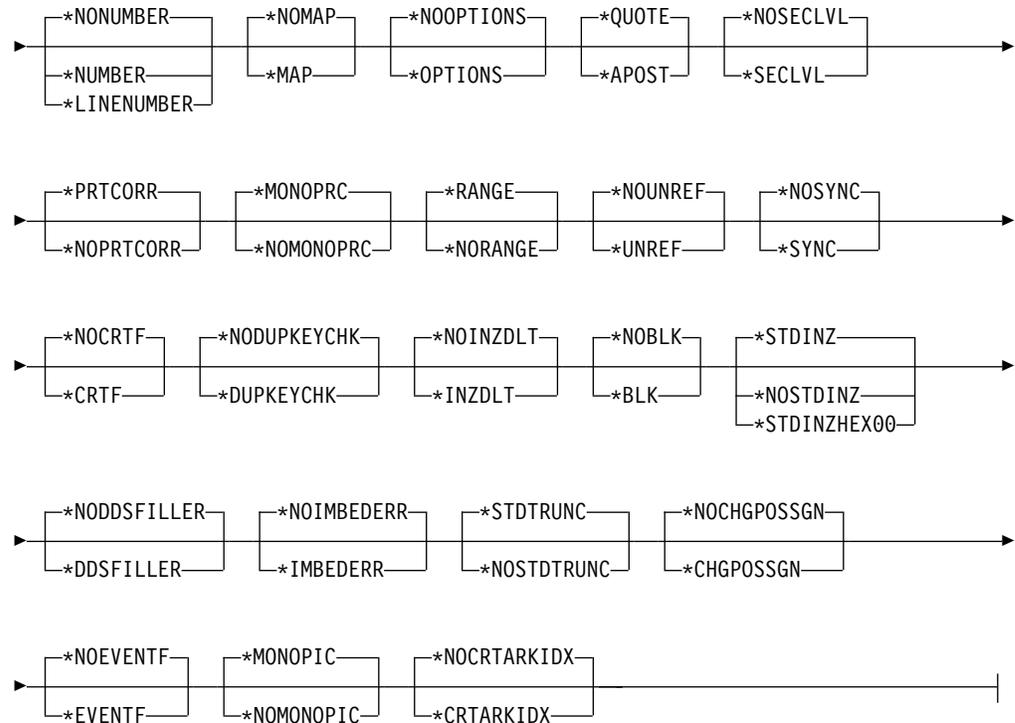




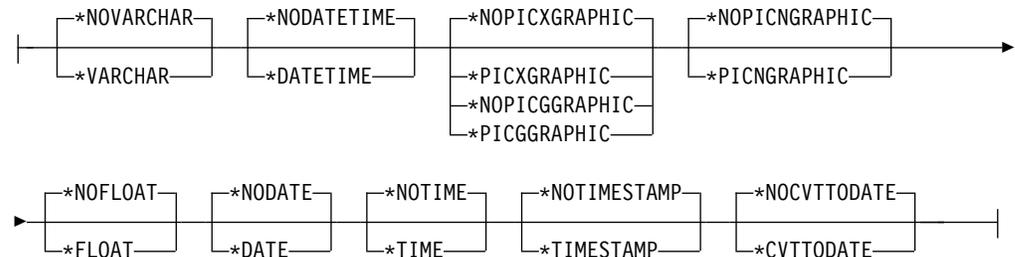


OPTION Details:





CVTOPT Details:



Parameters of the CRTBNDCBL Command

Nearly all of the parameters of CRTBNDCBL are identical to those shown for CRTCBMOD previously. Only the differences between the two commands will be described here.

CRTBNDCBL differs from CRTCBMOD in the following ways:

- The introduction of the PGM parameter instead of the MODULE parameter
- The SRCMBR parameter uses *PGM option (instead of *MODULE option)
- A different use of the REPLACE parameter
- The introduction of the USRPRF parameter
- The introduction of the SIMPLEPGM parameter
- A restriction on the objects affected by the PRFDTA parameter
- The introduction of the BNDDIR parameter
- The introduction of the ACTGRP parameter.
- The default value for the STGMDL parameter.

#

PGM Parameter:

Specifies the program name and library name for the program object you are creating. The program name and library name must conform to i5/OS naming conventions. The possible values are:

***PGMID**

The name for the compiled program object is taken from the PROGRAM-ID paragraph in the ILE COBOL source program. When SIMPLEPGM(*NO) is specified, the name of the compiled program object is taken from the PROGRAM-ID paragraph of the first ILE COBOL source program in the *sequence of source programs* (multiple compilation units in a single source file member).

program-name

Enter a name to identify the compiled ILE COBOL program. If you specify a program name for this parameter, and compile a sequence of source programs and if SIMPLEPGM(*YES) is specified, the first program object in the sequence uses this name; any other program objects use the name specified in the PROGRAM-ID paragraph in the corresponding ILE COBOL source program.

The possible library values are:

***CURLIB**

The created program object is stored in the current library. If you have not assigned a library as the current library, QGPL is used.

library-name

Enter the name of the library where the created program object is to be stored.

REPLACE Parameter:

Specifies if a new program object is created when a program object of the same name in the specified or implied library already exists. The intermediate module objects created during the processing of the CRTBNDCBL command are not subject to the REPLACE specifications, and have an implied REPLACE(*NO) against the QTEMP library. The intermediate module objects are deleted once the CRTBNDCBL command has completed processing. The possible values of the REPLACE parameter are:

***YES**

A new program object is created and it replaces any existing program object of the same name in the specified or implied library. The existing program object of the same name in the specified or implied library is moved to library QRPLOBJ.

***NO**

A new program object is not created if a program object of the same name already exists in the specified library. The existing program object is not replaced, a message is displayed, and compilation stops.

USRPRF Parameter:

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 object and control which objects can be used by the program object (including the authority the program object has for each object). This parameter is not updated if the program object already exists. To change the value of USRPRF, delete the program object and recompile using the correct value (or, if the constituent *MODULE objects exist, you may choose to invoke the CRTPGM command).

The possible values are:

***USER**

The user profile of the program user is to be used when the program object is run.

***OWNER**

The user profiles of both the owner and user of the program object are to be used when the program object is run. The collective sets of object authorities in both owner and user profiles are to be used to find and access objects during the running of the program object. Any objects that are created when the program is run are owned by the user of the program.

SIMPLEPGM Parameter:

Specifies if a program object is created for each of the compilation units in the sequence of source programs. This option is meaningful only if the input source member to this command contains a sequence of source programs which generate multiple module objects. If this option is specified but the input source member does not have a sequence of source programs, then the option is ignored. The possible values are:

***YES**

A program object is created for each of the compilation units in the sequence of source programs. If REPLACE(*NO) is specified and a program object of the same name already exists for a compilation unit in the sequence of source programs, that program object is not replaced and compilation continues at the next compilation unit.

***NO**

A single program object is created from all the compilation units in the sequence, with the first compilation unit representing the Program Entry. With SIMPLEPGM(*NO) specified, if one source program in a sequence of source programs fails to generate a module object then all subsequent source programs in the sequence will also fail to generate module objects.

PRFDTA Parameter:

This parameter works the same as described on page “PRFDTA Parameter” on page 45, with the following note.

Note: If you use the BNDDIR parameter to bind additional modules and service programs, these additional objects are not affected when *COL or *NOCOL is specified for the program. The program profiling data attribute for a module is set when the module is created.

BNDDIR Parameter:

Specifies the list of binding directories that are used in symbol resolution. Can specify up to 50 binding directories.

***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. This is the default.

***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.

STGMDL Parameter:

Specifies the storage model attribute of the program.

***SINGLVL**

The program is created with single-level storage model. When a single-level storage model program is activated and run, it is supplied single-level storage for automatic and static storage. A single-level storage program runs only in a single-level storage activation group.

***TERASPACE**

The program is created with teraspace storage model. When a teraspace storage model program is activated and run, it is supplied teraspace storage for automatic and static storage. A teraspace storage program runs only in a teraspace storage activation group.

***INHERIT**

The program is created with inherit storage model. When activated, the program adopts the storage model of the activation group into which it is activated. An equivalent view is that it inherits the storage model of its caller. When the *INHERIT storage model is selected, *CALLER must be specified for the Activation group (ACTGRP) parameter.

ACTGRP Parameter:

Specifies the activation group this program is associated with when it is called.

***STGMDL**

If STGMDL(*TERASPACE) is specified, the program will be activated into the QILETS activation group when it is called. Otherwise, this program will be activated into the QILE activation group when it is called. This is the default.

***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.

Invoking CRTPGM Implicitly from CRTBNDCBL

There are implied default values to be used in the CRTPGM step implicitly invoked from the CRTBNDCBL command. They are described in the description that follows.

The parameters used in CRTPGM when it is invoked from CRTBNDCBL are as follows:

PGM

When SIMPLEPGM(*YES) is specified or implied, CRTPGM is invoked for each compilation unit in the sequence of source programs. The program name specified in the PROGRAM-ID paragraph in the corresponding outermost ILE COBOL source program of each compilation unit is used with the PGM parameter for CRTPGM each time it is invoked. An individually bound program object is created for each compilation unit.

When SIMPLEPGM(*NO) is specified, CRTPGM is invoked only one time against all of the compilation units in the sequence of source programs at once. Only the program name specified in the PROGRAM-ID paragraph in the corresponding outermost ILE COBOL source program for the first compilation unit in the sequence of the source programs is used with the PGM parameter for CRTPGM when it is invoked. All of the compilation units are bound together to create one program object.

MODULE

When SIMPLEPGM(*YES) is specified or implied, the name of the module created in QTEMP for each compilation unit is used with the MODULE parameter for CRTPGM each time it is invoked.

When SIMPLEPGM(*NO) is specified, all the names of the modules created in QTEMP for the compilation units are listed in the MODULE parameter for the CRTPGM command when it is invoked.

BNDDIR

Specifies the list of binding directories that are used in symbol resolution.

When *NONE (the default) is specified, no binding directory is used.

When *binding-directory-name* is specified, the name of the binding directory you specify is 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. This is the default.

***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

Activation group specified is used

REPLACE

The REPLACE option specified in the CRTBNDCBL command is used

USRPRF

The USRPRF option specified in the CRTBNDCBL command is used

AUT

The AUT option specified in the CRTBNDCBL command is used

TEXT

The TEXT option specified in the CRTBNDCBL command is used

TGTRLS

The TGTRLS option specified in the CRTBNDCBL command is used

STGMDL

The STGMDL option specified in the CRTBNDCBL command is used

The default values are used for all of the remaining parameters of CRTPGM when
it is invoked from the CRTBNDCBL command. For a description of these default
values Refer to the CRTPGM command in the *CL and APIs* section of the
Programming category in the **i5/OS Information Center** at this Web site
-http://www.ibm.com/systems/i/infocenter/.

Example of Binding One Module Object to Create a Program Object

This example shows you how to create a program object from a module using the CRTBNDCBL command.

1. To create a program object, type:

```
CRTBNDCBL PGM(MYLIB/XMPLE1)
SRCFILE(MYLIB/QCBLLESRC) SRCMBR(XMPLE1)
OUTPUT(*PRINT)
TEXT('ILE COBOL Program')
CVTOPT(*FLOAT)
```

and press Enter.

The CRTBNDCBL command creates the program XMPLE1 in MYLIB. The OUTPUT(*PRINT) option specifies that you want a compilation listing.

2. Type one of the following CL commands to view the listing.

Note: In order to view a compiler listing you must have authority to use the commands listed below.

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

Specifying National Language Sort Sequence in CRTBNDCBL

At the time that you compile your COBOL source program, you can explicitly specify the collating sequence that the program will use when it is run, or you can specify how the collating sequence is to be determined when the program is run.

You specify the collating sequence through CRTBNDCBL in the same manner as you would through CRTCBMOD. For a full description of how to specify an NLS collating sequence, refer to “Specifying National Language Sort Sequence in CRTCBMOD” on page 49.

Reading 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 spool file for the job you use to enter the CRTPGM command. The command default, *NONE, is to not produce this information but you can select to generate this as printed output at three levels of detail as a value in the DETAIL parameter:

- *BASIC
- *EXTENDED

- *FULL

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

Table 6. Sections of the Binder Listing based on DETAIL Parameter

Section Name	*NONE	*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 to give feedback on what the binder encountered in the process.

A Sample Binder Listing

The following sample listings illustrate the binder listing produced using the CRTPGM command. References to the figures are made throughout the following text. These references are indexed by the reverse printing of letters on a black background, for example (**Z**). The reverse letters in the text correspond to the letter found in the figures.

Command Option Summary

The Command Option Summary is produced whenever a binder listing is requested. It shows what options were used when the ILE program was created. You may want to store this information description of the command for some future reference when you need to create the program again. Figure 20 on page 92 shows you the command option summary listing.

```

                                Create Program
5722SS1 V5R4M0 060210                                CBLGUIDE/EXTLFL  ISERIES1 06/02/15 13:14:03 Page 1
Program . . . . . : EXTLFL
Library . . . . . : CBLGUIDE
Program entry procedure module . . . . . : *FIRST
Library . . . . . :
Activation group . . . . . : *NEW
Creation options . . . . . : *GEN *NODUPPROC *NODUPVAR *WARN *RSLVREF
Listing detail . . . . . : *FULL
Allow update . . . . . : *YES
Allow bound *SRVPGM library name update . . . . . : *NO
User profile . . . . . : *USER
Replace existing program . . . . . : *YES
Authority . . . . . : *LIBCRTAUT
Target release . . . . . : *CURRENT
Allow reinitialization . . . . . : *NO
Storage model . . . . . : *SINGLVL
Interprocedural analysis . . . . . : *NO
IPA control file . . . . . : *NONE
IPA replace IL data . . . . . : *NO
Text . . . . . : *ENTMODTXT
Module Library Module Library Module Library Module Library
EXTLFL CBLGUIDE
Service Service Service Service
Program Library Program Library Program Library Program Library
*NONE
Binding Binding Binding Binding
Directory Library Directory Library Directory Library Directory Library
*NONE

```

Figure 20. CRTPGM Command Option Summary Listing

Extended Summary Table

The Extended Summary Table is provided if *EXTENDED or *FULL is supplied. It contains statistical information that provides a general view of the imports and exports that the binder resolved. Figure 21 shows the layout of the Extended Summary Table.

```

                                Create Program
5722SS1 V5R4M0 060210                                CBLGUIDE/EXTLFL  ISERIES1 06/02/15 13:14:03 Page 3
Valid definitions . . . . . : 341 A
Strong . . . . . : 340
Weak . . . . . : 1
Resolved references . . . . . : 16 B
To strong definitions . . . . . : 15
To weak definitions . . . . . : 1
***** END OF EXTENDED SUMMARY TABLE *****

```

Figure 21. CRTPGM Listing - Extended Summary Table

The Extended Summary Table provides statistical information on the following items:

- A** *Valid definitions:* This field provides the number of named variables or procedures available for exporting. The definitions are further categorized as **strong definitions** or **weak definitions**. For strong definitions, storage is allocated for the variable or procedure. For weak definitions, storage is referenced for the variable or procedure.

In ILE COBOL, the outermost COBOL source program and its associated CANCEL procedure will have strong definitions. EXTERNAL data and EXTERNAL files will have weak definitions. CALL, CANCEL, and SET ENTRY to an external static procedure will have module imports that are strong definitions. References to EXTERNAL data and EXTERNAL files will have module imports that are weak definitions.

B *Resolved references:* This field provides the number of imports that are matched with inter-module exports.

The usage counts shown in Figure 21 on page 92 are in decimal form.

Brief Summary Table

This Brief Summary Table, available when *BASIC, *EXTENDED, or *FULL is specified, provides information that reflects what was found to be in error during the binding process. Figure 22 shows the layout of the Brief Summary Table.

```

5722SS1 V5R4M0 060210                                Create Program                                Page 4
                                                         CBLGUIDE/EXTLFL  ISERIES1 06/02/15 13:14:03
                                                         Brief Summary Table
Program entry procedures . . . . . : 1 C
Symbol F      Type G      Library H      Object I      Bound J      Identifier K
*MODULE CBLGUIDE  EXTLFL *YES
Multiple strong definitions . . . . . : 0 D
Unresolved references . . . . . : 0 E
***** END OF BRIEF SUMMARY TABLE *****

```

Figure 22. CRTPGM Listing - Brief Summary Table

The table consists of three lists with the number of entries in each of the following categories:

- C** *Program entry procedures:* The number of procedures that get control from a calling program.
- D** *Multiple strong definitions:* The number of module export procedures with the same name. This should be 0.
- E** *Unresolved references:* The number of imported procedures or variables for which no export was located. This should be 0.
- F** *Symbol #:* The Symbol number is from the Binder Information Listing shown in “Binding Information Listing” on page 94. If *BASIC is specified for the DETAIL parameter, this area is blank.
- G** *Type:* The type of the object containing the identifier is shown in the Type field.
- H** *Library:* The name of the library containing the object is shown in the Library field.
- I** *Object:* The name of the object which has the program entry procedure, unresolved reference, or strong definition is shown in the Object field.
- J** *Bound:* If this field shows a value of *YES for a module object, the module object is bound by copy. If this field shows a value of *YES for a program, the program is bound by reference. If this field shows a value of *NO for a module object or program, that object is not included in the bind.
- K** *Identifier:* The name of the procedure or variable from module source is shown in the Identifier field.

In this example, the total number of program entry procedures, unresolved references, or multiple strong definitions are 1, 0, 0 respectively. The usage counts shown in Figure 22 are in decimal form.

Binding Information Listing

This listing, which provides much more detailed information about the binding process, is available if *EXTENDED or *FULL is specified. Figure 23 shows the layout of the Binder Information Listing.

```

5722SS1 V5R4M0 060210                                Create Program                                Page 5
                                                       CBLGUIDE/EXTLFL  ISERIES1 06/02/15 13:14:03
                                                       Binder Information Listing

Module . . . . . : EXTLFL L
Library . . . . . : CBLGUIDE
Bound . . . . . : *YES
Change date/time . . . . . : 00/08/15 13:11:40
Teraspace storage enabled : *YES
Storage model . . . . . : *SINGLVL

  Number  Symbol  Ref  Identifier  Type  Scope  Export  Key
  M      N      O      P      Q      R      S      T
00000001 Def      EF1_ffd      Data  Module  Weak      190
****
00000002 Def      EF1MAIN      Proc  Module  Strong
00000003 Def      EF1MAIN_reset Proc  Module  Strong
00000004 Ref      000000A7 _Qln_rut      Data
00000005 Ref      00000001 EF1_ffd      Data
00000006 Ref      000000C6 _Qln_cancel_handler Proc
00000007 Ref      000000D6 _Qln_cancel_handler_pep Proc
00000008 Ref      000000A8 _Qln_init_mod Proc
00000009 Ref      000000A9 _Qln_init_mod_bdry Proc
0000000A Ref      000000AA _Qln_init_oprg Proc
0000000B Ref      000000B9 _Qln_recurse_msg Proc
0000000C Ref      00000022 _Qln_disp_norm Proc
0000000D Ref      000000BE _Qln_stop_prg Proc
0000000E Ref      000000BB _Qln_cancel_msg Proc
0000000F Ref      000000BD _Qln_fc_hdlr Proc
00000010 Ref      0000012A Q LE leDefaultEh Proc
00000011 Ref      00000131 Q LE leBdyCh Proc
00000012 Ref      00000161 Q LE leActivationInit Proc
00000013 Ref      00000132 Q LE leBdyEpilog Proc

Service program . . . . . : QLNRACTP
Library . . . . . : QSYS
Bound . . . . . : *YES
Change date/time . . . . . : 00/08/14 18:54:04
Teraspace storage enabled
modules . . . . . : *ALL
Storage model . . . . . : *SINGLVL

  Number  Symbol  Ref  Identifier  Type  Scope  Export  Key
00000014 Def      _Qln_DateISODescriptor Data  Strong
00000015 Def      _Qln_TimeISODescriptor Data  Strong
00000016 Def      _Qln_acpt_norm Proc  Strong
00000017 Def      _Qln_acpt_console Proc  Strong
00000018 Def      _Qln_acpt_session Proc  Strong
00000019 Def      _Qln_acpt_time Proc  Strong
:
0000015B Def      CEESECI Proc  Strong
0000015C Def      CEEDYWK Proc  Strong
0000015D Def      CEELUCT Proc  Strong
0000015E Def      CEEUTC Proc  Strong
0000015F Def      CEEGMT Proc  Strong
00000160 Def      CEEUTCO Proc  Strong
00000161 Def      Q LE leActivationInit Proc  Strong
00000162 Def      Q LE leActivationInitRouter Proc  Strong
00000163 Def      QleActBndPgm Proc  Strong
00000164 Def      QleGetExp Proc  Strong
00000165 Def      Q LE leCheck Proc  Strong

***** END OF BINDER INFORMATION LISTING **

```

Figure 23. CRTPGM Listing - Binder Information Listing

- L** *Module and Library:* This field identifies the library and name of the module object or service program that was processed.
- M** *Number:* A unique identifier assigned to each data item or ILE procedure in this program. This number is used for cross referencing.
- N** *Symbol:* This field identifies the symbol as an export or an import. If this field shows a value of Def then the symbol is an export. If this field shows a value of Ref then the symbol is an import.

- O** *Ref:* This field is blank if Symbol is Def or contains a symbol number if the value in the Symbol column is Ref. If the Symbol column is Ref, this field contains the unique number identifying the export (Def) that satisfies the import request.
- P** *Identifier:* This is the name of the symbol that is exported or imported.
- Q** *Type:* If the symbol name is an ILEprocedure, this field contains Proc. If the symbol name is a data item, this field contains Data.
- R** *Scope:* This field identifies the level at which the exported symbol name can be accessed.
- S** *Export:* This field identifies whether the data items be exported has a weak definition or a strong definition.
- T** *Key:* This field contains the length of weak exported items. The values shown in this field are in hexadecimal form.

The columns of the listing contain the following information:

Cross Reference Listing

The cross reference listing, provided only if *FULL is specified, is useful to the programmer who has a large binder listing and wants a handy index for it. The cross reference listing alphabetically lists all the unique identifiers in the binder listing with a corresponding list of all the definitions and resolved references of that identifier. Figure 24 on page 96 shows the layout of the Cross Reference Listing.

5722SS1 V5R4M0 060210 Page 15

Create Program CBLGUIDE/EXTLFL ISERIES1 06/02/15 13:14:03

Cross-Reference Listing

Identifier	Defs	-----Refs-----		Type	Library	Object
		Ref	Ref			
U _CEEDOD	0000014F			*SRVPGM	QSYS	QLEAWI
__CEEGSI	00000150			*SRVPGM	QSYS	QLEAWI
__CEEHDLR	0000013C			*SRVPGM	QSYS	QLEAWI
__CEEHDLU	0000013D			*SRVPGM	QSYS	QLEAWI
__CEERTX	00000135			*SRVPGM	QSYS	QLEAWI
__CEETSTA	0000014E			*SRVPGM	QSYS	QLEAWI
__CEEUTX	00000136			*SRVPGM	QSYS	QLEAWI
_C_session_cleanup	00000144			*SRVPGM	QSYS	QLEAWI
_C_session_open	00000145			*SRVPGM	QSYS	QLEAWI
_Qln_acos	000000D7			*SRVPGM	QSYS	QLNRMATH
_Qln_acpt_attribute	0000001D			*SRVPGM	QSYS	QLNRACPT
_Qln_acpt_console	00000017			*SRVPGM	QSYS	QLNRACPT
_Qln_acpt_da	0000002B			*SRVPGM	QSYS	QLNRACPT
_Qln_acpt_date	0000001A			*SRVPGM	QSYS	QLNRACPT
_Qln_acpt_date_yyyy	0000002C			*SRVPGM	QSYS	QLNRACPT
_Qln_acpt_day	0000001B			*SRVPGM	QSYS	QLNRACPT
_Qln_acpt_day_of_week	0000001C			*SRVPGM	QSYS	QLNRACPT
_Qln_acpt_day_yyyy	0000002D			*SRVPGM	QSYS	QLNRACPT
_Qln_acpt_io_feed	00000021			*SRVPGM	QSYS	QLNRACPT
_Qln_acpt_lda	0000001F			*SRVPGM	QSYS	QLNRACPT
_Qln_acpt_norm	00000016			*SRVPGM	QSYS	QLNRACPT
_Qln_acpt_open_feed	00000020			*SRVPGM	QSYS	QLNRACPT
_Qln_acpt_pip	0000001E			*SRVPGM	QSYS	QLNRACPT
_Qln_acpt_session	00000018			*SRVPGM	QSYS	QLNRACPT
_Qln_acpt_time	00000019			*SRVPGM	QSYS	QLNRACPT
:						
Q LE leBdyCh	00000131	00000011		*SRVPGM	QSYS	QLEAWI
Q LE leBdyEpilog	00000132	00000013		*SRVPGM	QSYS	QLEAWI
Q LE leCheck	00000165			*SRVPGM	QSYS	QLEAWI
Q LE leDefaultEh	0000012A	00000010		*SRVPGM	QSYS	QLEAWI
Q LE AG_prod_rc	00000129			*SRVPGM	QSYS	QLEAWI
Q LE AG_user_rc	00000128			*SRVPGM	QSYS	QLEAWI
Q LE HdTrRouterEh	00000138			*SRVPGM	QSYS	QLEAWI
Q LE RtxRouterCh	00000137			*SRVPGM	QSYS	QLEAWI
QleActBndPgm	00000163			*SRVPGM	QSYS	QLEAWI
QleGetExp	00000164			*SRVPGM	QSYS	QLEAWI
QlnDumpCobol	000000C0			*SRVPGM	QSYS	QLNRMAIN
QlnRtvCobolErrorHandler	000000C1			*SRVPGM	QSYS	QLNRMAIN
QlnSetCobolErrorHandler	000000C2			*SRVPGM	QSYS	QLNRMAIN

***** END OF CROSS - REFERENCE LISTING *****

Figure 24. CRTPGM Listing - Cross Reference Listing

The fields contain the following information:

- U** *Identifier:* The name of the export that was processed during symbol resolution.
- V** *Defs:* The unique identification number associated with each export.
- W** *Refs:* Lists the unique identification numbers of the imports (Ref) that were resolved to this import (Def).
- X** *Type:* Identifies whether the export can from a module object (*MODULE) or a service program (*SRVPGM).
- Y** *Library:* The name of the library in which the symbol described on this line has been defined.
- Z** *Object:* The name of the module object or service program in which the symbol described on this line has been defined.

Binding Statistics

The Binding Statistics section is produced only when the *FULL value is used on the DETAIL parameter. It shows how much system CPU time was used to bind specific parts of the program. These values may only have meaning to you when compared to similar output from other ILE programs or other times when a

particular program has been created. The value for the binding language compilation CPU time is always zero for an ILE program. Figure 25 shows the layout of the Binding Statistics.

```

5722SS1 V5R4M0 060210                                Create Program                                Page 22
                                                    CBLGUIDE/EXTLFL  ISERIES1 06/02/15 13:14:03
                                                    Binding Statistics
Symbol collection CPU time . . . . . : .001
Symbol resolution CPU time . . . . . : .000
Binding directory resolution CPU time . . . . . : .009
Binder language compilation CPU time . . . . . : .000
Listing creation CPU time . . . . . : .082
Program/service program creation CPU time . . . . . : .020
Total CPU time . . . . . : .322
Total elapsed time . . . . . : 1.145
          * * * * *  E N D  O F  B I N D I N G  S T A T I S T I C S  * * * * *
*CPC5D07 - Program EXTLFL created in library CBLGUIDE.
          * * * * *  E N D  O F  C R E A T E  P R O G R A M  L I S T I N G  * * * * *

```

Figure 25. CRTPGM Listing - Binding Statistics

Modifying a Module Object and Binding the Program Object Again

Once a program object is created, it may need to be changed to address problems or to meet changing requirements. A program object is built from module objects, so you may not need to change the entire program object. You can isolate just the module object that needs to be changed, change it, and then bind the program object again. How you change the module object depends on what needs to be changed.

You can change a module object in five ways:

- Change the ILE COBOL source program of the module object
- Change the optimization level of the module object
- Change the observability of the module object
- Change the amount of performance collection enablement
- Change the profiling data enablement.

Note: You need authority to the source code and the necessary commands to make any of these changes to the module object.

If you want to change the optimization level or observability of a module object, you may not be required to create it again. This often happens when you want to debug a program object or when you are ready to put a program object into production. Such changes can be performed more quickly and use less system resources than creating the module object again.

In these situations you may have many module objects to create at the same time. You can use the Work with Modules (WRKMOD) command to get a list of module objects selected by library, name, generic symbol, or *ALL. You can also limit the list to just module objects created by the ILE COBOL compiler.

Once you have made a change to a module object, you must use the CRTPGM command or UPDPGM command to bind the program object again.

Changing the ILE COBOL Source Program

When you need to make a change to the ILE COBOL source program, do the following:

1. Change the ILE COBOL source program where required using SEU. Refer to “Entering Source Statements Using the Source Entry Utility” on page 12 for further details on change the source code.

See Using the application development tools in the client product for information about getting started with the client tools.

2. Compile the ILE COBOL source program using the CRTCLMOD command to create a new module object(s). Refer to “Using the Create COBOL Module (CRTCLMOD) Command” on page 24 for a description of compiling the ILE COBOL source program.
3. Bind the module objects using the CRTPGM command or UPDPMG command to create a new program object. Refer to “Using the Create Program (CRTPGM) Command” on page 79 for information about creating a program object.

Changing the Optimization Levels

You can change the levels at which the generated code is optimized to run on the system. When the compiler optimizes the code, it looks for processing shortcuts that reduce the amount of system resources necessary to produce the same output. It then translates the shortcuts into machine code.

For example:

$$a = (x + y) + (x + y) + 10$$

In solving for a, the compiler recognizes the equivalence between the two expressions (x + y) and uses the already computed value to supply the value of the second expression.

Greater optimization increases the efficiency with which the program object runs on the system. However, with greater optimization, you will encounter increased compile time and also you may not be able to view variables that have been optimized. You can change the optimization level of a module object to display variables accurately as you debug a program object and then change the optimization level when the program object is ready for production.

ILE compilers support a range of optimization levels. There are currently four optimization levels, three of which are available to ILE COBOL users, these are:

***NONE or 10**

No additional optimization is performed on the generated code. This optimization level allows variables to be displayed and changed when the program object is being debugged. This value provides the lowest level of runtime performance.

***BASIC or 20**

Some optimization (only at the local block level) is performed on the generated code. When the program object is being debugged, variables can be displayed but not changed. This level of optimization slightly improves runtime performance.

***FULL or 30**

Full optimization (at the global level) is performed on the generated code. Variables cannot be changed but can be displayed while the program object is being debugged. However, the displayed value of a variable during debugging may not be its current value.

The effect of optimization on runtime performance varies depending on the type of application. For example, for an application that is compute intensive, optimization may improve runtime performance significantly whereas for an application that is I/O intensive, optimization may improve runtime performance only minimally.

To change the optimization level of a module object in a program object, use the Work with Modules (WRKMOD) command. Type WRKMOD on the command line and the Work with Modules display is shown. Select option 5 (Display) from the Work with Modules display to view the attribute values that need to be changed. The Display Module Information display is shown in Figure 26.

```

                                Display Module Information
                                Display 1 of 1
Module . . . . . : COPYPROC
Library . . . . . : TESTLIB
Detail . . . . . : *BASIC
Module attribute . . . . . : CBLLE
Module information:
Module creation date/time . . . . . : 98/08/25 12:57:17
Source file . . . . . : QCBLLSRC
Library . . . . . : TESTLIB
Source member . . . . . : COPYPROC
Source file change date/time . . . . . : 98/08/19 12:04:57
Owner . . . . . : TESTLIB
Coded character set identifier . . . . . : 37
Text description . . . . . : PG - COPY within PR
PROCESS Statement Example
Creation data . . . . . : *YES
Intermediate language data . . . . . : *NO
                                More...

Press Enter to continue.
F3=Exit  F12=Cancel

```

Figure 26. First screen of Display Module Information display

First, check that the *Creation data* value is *YES. This means that the module object can be translated again once the *Optimization level* value is changed. If the value is *NO, you must create the module object again and include the machine instruction template to change the optimization level.

Next, press the Roll Down key to see more information about the module object.

```

                                Display Module Information
                                Display 1 of 1
Module . . . . . : COPYPROC
Library . . . . . : TESTLIB
Detail . . . . . : *BASIC
Module attribute . . . . . : CBLLE
Sort sequence table . . . . . : *HEX
Language identifier . . . . . : *JOB RUN
Optimization level . . . . . : *NONE
Maximum optimization level . . . . . : *FULL
Debug data . . . . . : *YES
Compressed . . . . . : *NO
Program entry procedure name . . . . . : _Q1n_pep
Number of parameters . . . . . : 0
Module state . . . . . : *USER
Module domain . . . . . : *SYSTEM
Number of exported defined symbols . . . . . : 2
Number of imported (unresolved) symbols . . . . . : 14
                                More...

Press Enter to continue.
F3=Exit  F12=Cancel

```

Figure 27. Second screen of Display Module Information display

Check the *Optimization level* value. It may already be at the level you desire.

If the module has the machine instruction template and you want to change the optimization level, press F12 (Cancel). The Work with Modules display is shown. Select option 2 (Change) for the module object whose optimization level you want

to change. the CHGMOD command prompt is shown as in Figure 29. Type over the value specified for the *Optimize module* prompt.

Next, press the Roll Down key to see the final set of information about the module object.

```

                                Display Module Information
                                Display 1 of 7
Module . . . . . : COPYPROC
  Library . . . . . : TESTLIB
Detail . . . . . : *BASIC
Module attribute . . . . . : CBLLE
  Profiling data . . . . . : *NOCOL
  Enable performance collection . . . . . : *PEP

  Teraspace storage enabled . . . . . : *YES

Module compatibility:
  Module created on . . . . . : V4R4M0
  Module created for . . . . . : V4R4M0
  Earliest release module can be restored to . . . . . : V4R4M0
  Conversion required . . . . . : *NO

                                Bottom

Press Enter to continue.

F3=Exit  F12=Cancel

```

Figure 28. Third screen of Display Module Information display

The *Enable performance collection* prompt shows that the module has been created with performance measurement code for the entry into and exit from program entry point only. The module compatibility prompts show the release and version of the operating system that the module is compatible with.

```

                                Change Module (CHGMOD)
Type choices, press Enter.
Module . . . . . COPYPROC      Name, generic*, *ALL
  Library . . . . . TESTLIB_    Name, *USRLIBL, *LIBL
Optimize module . . . . . *NONE_ *SAME, *FULL, *BASIC...
Remove observable info . . . . . *DBGDTA *SAME, *NONE, *ALL...
      + for more values
Enable performance collection:
  Collection level . . . . . *PEP *SAME, *NONE, *PEP, *FULL...
  Procedures . . . . . *ALLPRC, *NONLEAF
Profiling data . . . . . *COL *SAME, *NOCOL, *COL
Force module recreation . . . . . *NO *NO, *YES
Text 'description' . . . . . 'PG - COPY within PROCESS Statement Example
      _____

                                Bottom

F3=Exit  F4=Prompt  F5=Refresh  F12=Cancel  F13=How to use this display
F24=More keys

```

Figure 29. Prompt of the CHGMOD Command

Changing the module object to a lower level of optimization allows you to display and possibly change the value of variables while debugging.

Repeat the process for any additional module objects whose optimization you may want to change. Whether you are changing one module object or several in the same program object, the program creation time is the same because all imports are resolved when the system encounters them.

When you are finished changing the optimization level for the module objects in a program object, create the program object again using the CRTPGM command, or update the existing program object with the new module objects using the UPDPM command.

Removing Module Observability

Module observability refers to three kinds of data that can be stored with a module object. This data allows the module object to be debugged, to be changed without being created again, or to be optimized with an intermediate language optimizer. Once a module object is created, you can only remove this data. Once the data is removed, you must create the module object again to replace it. The three types of data are:

Intermediate Language Data

Represented by the *ILDTA value. This data is necessary to allow a module object to be optimized with an intermediate language optimizer. At the current time, only ILE C compiler supports the creation of this type of data.

Create Data

Represented by the *CRTDTA value. This data is necessary to translate the code to machine instructions. The machine instruction (MI) template is included with the module object when the module object is created using the CRTCLMOD command. The MI template continues to be there until it is explicitly removed. The module object must have this data for you to change its optimization level.

Debug Data

Represented by the *DBGDTA value. This data is necessary to allow a module object to be debugged. Debug data is included with the module object when the module object is created using the CRTCLMOD command. The type and amount of debug data is determined by the DBGVIEW parameter.

Removing all observability reduces the module object to its minimum size (with compression). Once all observability is removed, you cannot change the module object in any way unless you create the module object again.

To remove a type of data from the module object, remove all types, or remove none, use the Work with Modules (WRKMOD) command. Type WRKMOD on the command line and the Work with Modules display is shown. Select option 5 (Display) to view the attribute values that need to be changed. The Display Module Information display is shown in Figure 26 on page 99.

First, check the value of the *Creation Data* parameter. If it is *YES, the Create Data exists and can be removed. If the value is *NO, there is no Create Data to remove. The module object cannot be translated again unless you create it again and include the machine instruction template.

Next, press the Roll Down key to see more information about the module object. Check the value of the *Debug data* parameter. If it is *YES, the Debug Data exists and the module object can be debugged. If it is *NO, the Debug Data does not exist and the module object cannot be debugged unless you create it again and include the Debug Data. Select option 2 (Change) for the module object whose observability you want to change. The CHGMOD command prompt is shown. Type over the value specified for the *Remove observable info* prompt.

You can ensure that the module object is created again using the *Force module recreation* parameter. When the optimization level is changed, the module object is always created again if the Create Data has not been removed. If you want the program object to be translated again removing the debug data and not changing the optimization level, you must change the *Force module recreation* parameter value to *YES.

Repeat the process for any additional module objects you may want to change. Whether you are changing one module object or several in the same program object, the program creation time is the same because all imports are resolved when the system encounters them.

When you are finished changing the optimization level for the module objects in a program object, create the program object again using the CRTPGM command, or update the existing program object with the new module objects using the UPDPM command.

Enabling Performance Collection

The following are the options that may be specified when performance measurements are invoked for a compilation unit.

Collection Levels

The collection levels are:

***PEP** Performance statistics are gathered on the entry and exit of the program entry procedure only. Choose this value when you want to gather overall performance information for an application. This support is equivalent to the support formally provided with the TPST tool. This is the default.

***ENTRYEXIT**

Performance statistics are gathered on the entry and exit of all the procedures of the program. This includes the program PEP routine.

This choice would be useful if you want to capture information on all routines. Use this option when you know that all the programs called by your application were compiled with either the *PEP, *ENTRYEXIT or *FULL option. Otherwise, if your application calls other programs that are not enabled for performance measurement, the performance tool will charge their use of resources against your application. This would make it difficult for you to determine where resources are actually being used.

***FULL** Performance statistics are gathered on the entry and exit of all procedures. Also statistics are gathered before and after each call to an external procedure.

Use this option when you think that your application will call other programs that were not compiled with either *PEP, *ENTRYEXIT or *FULL. This option allows the performance tools to distinguish between resources that are used by your application and those used by programs it calls (even if those programs are not enabled for performance measurement). This option is the most expensive but allows for selectively analyzing various programs in an application.

Procedures

The procedure level values are:

***ALLPRC**

The performance data is collected for all procedures.

***NONLEAF**

Performance data is collected for procedures that are non-leaf procedures and for the PEP.

Note: *NOLEAF has no effect on ILE COBOL programs.

Chapter 5. Creating a Service Program

A service program is a special kind of system object that provides a set of services to ILE program objects that are bound to it.

This chapter describes:

- How service programs can be used
- How to write binder language commands for a service program
- How to create a service program using the CRTSRVPGM command
- How to call and share data with a service program.

Use WebSphere Development Studio Client for i5/OS. This is the recommended
method and documentation about creating a service program appears in that
product's online help.

See Using the application development tools in the client product for information about getting started with the client tools.

Definition of a Service Program

A **service program** is a collection of runnable procedures and available data items that are used by other ILE program objects and service programs. Service programs are system objects of type *SRVPGM and have a name specified when the service program is created.

You use the Create Service Program (CRTSRVPGM) command to create a service program. A service program resembles a program object in that both consist of one or more module objects bound together to make a runnable object. However, a service program differs in that it has no PEP. Since it has no PEP, it cannot be called nor canceled. In place of a PEP, the service program can export procedures. Only the exported procedures from the service program can be called through a static procedure call made from outside of the service program. Exports of service programs are defined using the binder language.

Refer to the *ILE Concepts* book for further information on service programs.

Using Service Programs

Service programs are typically used for common routines that are frequently called within an application and across applications. For example, the ILE COBOL compiler uses service programs to provide runtime services such as math functions and input/output routines. Service programs enable reuse of source programs, simplify maintenance, and reduce storage requirements. In many respects, a service program is similar to a subroutine library or procedure library.

You can update a service program without having to re-create the other program objects or service programs that use the updated service program provided that the interface is unchanged or changed only in an upward compatible manner. You control whether the changes are compatible with the existing support provided by the service program. To make compatible changes to a service program, new procedure names or data names should be added to the end of the export list and the same signature as before must be retained.

Writing the Binder Language Commands for an ILE COBOL Service Program

The **binder language** allows you to define the list of procedure names and data items that can be exported from a service program. For a full description of the binder language and the binder language commands, refer to the *ILE Concepts* book.

A **signature** is generated from the names of procedures and data items and from the order in which they are specified in the binder language. A signature is a value that identifies the interface supported by the service program. You can also explicitly specify the signature with the SIGNATURE parameter in the binder language.

For service programs created from ILE COBOL source programs, the following language elements are module exports that can be included in the export list of the binder language:

- The name in the PROGRAM-ID paragraph in the outermost ILE COBOL program of a compilation unit.
- The ILE COBOL compiler generated name derived from the name in the PROGRAM-ID paragraph in the outermost ILE COBOL program of a compilation unit provided that program does not have the INITIAL attribute. The name is derived by adding the suffix `_reset` to the name in the PROGRAM-ID paragraph. This name needs to be included in the export list only if the ILE COBOL program in the service program needs to be canceled.

Using the Create Service Program (CRTSRVPGM) Command

You create a service program using the Create Service Program (CRTSRVPGM) command. Any ILE module object can be bound into a service program. The module objects must exist before you can create a service program with it. You can create module objects from ILE COBOL source programs using the CRTCBMOD command. Refer to "Using the Create COBOL Module (CRTCBMOD) Command" on page 24 for a description of how to create a module object using the CRTCBMOD command.

Table 7 lists the CRTSRVPGM parameters and their defaults. For a full description
of the CRTSRVPGM command and its parameters, refer to the *CL and APIs* section
of the *Programming* category in the **i5/OS Information Center** at this Web site
-[http://www.ibm.com/systems/i/infocenter/..](http://www.ibm.com/systems/i/infocenter/)

Table 7. Parameters for CRTSRVPGM Command and Their Default Values

Parameter Group	Parameter(Default Value)
Identification	SRVPGM(*CURLIB/ <i>service-program-name</i>) MODULE(*SRVPGM)
Program access	EXPORT(*SRCFILE) SRCFILE(*LIBL/QSRVSRC) SRCMBR(*SRVPGM)
Binding	BNDSRVPGM(*NONE) BNDDIR(*NONE)
Run time	ACTGRP(*CALLER)

Table 7. Parameters for CRTSRVPGM Command and Their Default Values (continued)

Parameter Group	Parameter(Default Value)
Miscellaneous	OPTION(*GEN *NODUPPROC *NODUPVAR *WARN *RSLVREF) DETAIL(*NONE) REPLACE(*YES) AUT(*LIBCRTAUT) ALWUPD(*YES) ALWRINZ(*NO) TEXT(*BLANK) ALWLIBUPD(*NO) USRPRF(*USER) TGTRLS(*CURRENT)

Example of Creating a Service Program

This example shows you how to use the binder language to create a service program to perform financial calculations.

Assume that the following ILE COBOL source programs comprise the module objects that make up the service program.

- RATE
Computes the interest rate, given a loan amount, term, and payment amount.
 - AMOUNT
Computes the loan amount, given an interest rate, term, and payment amount.
 - PAYMENT
Computes the payment amount, given an interest rate, term, and loan amount.
 - TERM
Computes the term of payment, given an interest rate, loan amount, and payment amount.
1. The binder language for the service program that makes the RATE, AMOUNT, PAYMENT, and TERM ILE COBOL programs available looks like the following:

```
FILE: MYLIB/QSRVSRC MEMBER: FINANCIAL
STRPGMEXP PGMLVL(*CURRENT)
EXPORT SYMBOL('TERM')
EXPORT SYMBOL('RATE')
EXPORT SYMBOL('AMOUNT')
EXPORT SYMBOL('PAYMENT')
ENDPGMEXP
```

You can use SEU to enter the binder language source statement. The syntax checker in SEU will prompt and validate the binder language input when you specify a source type of BND. To start an edit session to enter the binder language source, type:

```
STRSEU SRCFILE(MYLIB/QSRVSRC) SRCMBR(FINANCIAL)
TYPE(BND) OPTION(2)
```

and press Enter.

2. Compile the four ILE COBOL source programs into module objects using the CRTCLMOD command. Assume that the module objects also have the names RATE, AMOUNT, PAYMENT, and TERM.

To create the service program you can run the required binder statements with this command:

```
CRTSRVPGM SRVPGM(MYLIB/FINANCIAL)
           MODULE(MYLIB/TERM MYLIB/RATE MYLIB/AMOUNT MYLIB/PAYMENT)
           EXPORT(*SRCFILE)
           SRCFILE(MYLIB/QSRVSRC)
           SRCMBR(*SRVPGM)
```

Notes:

- a. Source file QSRVSRC in library MYLIB is the file that contains the binder language source.
- b. A binding directory is not required here because all module objects needed to create the service program have been specified with the MODULE parameter.

Further examples of using the binder language and creating service programs can be found in the *ILE Concepts* book.

Using the Retrieve Binder Source (RTVBNDSRC) Command as Input

The Retrieve Binder Source (RTVBNDSRC) command can be used to retrieve the exports from a module or a set of modules, and place them (along with the binder language statements needed for the exports) in a specified file member. After the binder language has been retrieved into a source file member, you can edit the binder language to make changes as needed. This file member can later be used as input to the SRCMBR parameter of the Create Service Program (CRTSRVPGM) command.

By default, the CRTSRVPGM command has a binder language file specified on the EXPORT and SRCFILE parameters to identify the exports from the service program. The RTVBNDSRC command can be useful in helping you automatically create this binder language.

For more information about the Retrieve Binder Source (RTVBNDSRC) command,
refer to the *CL and APIs* section of the *Programming* category in the **i5/OS**
Information Center at this Web site -<http://www.ibm.com/systems/i/infocenter/>.

Calling Exported ILE Procedures in Service Programs

Exported ILE procedures in service programs can only be called from an ILE program object or another service program using a static procedure call.

You can call an exported ILE procedure in a service program from an ILE COBOL program by using the CALL *literal* statement (where *literal* is the name of an ILE procedure in the service program). See “Performing Static Procedure Calls using CALL literal” on page 221 for detailed information on how to write the CALL statement in your ILE COBOL program to call an exported ILE procedure in a service program.

Sharing Data with Service Programs

External data can be shared among the module objects in a service program, across service programs, across program objects, and between service programs and program objects.

In the ILE COBOL program, the data items to be shared among different module objects must be described with the EXTERNAL clause in the Working Storage Section. See “Sharing EXTERNAL Data” on page 237 or refer the section on the

EXTERNAL clause in the *IBM Rational Development Studio for i: ILE COBOL Reference* for a further description of how external data is used in an ILE COBOL program.

Data and files declared as EXTERNAL in an ILE COBOL program in a service program cannot be in the export list on the binder language for the service program. Data and files declared as EXTERNAL in an ILE COBOL program that is outside of the service program can share this data with an ILE COBOL program that is inside the service program by the activation time resolution to the EXTERNAL data and EXTERNAL files. This same mechanism also allows you to share EXTERNAL data and EXTERNAL files between two completely separate program objects activated in the same activation group.

Canceling an ILE COBOL Program in a Service Program

For you to cancel an ILE COBOL program that is part of a service program from outside that service program, you must specify the CANCEL procedure name of the ILE COBOL program in the export list of the binder language.

Chapter 6. Running an ILE COBOL Program

This chapter provides the information you need to run your ILE COBOL program.

The most common ways to run an ILE COBOL program are:

- Using a Control Language (CL) CALL command
- Using a High Level Language CALL statement (for example, ILE COBOL's CALL statement)
- Using a menu-driven application program
- Issuing a user-created command.
- Selecting the Run menu action or Run toolbar icon on the WebSphere Development Studio Client for i5/OS workbench. Use WebSphere Development Studio ILE COBOL. This is the recommended method and documentation about running an ILE COBOL program and appears in that product's online help.

See Using the application development tools in the client product for information about getting started with the client tools.

Running a COBOL Program Using the CL CALL Command

You can use the CL CALL command to run an ILE COBOL program. You can use a CL CALL command interactively, as part of a batch job, or include it in a CL program. An example of a CL CALL command is as follows:

```
CALL program-name
```

The program object specified by program-name 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 library-name/program-name
```

```
#  
#  
#
```

For further information about using the CL CALL command, see the *CL and APIs* section of the *Programming* category in the **i5/OS Information Center** at this Web site -<http://www.ibm.com/systems/i/infocenter/>.

When you are running a batch job that calls an ILE COBOL program that uses the Format 1 ACCEPT statement, the input data is taken from the job stream. This data must be placed immediately following the CL CALL for the ILE COBOL program. You must ensure that your program requests (through multiple ACCEPT statements) the same amount of data as is available. See the "ACCEPT Statement" section of the *IBM Rational Development Studio for i: ILE COBOL Reference* for more information.

If more data is requested than is available, the CL command following the data is treated as input data. If more data is available than is requested, each extra line of data is treated as a CL command. In each instance, undesirable results can occur.

Passing Parameters to an ILE COBOL Program Through the CL CALL Command

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

```
CALL PGM(program-name) PARM(parameter-1 parameter-2 parameter-3)
```

Each of the parameter values can only be specified in only one of the following ways:

- a character string constant
- a numeric constant
- a logical constant
- a double-precision floating point constant
- a program variable.

Refer to the section on passing parameters between programs in the *CL Programming* book for a full description of how parameters are handled.

Running an ILE COBOL Program Using a HLL CALL Statement

You can run an ILE COBOL program by calling it from another HLL program.

You can use the ILE COBOL CALL statement in a ILE COBOL program to call another ILE COBOL program. If the ILE COBOL call is a dynamic program call, the program object can be library qualified by using the IN LIBRARY phrase. For example, to call program object PGMNAME in library LIBNAME, you would specify:

```
CALL "PGMNAME" IN LIBRARY "LIBNAME" USING variable1.
```

Without the IN LIBRARY phrase, a program object is found by searching the library list *LIBL. See the "CALL Statement" section of the *IBM Rational Development Studio for i: ILE COBOL Reference* for more information.

To run an ILE COBOL program from ILE C, use an ILE C function call. The name of the function corresponds to the name of the ILE COBOL program. By default, this function call is a static procedure call. To perform a dynamic program call, use the #pragma linkage (PGMNAME, OS) directive. PGMNAME represents the name of the ILE COBOL program that you want to run from the ILE C program. Once you have used the #pragma linkage (PGMNAME, OS) directive to tell the ILE C compiler that PGMNAME is an external program, you can run your ILE COBOL program through an ILE C function call. For more information, refer to the chapter on writing programs that call other programs in the *IBM Rational Development Studio for i: ILE C/C++ Programmer's Guide*.

To run an ILE COBOL program from an ILE RPG program, use the CALL operation code to make a dynamic program call or the CALLB operation code to make a static procedure call. You identify the program to be called by specifying its name as the Factor 2 entry. For more information, refer to the chapter on calling programs and procedures in the *IBM Rational Development Studio for i: ILE RPG Programmer's Guide*.

To run an ILE COBOL program from C++, use a C++ function call. The name of the function corresponds to the name of the ILE COBOL program. To prevent C++ from internally changing the name of the function, that is to prevent the VisualAge® C++ function name from mangling, you must prototype the function call using the extern keyword. To call an ILE COBOL procedure that returns nothing, and takes one 2 byte binary number, the C++ prototype would be:

```
extern "COBOL" void PGMNAME(short int);
```

To call the same COBOL program object, you would specify a linkage of "OS". The prototype becomes:

```
extern "OS" void PGMNAME(short int);
```

A linkage of "COBOL" on a C++ function call not only prevents function name mangling but causes any arguments passed to the ILE COBOL procedure to be passed BY REFERENCE. If the ILE COBOL procedure is expecting a BY VALUE parameter then a linkage of "C" should be specified.

Running an ILE COBOL Program From a Menu-Driven Application

Another way to run an ILE COBOL program is from a menu-driven application. The workstation user selects an option from a menu, calling the appropriate program. The following figure illustrates an example of an application menu.

PAYROLL DEPARTMENT MENU

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

Option:

Figure 30. Example of an Application Menu

The menu shown in this figure is normally displayed by a CL program in which each option calls a separate COBOL program.

The DDS for the display file of the above PAYROLL DEPARTMENT MENU looks like the following:

```

.....+.....1.....+.....2.....+.....3.....+.....4.....+.....5.....+.....6.....+.....7.....+.....8
A* MENU PAYROLLD PAYROLL DEPARTMENT MENU
A
A          R MENU          TEXT('PAYROLL DEPARTMENT MENU')
A          1 29'PAYROLL DEPARTMENT MENU'
A          5 4'1. Inquire into employee master'
A          6 4'2. Change employee master'
A          7 4'3. Add new employee'
A          8 4'4. Return'
A          12 2'Option:'
A          RESP          12 10VALUES(1 2 3 4)
A                          DSPATR(MDT)

```

Figure 31. Data Description Specification of an Application Menu

Figure 31 shows an example of the CL program for the application menu illustrated in Figure 30.

```

PGM /* PAYROLL Payroll Department Menu */
DCLF FILE (PAYROLLD)
START: SNDRCVF RCDfmt(MENU)
IF (&RESP=1); THEN(CALL CBLINQ)
/* Inquiry */
ELSE +
  IF (&RESP=2); THEN(CALL CBLCHG)
  /* Change */
  ELSE +
    IF (&RESP=3); THEN(CALL CBLADD)
    /* Add */
    ELSE +
      IF (&RESP=4); THEN(RETURN)
      /* Return */
GOTO START
ENDPGM

```

Figure 32. Example of a CL program which calls ILE COBOL Programs

If the user enters 1, 2, or 3 from the application menu, the CL program in Figure 32 calls the ILE COBOL programs CBLINQ, CBLCHG, or CBLADD respectively. If the user enters 4 from the application menu, the CL program returns to the program that called it.

Running an ILE COBOL Program Using a User Created Command

You can also create a command yourself to run an ILE COBOL program by using a command definition. A **command definition** is an object 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. The system-recognized object type is *CMD.

For example, you can create a command, PAY, that calls a program, PAYROLL. PAYROLL is the name of an ILE COBOL program that is called and run. You can enter the command interactively, or in a batch job. See the *CL Programming* book for further information about using the command definition.

Ending an ILE COBOL Program

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

```

# If an ILE COBOL program ends abnormally during run time, the escape message
# CEE9901
# Application error. message-id unmonitored by program-name
# at statement statement-number, instruction instuction-number.
#
# is issued to the caller of the run unit. A CL program can monitor for this exception
# by using the Monitor Message (MONMSG) command. For more information about
# control language commands, see the CL and APIs section of the Programming
# category in the i5/OS Information Center at this Web site -http://www.ibm.com/systems/i/infocenter/.
#

```

If a program ends for any reason other than by:

- Use of the STOP RUN statement
- Use of the GOBACK statement in the main program

- Use of the EXIT-PROGRAM AND CONTINUE RUN UNIT statement in the main program
 - Falling through to the end of the program,
- the RTNCDE job attribute is set to 2.

See the RTVJOBA and DSPJOB commands in the *CL Programming* book for more information about return codes.

Replying to Run Time Inquiry Messages

When you run an ILE COBOL program, run-time inquiry messages may be generated. The messages require a response before the program continues running.

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

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

- LNR7200
- LNR7201
- LNR7203
- LNR7204
- LNR7205
- LNR7206
- LNR7207
- LNR7208
- LNR7209
- LNR7210
- LNR7211
- LNR7212
- LNR7213
- LNR7214
- LNR7604.

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

The INQMSGRPY parameter occurs on the following CL commands:

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

You can select one of four reply modes by specifying one of the following values for the INQMSGRPY parameter:

SAME	No change is made in the way that replies are sent to inquiry messages
RQD	All inquiry messages require a reply by the receiver of the inquiry messages
DFT	A default reply is issued

SYSRPYL The system reply list is checked for a matching reply list entry. If a match occurs, the reply value in that entry is used. If no entry exists for that inquiry message, a reply is required.

You can use the Add Reply List Entry (ADDRPYLE) command to add entries to the system reply list, or the Work with Reply List Entry (WRKRPYLE) command to change or remove entries in the system reply list. You can also reply to run time inquiry messages with a user-defined error-handler.

For details of the ADDRPYLE and WRKRPYLE commands, and for more
information about error-handling APIs, refer to the *CL and APIs* section of the
Programming category in the **i5/OS Information Center** at this Web site
-<http://www.ibm.com/systems/i/infocenter/>.

Chapter 7. Debugging a Program

Debugging allows you to detect, diagnose, and eliminate errors in a program.

Use WebSphere Development Studio, integrated i5/OS debugger. This is the
recommended method and documentation about debugging ILE COBOL programs
and appears in that product's online help.

With the integrated i5/OS debugger you can debug your program running on the
i5/OS from a graphical user interface on your workstation. You can also set
breakpoints directly in your source before running the debugger. The integrated
i5/OS debugger client user interface also enables you to control program
execution. For example, you can run your program, set line, watch, and service
entry point breakpoints, step through program instructions, examine variables, and
examine the call stack. You can also debug multiple applications, even if they are
written in different languages, from a single debugger window. Each session you
debug is listed separately in the Debug view.

You can also debug your OPM and ILE COBOL programs using the ILE source debugger. This chapter describes how to use the ILE source debugger to:

- Prepare your ILE COBOL program for debugging
- Start a debug session
- Add and remove programs from a debug session
- View the program source from a debug session
- Set and remove conditional and unconditional breakpoints
- Set and remove watch conditions
- Step through a program
- Display the value of variables, records, group items, and arrays
- Change the value of variables
- Change the reference scope
- Equate a shorthand name to a variable, 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 CL commands:

- Use the Start Debug (STRDBG) command and specify the UPDPROD(*NO) 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. The syntax for preventing file modification would be:

```
SET UPDPROD NO
```

which can be abbreviated as

```
SET U N
```

See the chapter on debugging in the *ILE Concepts* book for more information on the ILE source debugger (including authority required to debug a program object or a service program and the affects 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:

- Debug any ILE COBOL or mixed ILE language application
- Monitor the flow of a program by using the debug commands while the program is running.
- View the program source or change the debug view
- Set and remove conditional and unconditional breakpoints
- Set and remove watch conditions
- Step through a specified number of statements
- Display or change the value of variables, records, group items, and arrays.

Note: The ILE COBOL COLLATING SEQUENCE is not supported by the ILE source debugger. If you use the ILE COBOL COLLATING SEQUENCE clause in your ILE COBOL program to specify your own collating sequence, this collating sequence will not be used by the ILE source debugger.

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 enter more debug commands.

Before you can use the source debugger, you must specify the DBGVIEW parameter with a value other than *NONE when you create a module object or program object using the CRTCBMOD or CRTBNDCBL command. After you have started the debugger, you can set breakpoints or other ILE source debugger options, and then run the program.

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 upper, lower or mixed case. Refer to *ILE Concepts* book for a further discussion of the debug commands.

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

Table 8 summarizes these debug commands. The online help for the ILE source debugger describes the debug commands and explains their allowed abbreviations.

Table 8. ILE Source Debugger Commands

Debug 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. Refer to Table 9 on page 120 for a list of attributes and their ILE COBOL equivalences. These attributes are not the same as the attributes defined by ILE COBOL.
BREAK	Permits you to enter either an unconditional or conditional job breakpoint at a position in the program being tested. Use <code>BREAK position WHEN expression</code> to enter a conditional job breakpoint.

Table 8. ILE Source Debugger Commands (continued)

Debug Command	Description
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, group items, or arrays.
QUAL	Allows you to define the scope of variables that appear in subsequent EVAL or WATCH commands.
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 program being debugged.
TBREAK	Permits you to enter either an unconditional or a conditional breakpoint in the current thread at a position in the program being tested.
THREAD	Allows you to display the Work with Debugged Threads display or change the current thread.
WATCH	Allows you to request a breakpoint when the contents of a specified storage location is changed from its current value.
FIND	Searches ahead 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 characters entered.
RIGHT	Moves the displayed window of source to the right by the number of characters entered.
TOP	Positions the view to show the first line.
BOTTOM	Positions the view to show the last line.
NEXT	Positions the view to the next breakpoint in the source currently displayed.
PREVIOUS	Positions the view to the previous breakpoint in the source currently displayed.
HELP	Shows the online help information for the available source debugger commands.

Attributes of Variables

The ILE source debugger does not describe the attributes of variables in the same manner as ILE COBOL. Table 9 on page 120 shows the equivalence between the attributes of variables as described by the ILE source debugger and ILE COBOL data categories.

Table 9. Equivalence Between ILE Source Debugger Variable Attributes and ILE COBOL Data Categories

ILE source debugger attributes of variables	ILE COBOL data categories
FIXED LENGTH STRING	Alphabetic Alphanumeric Alphanumeric-edited Numeric-edited External floating-point Date Time Timestamp
GRAPHIC	DBCS DBCS-edited
CHAR	Boolean
INTEGER	Binary
CARDINAL	Unsigned native binary (USAGE COMP-5)
ZONED(2,0)	Zoned Decimal
PACKED(2,0)	Packed Decimal Packed Date Packed Time
PTR	Pointer Procedure-pointer
REAL	Internal floating-point

Preparing a Program Object for a Debug Session

Before you can use the ILE source debugger, you must use either the CRTCBMOD or CRTBNDCBL command specifying the DBGVIEW parameter.

You can create one of three views for each ILE COBOL module object that you want to debug. They are:

- Listing view
- Source view
- Statement view.

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 122.

Using a Listing View

A listing view is similar to the source listing portion of the compile listing or spool file produced by the ILE COBOL compiler.

In order to debug an ILE COBOL module object using a listing view, use the *LIST or *ALL value on the DBGVIEW parameter for either the CRTCBMOD or CRTBNDCBL commands when you create the module object or program object.

One way to create a listing view, is as follows:

```
CRTCBMOD MODULE(MYLIB/xxxxxxx)
SRCFILE(MYLIB/QCBLLESRC) SRCMBR(xxxxxxxx)
TEXT('CBL Program') DBGVIEW(*LIST)
```

When you generate the listing view by specifying `DBGVIEW(*LIST)` on the `CRTCBLMOD` or `CRTBNDCBL` commands, the size of the created module object is increased because of the listing view. The listing view provides all expansions (for example, `COPY` and `REPLACE` statements) made by the ILE COBOL compiler when it creates the module object or program object. The listing view exist independent of the source member. The source member can be changed or deleted without affecting the listing view.

If the source member contains multiple compilation units, the listing view will contain the source listings of all of the compilation units, even if only one of them will be debugged. However, any debug commands issued from the Display Module Source display will be applied only to the compilation unit that can be debugged.

Using a Source View

A source view contains references to the source statements of the source member.

To use the source view with the ILE source debugger, the ILE COBOL compiler creates references to the source member while the module object (`*MODULE`) is being created.

Note: The module object is created using references to locations of the source statements in the root source member instead of copying the source statements into the view. Therefore, you should not modify, rename, or move root source members between the creation of the module and the debugging of the module created from these members.

In order to debug an ILE COBOL module object using a source view, use the `*SOURCE` or `*ALL` value on the `DBGVIEW` parameter for either the `CRTCBLMOD` or `CRTBNDCBL` commands.

One way to create a source view, is as follows:

```
CRTCBLMOD MODULE(MYLIB/xxxxxxx)
SRCFILE(MYLIB/QCBLLESRC) SRCMBR(xxxxxxx)
TEXT('CBL Program') DBGVIEW(*SOURCE)
```

When you generate the source view by specifying `DBGVIEW(*SOURCE)` on the `CRTCBLMOD` or `CRTBNDCBL` commands, the size of the created module object is increased because of the source view but the size is smaller than that generated with the listing view. The size of the generated module object will be the same as for the statement view. The source view does not provide any expansions made by the ILE COBOL compiler when it creates the module object or program object. The source view depends on the unchanged existence of the source member. Any changes made to the source member will affect the source view.

If the source member contains multiple compilation units, the source view will contain the source code of all of the compilation units, even if only one of them can be debugged. However, any debug commands issued from the Display Module Source display will be applied only to the compilation unit being debugged.

Using a Statement View

A statement view does not contain source statements. It contains line numbers and statement numbers. To debug an ILE COBOL module object using a statement view, you need a hard copy of the compiler listing.

Note: No source code is shown in the Display Module Source display when a statement view is used to debug an ILE COBOL module object.

To debug an ILE COBOL module object using a statement view, use the *STMT, *SOURCE, *LIST or *ALL value on the DBGVIEW parameter for either the CRTCBMOD or CRTBNDCBL commands when you create the module.

One way to create a statement view, is as follows:

```
CRTCBMOD MODULE(MYLIB/xxxxxxx)
SRCFILE(MYLIB/QCBLLESRC) SRCMBR(xxxxxxx)
TEXT('CBL Program') DBGVIEW(*STMT)
```

When you generate the statement view by specifying DBGVIEW(*STMT) on the CRTCBMOD or CRTBNDCBL commands, the size of the created module object is increased minimally because of the statement view. The size of the created module object is smaller than that generated with the listing view or the source view. The statement view minimizes the size of the created module object while still allowing some form of debugging. The statement view only provides the symbol table and a mapping between statement numbers and debug line numbers.

Starting the ILE Source Debugger

Once you have created a debug view, 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. You can change the attributes of the debug mode later in the job by using the Change Debug (CHGDBG) command.

#

Table 10 lists the parameters and their default values for the STRDBG command and the CHGDBG command. The ENDDBG command does not have any parameters associated with it. For a full description of the STRDBG, CHGDBG, and ENDDBG commands and their parameters, refer to the *CL and APIs* section of the *Programming* category in the **i5/OS Information Center** at this Web site -<http://www.ibm.com/systems/i/infocenter/>.

Table 10. Parameters for STRDBG and CHGDBG Commands and their Default Values

Parameter Group	STRDBG Command Parameter(Default Value)	CHGDBG Command Parameter(Default Value)
Identification	PGM(*NONE) DFTPGM(*PGM)	DFTPGM(*SAME)
Trace	MAXTRC(200) TRCFULL(*STOPTRC)	MAXTRC(*SAME) TRCFULL(*SAME)
Miscellaneous	UPDPROD(*NO) OPMSRC(*NO) SRVPGM(*NONE) CLASS(*NONE) DSPMODSRC(*PGMDEP) SRCDBGPGM(*SYSDFT) UNMONPGM(*NONE)	UPDPROD(*SAME) OPMSRC(*SAME)

Note: Trace applies only to OPM programs and is not applicable to ILE programs and service programs.

You can initially add as many as 20 program objects to a debug session by using the Program (PGM) parameter on the STRDBG command. (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.) They can be any combination of ILE or OPM programs.

Only program objects can be specified on the PGM parameter of the STRDBG command. Up to 20 service programs can initially be added to the debug session by using the Service Program (SRVPGM) parameter of the STRDBG command. Additional service programs can be added to the debug session after it has been started. 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.

The first program specified on the STRDBG command is shown if it has debug data, and, if OPM, the OPMSRC parameter is *YES. If ILE, the entry module is shown, if it has debug data; otherwise, the first module bound to the ILE program with debug data is shown.

To debug an OPM program using the ILE source debugger, the following conditions must be 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

For example, to start a debug session for the sample debug program MYPGM1 and a called OPM program MYPGM2, type:

```
STRDBG PGM(TESTLIB/MYPGM1 MYLIB/MYPGM2) OPMSRC(*YES)
```

Note: You must have *CHANGE authority to a program object to add it to a debug session.

After entering the STRDBG command, the Display Module Source display appears. When a mixture of ILE programs and ILE debugger-enabled OPM programs are specified on the STRDBG command, the first program with debug data is shown. If the first program is an ILE program, the first module object bound to the program object with debug data is shown as in Figure 33 on page 124.

```

Display Module Source
Program:  MYPGM1      Library:  TESTLIB      Module:  MYPGM1
  1      IDENTIFICATION DIVISION.
  2      PROGRAM-ID.  MYPGM1.
  3      *
  4      * This is the main program that controls
  5      * the external file processing.
  6      *
  7
  8      ENVIRONMENT DIVISION.
  9      INPUT-OUTPUT SECTION.
 10      FILE-CONTROL.
 11          SELECT EF1
 12              ASSIGN TO DISK-EFILE1
 13              FILE STATUS IS EFS1
 14              ORGANIZATION IS SEQUENTIAL.
 15
                                          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 33. Starting a Debug Session

Setting Debug Options

After you start a debug session, you can set or change the following debug options using the SET debug command on the debug command line:

- 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 cannot be changed by the CHGDBG command. OPMSRC can only be changed by the SET debug 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
2. Type:
SET
The Set Debug Options display appears.
3. On this display, type Y (Yes) in the *OPM source debug support* field, and press the Enter key 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 for that program.

Running a Program Object in a Debug Session

Once the debug session has been started, you can run a program object in the debug session by pressing:

- F3 (End Program),
- F12 (Resume), or
- F21 (Command Line)

from the Display Module Source display. Then, call the program object from the command line using the CALL CL command.

When an exception occurs in a program object during a debug session, the exception is handled by the error and exception handling routines specified for the program object. If the exception is not handled by any exception handler prior to the exception being turned into a function check, then the debugger is invoked and the Display Module Source display is shown. The module object within which the exception occurred is displayed at the statement which caused the exception. Refer to Chapter 16, “ILE COBOL Error and Exception Handling,” on page 369 for more information on error and exception handling.

You can stop program execution by setting breakpoints or by pressing F3 (End Program) from the Display Module Source display. Refer to “Setting and Removing Breakpoints” on page 129 for more information on setting breakpoints.

Adding Program Objects and Service Programs to a Debug Session

You can add more program objects and service programs to a debug session after starting the session.

To add **ILE program objects** and service programs to a debug session, use option 1 (Add program) and type the name of the program object on the first line of the Work with Module List display (see Figure 34 on page 126). The Work with Module List display can be accessed from the Display Module Source display by pressing F14 (Work with Module List). To add a service program, change the default program type from *PGM to *SRVPGM. There is no limit to the number of ILE program objects and service programs that can be included in a debug session at any given time.

To add **OPM program objects** to a debug session, 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 an OPM program using the Work with Module List 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 at one time when OPMSRC(*YES) is specified. If you specified OPMSRC(*NO), then you must use the Add Program (ADDPGM) command. Only 20 OPM programs can be in a debug session 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 an 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.

```

Work with Module List
System: ISERIES

Type options, press enter.
1=Add program 4=Remove program 5=Display module source
8=Work with module breakpoints

Program
Opt  Program/module  Library  Type
1    TEST            TESTLIB  *PGM
-    MYPGM1          TESTLIB  *PGM
-    MYPGM1          TESTLIB  *MODULE  Selected
-    USERDSP        DSPLIB   *SRVPGM
-    SAMPMDF        DSPLIB   *MODULE
-    GETUSER        DSPLIB   *MODULE

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

```

Figure 34. Adding an ILE Program Object to a Debug Session

When you have finished adding program objects or service programs to the debug session, press F3 (Exit) from the Work with Module List display to return to the Display Module Source display.

Note: You must have *CHANGE authority to a program to add it to a debug session. ILE service programs can be added to a debug session only by using option 1 on the Work with Module List display. ILE service programs cannot be specified on the STRDBG command.

Removing Program Objects or Service Programs from a Debug Session

You can remove program objects or service programs from a debug session after starting the session.

To remove ILE program objects and service programs from a debug session, use option 4 (Remove program), next to the program object or service program you want to remove, on the Work with Module List display (see Figure 35 on page 127). The Work with Module List display can be accessed from the Display Module Source display by pressing F14 (Work with Module List).

To remove **OPM program objects** from a debug session, 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 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 removed from a debug session at one time when OPMSRC(*YES) is specified. If you specified OPMSRC(*NO), then you must use the Remove Program (RMVPGM) command. Only ten OPM programs can be in a debug session when OPMSRC(*NO) is specified.

```

Work with Module List
System: ISERIES
Type options, press enter.
1=Add program 4=Remove program 5=Display module source
8=Work with module breakpoints
Program
Opt Program/module Library Type
- - - - -
4 TEST TESTLIB *PGM
- SAMPMDF *MODULE
- MYPGM1 TESTLIB *PGM
- MYPGM1 *MODULE Selected
- USERDSP DSPLIB *SRVPGM
- SAMPMDF *MODULE
- GETUSER *MODULE
Bottom
Command
====>
F3=Exit F4=Prompt F5=Refresh F9=Retrieve F12=Cancel

```

Figure 35. Removing an ILE Program Object from a Debug Session

When you have finished removing program objects or service programs from the debug session, press F3 (Exit) from the Work with Module List display to return to the Display Module Source display.

Note: You must have *CHANGE authority to a program to remove it to a debug session.

Viewing the Program Source

The Display Module Source display shows the source of an ILE program object or service program, one module object at a time. A module object's source can be shown if you created the module object with debug data, using one of the following debug view options:

- DBGVIEW(*STMT)
- DBGVIEW(*SOURCE)
- 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 the module object that is shown
- Change the view of the module object that is shown.

The ILE source debugger remembers the last position in which the module object is displayed and displays it in the same position when a module object is re-displayed. Lines 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 source line where the event occurred will be highlighted.

Changing the Module Object that is Shown

You can change the module object that is shown on the Display Module Source display by using option 5 (Display module source) on the Work with Module List display. The Work with Module List display can be accessed from the Display Module Source display by pressing F14 (Work with Module List). The Work with Module List display is shown in Figure 36.

To select a module object, type 5 (Display module source) next to the module object you want to show. If you use this option with an ILE program object, the module object containing the source 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).

```
Work with Module List                               System:  ISERIES
Type options, press enter.
1=Add program   4=Remove program   5=Display module source
8=Work with module breakpoints
Program
Opt  Program/module  Library  Type
-----
-      TEST              TESTLIB   *PGM
5      SAMPMDF           TESTLIB   *MODULE
-      MYPGM1            TESTLIB   *PGM
-      MYPGM1            TESTLIB   *MODULE   Selected
-      USERDSP          DSPLIB    *SRVPGM
-      SAMPMDF           DSPLIB    *MODULE
-      GETUSER           DSPLIB    *MODULE
                                             Bottom
Command
====>
F3=Exit  F4=Prompt  F5=Refresh  F9=Retrieve  F12=Cancel
```

Figure 36. Display a Module View

Once you have selected the module object that you want to view, press Enter and the selected view will be shown in the Display Module Source display.

An alternate method of changing the module object that is shown is to use the DISPLAY debug command. On the debug command line, type:

```
DISPLAY MODULE module-name
```

The module object *module-name* will now be shown. The module object must exist in a program object that has been added to the debug session.

Changing the View of the Module Object that is Shown

Several different views of an ILE COBOL module object are available depending on the values you specify when you create an ILE COBOL module object. These views are:

- ILE COBOL Listing view
- ILE COBOL Source 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 37 on page 129. The current view is listed at the top of the window, and the other views that are available are

- 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.
- Breakpoint functions are specified through debug commands.
These functions include:
 - Adding breakpoints
 - Removing breakpoints
 - Displaying breakpoint information
 - Resuming the running of a program object or service program after a breakpoint has been reached
- You can either have a job breakpoint or a thread breakpoint on a specified position at the same time, but not both.

Setting and Removing Unconditional Job Breakpoints

You can set or remove an unconditional job breakpoint by using:

- F6 (Add/Clear breakpoint) from the Display Module Source display
- F13 (Work with Module Breakpoints) from the Display Module Source display
- The BREAK debug command to set a job breakpoint
- The CLEAR debug command to remove a job breakpoint

The simplest way to set and remove an unconditional job breakpoint is to use F6 (Add/Clear breakpoint) from the Display Module Source display.

To set an unconditional job breakpoint using F6 (Add/Clear breakpoint), place your cursor on the line to which you want to add the breakpoint and press F6 (Add/Clear Breakpoint). An unconditional job breakpoint is set on the line.

To remove an unconditional job breakpoint using F6 (Add/Clear breakpoint), place your cursor on the line from which you want to remove the job breakpoint and press F6 (Add/Clear Breakpoint). The job breakpoint is removed from the line.

Repeat the previous steps for each unconditional job breakpoint you want to set.

If the line on which you want to set a job breakpoint has multiple statements, pressing F6 (Add/Clear Breakpoint) will set the job breakpoint at the first statement on the line.

Note: If the line on which you want to set a job breakpoint is not a runnable statement, the job breakpoint will be set at the next runnable statement.

To remove an unconditional 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.

After the breakpoints are set, press F3 (End Program) to leave the Display Module Source display. You can also use F21 (Command Line) from the Display Module Source display to call the program from a command line.

Call the program object. When a breakpoint is reached, the program object or service program stops and the Display Module Source display is shown again. At this point, you can evaluate variables, set more breakpoints, and run any of the debug commands.

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. *line-number* is the number in the currently displayed view of the module object on which you want to set a breakpoint.

If the line on which you want to set a breakpoint has multiple statements, the BREAK debug command will set the breakpoint at the first statement on the line.

To remove an unconditional job breakpoint using the CLEAR debug command, type:

```
CLEAR line-number
```

on the debug command line. *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 cleared for all threads.

Setting and Removing Unconditional Thread Breakpoints

You can set or remove an unconditional thread breakpoint by using:

- F13 (Work with Module Breakpoints) from the Display Module Source display
- The TBREAK debug command to set a thread breakpoint in the current thread
- The CLEAR debug command to remove a thread breakpoint.

Setting

Using the Work with Module Breakpoints Display: To set an unconditional thread breakpoint using the Work with Module Breakpoints display:

- Type 1 (Add) in the *Opt* field.
- In the *Thread* field, type the thread identifier.
- Fill in the remaining fields as if it were an unconditional job breakpoint.
- Press Enter.

Note: The *Thread* field is displayed when the DEBUG option on the SPAWN command is greater than or equal to one. For more information, see “Example of Using ILE COBOL in a Multithreaded Environment” on page 365.

Using the TBREAK Command: 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.

Removing

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 job breakpoint

Note: The relational operators supported for conditional breakpoints are <, >, =, =<, =>, and <>.

One way you can set or remove conditional job breakpoints is through the Work with Module Breakpoints display. The Work with Module Breakpoints display can be accessed from the Display Module Source display by pressing F13 (Work with module breakpoints). The Work with Module Breakpoints display is shown in Figure 38.

Setting

You can set conditional job breakpoints using the Work with Module Breakpoints display or using the BREAK debug command.

To set a conditional job breakpoint using the Work with Module Breakpoints display:

1. Type 1 (Add) in the *Opt* field.
2. Type the debugger line number, to which you want to set the breakpoint, in the *Line* field.
3. Type an conditional expression in the *Condition* field.
4. If a thread column is shown, before pressing Enter, type *JOB in the *Thread* field.
5. Press Enter.

```

                                Work with Module Breakpoints
                                System:  ISERIES
Program . . . : TEST              Library . . . : TESTLIB
Module . . . : SAMPMDF           Type . . . . : *PGM
Type options, press Enter.
1=Add  4=Clear
Opt    Line    Condition
1      35_____ I=21_____
-      _____

```

Figure 38. Setting a Conditional Breakpoint

If the line on which you want to set a breakpoint has multiple statements, the breakpoint is set at the first statement on the line.

Note: If the line on which you want to set a breakpoint is not a runnable statement, the breakpoint will be set at the next runnable statement.

Once you have finished specifying all of the breakpoints that you want to set or remove, press F3 (Exit) to return to the Display Module Source display.

Then press F3 (End Program) to leave the Display Module Source display. You can also use F21 (Command Line) from the Display Module Source display to call the program object from a command line.

Run the program object or service program. When a statement with a conditional job breakpoint is reached, the conditional expression associated with the breakpoint is evaluated before the statement is run. If the result is false, the program object continues to run. If the result is true, the program object stops, and the Display Module Source display is shown. At this point, you can evaluate variables, set more breakpoints, and run any of the debug commands.

To set a conditional job breakpoint using the BREAK debug command, type:

```
BREAK line-number WHEN expression
```

on the debug command line. *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 conditional expression can only be a simple expression. The term on the right hand side of the equation can only contain a single value. For example, I=21 is accepted but I=A+2 or I=3*2 are not accepted.

If the line on which you want to set a breakpoint has multiple statements, the BREAK debug command will set the breakpoint at the first statement on the line.

Example: For example, to set a conditional job breakpoint at debugger line 35:

1. Type 1 (Add) in the *Opt* field.
2. Type 35 in the *Line* field.
3. Type I=21 in the *Condition* field, and press Enter as shown in Figure 38 on page 132. (If a thread column is shown, before pressing Enter, type *JOB in the *Thread* field.)
4. Repeat the previous steps for each conditional job breakpoint you want to set.

Removing

You can remove conditional job breakpoints using the Work with Module Breakpoints display or using the CLEAR debug command.

To remove a conditional job breakpoint using the Work with Module Breakpoints Display, type 4 (Clear) in the *Opt* next to the breakpoint you want to remove, and press Enter. You can also remove unconditional breakpoints in this manner. Figure 38 on page 132 shows a typical display where 4 (Clear) could be entered in the *Opt* field.

Repeat the previous steps for each conditional job breakpoint you want to remove.

To remove a conditional job breakpoint using the CLEAR debug command, type:

```
CLEAR line-number
```

on the debug command line. *line-number* is line number in the currently displayed view of the module object from which you want to remove a job breakpoint.

Setting and Removing Conditional Thread Breakpoints

You can set or remove a conditional thread breakpoint by using:

- The Work with Module Breakpoints display
- The TBREAK debug command to set a conditional thread breakpoint in the current thread
- The CLEAR debug command to remove a conditional thread breakpoint.

Using the Work with Module Breakpoints Display

To set a conditional thread breakpoint using the Work with Module Breakpoints display:

1. Type 1 (Add) in the *Opt* field.
2. In the *Thread* field, type the thread identifier.
3. Fill in the remaining fields as if it were a conditional job breakpoint.
4. Press Enter.

To remove a conditional thread breakpoint using the Work with Module Breakpoints display:

1. Type 4 (Clear) in the *Opt* field next to the breakpoint you want to remove.
2. Press Enter.

Using the TBREAK or CLEAR Debug Commands

You use the same syntax for the TBREAK debug command as you would for the BREAK debug command. The difference between these commands is that the BREAK debug command sets a conditional job breakpoint at the same position in all threads, while the TBREAK debug command sets a conditional thread breakpoint in the current thread.

To remove a conditional thread breakpoint, use the CLEAR debug command. When a conditional thread breakpoint is removed, it is removed for the current thread only.

Removing All 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 request a job breakpoint when the contents of a specified variable (or an expression that relates to a substring or an array element) is changed from its current value. Setting watch conditions is similar to setting conditional job breakpoints, with these important differences:

- 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 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 will 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 on a system-wide basis 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 COBOL procedure, which can be re-used after the procedure ends.

A watch condition may be registered although the watched variable is no longer in scope. You must not assume that a variable is in scope just because a watch condition has been reported.

- Two watch locations in the same job must not overlap in any way. Two watch locations in different jobs must not start at the same storage address; otherwise, overlap is allowed. If these restrictions are violated, an error message is issued.

Note: Changes made to a watched storage location are ignored if they are made by a job other than the one that set the watch condition.

- After the command is successfully run, your application is stopped if a program in your session changes the contents of the watched storage location, and the Display Module Source display is shown.

If the program has debug data, and a source text view is available, it will be shown. The source line of the statement that was about to be run when the content change at the storage-location was detected is highlighted. A message indicates which watch condition was satisfied.

If the program cannot be debugged, the text area of the display will be blank.

- Eligible programs are automatically added to the debug session if they cause the watch-stop condition.

- When multiple watch conditions are hit on the same program statement, only the first one will be reported.
- You can set watch conditions also when you are using service jobs for debugging, that is, when you debug one job from another job.

Setting Watch Conditions

Before you can set a watch condition, your program must be stopped under debug, and the expression or variable you want to watch must be in scope:

- To watch a global variable, you must ensure that the COBOL program in which the variable is defined is active before setting the watch condition.
- To watch a local variable, you must step into the COBOL program 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 (a COBOL data item) 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    KOUNT              080090506F027004  4

                                Bottom

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

```

Figure 39. Example of a Work with Watch Display

The Work with Watch display shows all watches currently active in the debug session. You can remove or display watches from this display. When you select Option 5 (Display), the Display Watch window shown in Figure 40 on page 137 displays information about the currently active watch.

Displaying Active Watches

To display a system-wide list of active watches and show which job set them, type:
DSPDBGWCH

on a CL command line. This command displays the Display Debug Watches display shown below.

Display Debug Watches					System: DEBUGGER
-----Job-----			NUM	LENGTH	ADDRESS
MYJOBNAME1	MYUSERPRF1	123456	1	4	080090506F027004
JOB4567890	PRF4567890	222222	1	4	09849403845A2C32
JOB4567890	PRF4567890	222222	2	4	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 41. Example of a Display Debug Watch Display

Note: This display does not show watch conditions set by the system.

Removing Watch Conditions

Watches can be removed in the following ways:

- The CLEAR command used with the WATCH keyword selectively ends one or all watches. For example, to clear the watch identified by *watch-number*, type:

```
CLEAR WATCH watch-number
```

The watch number can be obtained from the Work With Watches display.

To clear all watches for your session, type:

```
CLEAR WATCH ALL
```

on a debug command line.

Note: While the CLEAR PGM command removes all breakpoints in the program that contains the module being displayed, it has no effect on watches. You must explicitly use the WATCH keyword with the CLEAR command to remove watch conditions.

- The CL End Debug (ENDDBG) command removes watches set in the local job or in a service job.

Note: ENDDBG will be called automatically in abnormal situations to ensure that all affected watches are removed.

- The initial program load (IPL) of your IBM i system removes all watch conditions system-wide.

Example of Setting a Watch Condition

In this example, you watch a variable *kount* in program MYLIB/PAYROLL. To set the watch condition, type:

```
WATCH kount
```

on a debug line, accepting the default value for the watch-length.

If the value of the variable *kount* changes subsequently, the application stops and the Display Module Source display is shown, as illustrated in Figure 42.

```

Display Module Source
Program:  PAYROLL      Library:  MYLIB      Module:  PAYROLL
 42      * THE FOLLOWING 3 PARAGRAPHS CREATE INTERNALLY THE *
 43      * RECORDS TO BE CONTAINED IN THE FILE, WRITES THEM *
 44      * ON THE DISK, AND DISPLAYS THEM                      *
 45      *****
 46      STEP-2.
 47          ADD 1 TO KOUNT, NUMBR.
 48          MOVE ALPHA (KOUNT) TO NAME-FIELD.
 49          MOVE DEPEND (KOUNT) TO NO-OF-DEPENDENTS.
 50          MOVE NUMBR          TO RECORD-NO.
 51      STEP-3.
 52          DISPLAY WORK-RECORD.
 53          WRITE RECORD-1 FROM WORK-RECORD.
 54      STEP-4.
 55          PERFORM STEP-2 THRU STEP-3 UNTIL KOUNT IS =
                                         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 55, variable: KOUNT
```

Figure 42. Example of Message Stating WATCH Was Successfully Set

- The line number of the statement where the change to the watch variable was detected is highlighted. This is typically the first executable line *following* the statement that changed the variable.
- A message indicates that the watch condition was satisfied.

Note: If a text view is not available, a blank Display Module Source display is shown, with the same message as above in the message area.

The following programs cannot be added to the ILE debug environment:

1. ILE programs without debug data
2. OPM programs with non-source debug data only
3. OPM programs without debug data

In the first two cases, the stopped statement number is passed. In the third case, the stopped MI instruction is passed. The information is displayed at the bottom of a blank Display Module Source display as shown below. Instead of the line number, the statement or the instruction number is given.

```

                                Display Module Source
Program:  PAYROLL          Library:  MYLIB          Module:  PAYROLL
        (Source not available.)

                                                                    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
Watch number 1 at instruction 18, variable: KOUNT

```

Figure 43. Example of a Display Module Source Display

Running a Program Object or ILE Procedure After a Breakpoint

After a breakpoint is encountered, you can resume running the program object or ILE procedure in two ways:

- resume running the program object or ILE procedure at the next statement after the breakpoint, and stop at the next breakpoint or when the program object ends.
- step through a specified number of statements after the breakpoint and stop the program object again.

Resuming a Program Object or ILE Procedure

After a breakpoint is encountered, you can resume running a program object or ILE procedure by pressing F12 (Resume) from the Display Module Source display. The program object or ILE procedure begins running on the next statement of the module object in which the program stopped. The program object or ILE procedure will stop again at the next breakpoint or when the program object ends.

Stepping Through the Program Object or ILE Procedure

After a breakpoint is encountered, you can run a specified number of statements of a program object or ILE procedure, then stop the program again and return to the Display Module Source display. The program object or ILE procedure begins running on the next statement of the module object in which the program stopped.

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 or ILE procedure 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 or ILE procedure 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), the next statement of the module object shown in the Display Module Source display is run, and the

program object or ILE procedure is stopped again. If multiple statements are contained in a line on which F10 (Step) or F22 (Step into) is pressed, all of the statements on that line are run and the program object or ILE procedure is stopped at the next statement on the next line.

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 or ILE procedure 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 or ILE procedure using the STEP debug command, type:

```
STEP number-of-statements
```

on the debug command line. *number-of-statements* is the number of statements that you want to run in the next step before the application is halted again. For example, if you type

```
STEP 5
```

on the debug command line, the next five statements of your program object or ILE procedure are run, then the program object or ILE procedure is stopped again and the Display Module Source display is shown.

When a CALL statement to another program object or ILE procedure is encountered in a debug session, you can:

- Step over the called program object or ILE procedure, or
- Step into the called program object or ILE procedure.

If you choose to **step over** the called program object or ILE procedure then the CALL statement and the called program object are run as a single step. The called program object or ILE procedure is run to completion before the calling program object or ILE procedure is stopped at the next step. Step over is the default step mode.

If you choose to **step into** the called program object or ILE procedure then each statement in the called program object or ILE procedure is run as a single step. If the next step at which the running program object or ILE procedure is to stop falls within the called program object or ILE procedure then the called program object or ILE procedure is halted at this point and the called program object or ILE procedure is shown in the Display Module Source display.

Stepping Over Program Objects or ILE Procedures

You can step over program objects or ILE procedures by using:

- F10 (Step) on the Display Module Source display
- The STEP OVER debug command

You can use F10 (Step) on the Display Module Source display to step over a called program object or ILE procedure in a debug session. If the next statement to be run is a CALL statement to another program object or ILE procedure, then pressing F10 (Step) will cause the called program object or ILE procedure to run to completion before the calling program object or ILE procedure is stopped again.

Alternately, you can use the STEP OVER debug command to step over a called program object or ILE procedure in a debug session. To use the STEP OVER debug command, type:

```
STEP number-of-statements OVER
```

on the debug command line. *number-of-statements* is the number of statements that you want to run in the next step before the application is halted again. If one of the statements that are run contains a CALL statement to another program object or ILE procedure, the ILE source debugger will step over the called program object or ILE procedure.

Stepping Into Program Objects or ILE Procedures

You can step into program objects or ILE procedure by using:

- F22 (Step into) on the Display Module Source display
- The STEP INTO debug command

You can use F22 (Step into) on the Display Module Source display to step into a called program object or ILE procedure in a debug session. If the next statement to be run is a CALL statement to another program object or ILE procedure then pressing F22 (Step into) will cause the first executable statement in the called program object or ILE procedure to be run. The called program object or ILE procedure will then be shown in the Display Module Source display.

Note: A called ILE program object or procedure must have debug data associated with it, in order for it to be shown in the Display Module Source display. A called OPM program object will be shown in the Display Module Source display if the ILE source debugger is set up to accept OPM programs, and the OPM program has debug data. (An OPM program has debug data if it was compiled with OPTION(*SRCDBG) or OPTION(*LSTDBG).)

Alternately, you can use the STEP INTO debug command to step into a called program object or ILE procedure in a debug session. To use the STEP INTO debug command, type:

```
STEP number-of-statements INTO
```

on the debug command line. *number-of-statements* is the number of statements of the program object or ILE procedure that you want to run in the next step before the program object or ILE procedure is halted again. If one of the statements that are run contains a CALL statement to another program object or ILE procedure, the debugger will step into the called program object or ILE procedure. Each statement in the called program object or ILE procedure will be counted in the step. If the step ends in the called program object or ILE procedure then the called program object or ILE procedure will then 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 or ILE procedure are run. If the third statement is a CALL statement to another program object or ILE procedure then two statements of the calling program object or ILE procedure are run and the first three statements of the called program object or ILE procedure are run.

Displaying Variables, Constant-names, Expressions, Records, Group Items, and Arrays

You can display the value of variables, constant-names, expressions, group items, records, and arrays by using:

- F11 (Display variable) on the Display Module Source display
- The EVAL debug command

The scope of the variables used in the EVAL command is defined by using the QUAL command.

Note: ILE COBOL special registers are not supported by the ILE source debugger. Thus, the values contained in the ILE COBOL special registers cannot be displayed in a debug session. The ILE source debugger cannot evaluate the result of a COBOL function identifier.

Displaying Variables and Expressions

The simplest way to display the value of a variable is to use F11 (Display variable) on the Display Module Source display. To display a variable using F11 (Display variable), place your cursor on the variable that you want to display and press F11 (Display variable). The current value of the variable is shown on the message line at the bottom of the Display Module Source display. For example, if you want to see the value of variable *COUNTER* on line 221 of the module object shown in Figure 44, place your cursor on top of the variable and press F11 (Display variable). The current value of the variable *COUNTER* is shown on the message line.

```

                                Display Module Source
Program:  TEST                Library:  TESTLIB        Module:  SAMPMPDF
213
214      PROCEDURE-SECTION SECTION.
215      FILL-TERMINAL-LIST.
216          READ TERMINAL-FILE RECORD INTO LIST-OF-TERMINALS(COUNTER)
217          AT END
218              SET END-OF-TERMINAL-LIST TO TRUE
219              SUBTRACT 1 FROM COUNTER
220              MOVE COUNTER TO NO-OF-TERMINALS.
221          ADD 1 TO COUNTER.
222
223      ACQUIRE-AND-INVITE-TERMINALS.
224          ACQUIRE LIST-OF-TERMINALS(COUNTER) FOR MULTIPLE FILE.
225          WRITE MULTIPLE-REC
226              FORMAT IS "SIGNON"
227              TERMINAL IS LIST-OF-TERMINALS(COUNTER).
                                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
COUNTER = 89.
```

Figure 44. Displaying a Variable using F11 (Display variable)

In cases where you are evaluating records, group items, 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.

You can also use the EVAL debug command to determine the value of a variable. First, you use the QUAL debug command to identify the line number of the variable that you want to show. Scoping rules for the variable are applied from this line.

Note: The default QUAL position is the current line.

To display the value of a variable using the EVAL debug command, type:

```
EVAL variable-name
```

on the debug command line. *variable-name* is the name of the variable that you want to display. The value of the variable 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, the value of the variable is shown on the Evaluate Expression display.

For example, to display the value of the variable *COUNTER* on line 221 of the module object shown in Figure 44 on page 143, type:

```
EVAL COUNTER
```

The message line of the Display Module Source display shows *COUNTER* = 89 as in Figure 45.

```

Display Module Source
Program:  TEST          Library:  TESTLIB      Module:  SAMPMDF
213
214      PROCEDURE-SECTION SECTION.
215      FILL-TERMINAL-LIST.
216      READ TERMINAL-FILE RECORD INTO LIST-OF-TERMINALS(COUNTER)
217      AT END
218          SET END-OF-TERMINAL-LIST TO TRUE
219          SUBTRACT 1 FROM COUNTER
220          MOVE COUNTER TO NO-OF-TERMINALS.
221      ADD 1 TO COUNTER.
222
223      ACQUIRE-AND-INVITE-TERMINALS.
224      ACQUIRE LIST-OF-TERMINALS(COUNTER) FOR MULTIPLE FILE.
225      WRITE MULTIPLE-REC
226          FORMAT IS "SIGNON"
227          TERMINAL IS LIST-OF-TERMINALS(COUNTER).
                                          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
COUNTER = 89.

```

Figure 45. Displaying a Variable using the EVAL debug command

Displaying Variables as Hexadecimal Values

You can use the EVAL debug command to display the value of variables in hexadecimal format. To display a variable in hexadecimal format, type:

```
EVAL variable-name: x 32
```

on the debug command line. *variable-name* is the name of the variable that you want to display in hexadecimal format. 'x' specifies that the variable is to be displayed in hexadecimal format and '32' indicates that a dump of 32 bytes after the start of the variable is to be displayed. The hexadecimal value of the variable is shown on the Evaluate Expression display as in Figure 46 on page 145. If no length is specified after the 'x', the size of the variable is used as the length. A minimum of 16 bytes is always displayed. If the length of the variable is less than 16 bytes, then the remaining space is filled with zeroes until the 16 byte boundary is

reached.

```

                                Evaluate Expression
Previous debug expressions
> BREAK 221
> EVAL COUNTER
COUNTER = 89.
> EVAL B : X 32
      00000      F8F90000 00000000 00000000 00000000 - 89.....
      00010      00000000 00000000 00000000 00000000 - .....

                                Bottom
Debug . . . _____
F3=Exit  F9=Retrieve  F12=Cancel  F19=Left  F20=Right  F21=Command entry
```

Figure 46. Displaying the Hexadecimal Value of a Variable using the EVAL debug command

Displaying a Substring of a Character String Variable

The ILE source debugger does not support the reference modification syntax of ILE COBOL. Instead, you can use the %SUBSTR operator with the EVAL command to display a substring of a character string variable. The %SUBSTR operator obtains the substring of a character string variable from a starting element position for some number of elements.

Note: The ILE source debugger does not support COBOL's reference modification syntax for handling substrings. You need to use the %SUBSTR operator of the ILE source debugger to handle substrings.

The syntax for the %SUBSTR operator is as follows:

```
%SUBSTR(identifier start-element number-of-elements)
```

where *identifier* must be a character string variable, and *start-element* and *number-of-elements* must be non-zero, positive integer literals. *identifier* can be a qualified, subscripted, or indexed variable. *start-element* + *number-of-elements* - 1 cannot be greater than the total number of elements in *identifier*.

For example, you can obtain the first 10 elements of a 20-element character string by using %SUBSTR(char20 1 10). You can obtain the last 5 elements of a 8-element character string by using %SUBSTR(char8 4 5). In the case of a DBCS or DBCS-edited item, element refers to a DBCS character (in other words, a two-byte character).

You can use the %SUBSTR operator to assign a substring of a character string variable to another variable or substring of a variable. Data is copied from the source variable to the target variable from left to right. When the source or target variables or both are substrings, then the operand is the substring portion of the character string variable, not the entire character string variable. When the source and target variable are of different sizes, then the following truncation and padding rules apply:

- If the length of the source variable is greater than the length of the target variable, the character string is truncated to the length of the target variable.

- If the length of the source variable is less than the length of the target variable, the character string is left justified in the target variable and the remaining positions are filled with blanks.
- If the length of the source variable is equal to the length of the target variable, the two variables will be exact copies of one another after the assignment.

Note: It is possible to use a substring of the same character string variable in both the source variable and the target variable; however, if any portion of the target string overlaps the source string, an error will result.

Figure 47 shows some example of how the %SUBSTR operator can be used.

```

                                Evaluate Expression
Previous Debug expressions
> EVAL CHAR10
  CHAR10 = '10CHARLONG'
> EVAL CHARA
  CHARA = 'A'
> EVAL CHARA = %SUBSTR(CHAR10 3 5)
  CHARA = 'C'
> EVAL %SUBSTR(CHAR10 1 2) = 'A'
  CHAR10 = 'A CHARLONG'
> EVAL %SUBSTR(CHAR10 1 2) = 'XYZ'
  CHAR10 = 'XYCHARLONG'
> EVAL %SUBSTR(CHAR10 7 4) = 'ABCD'
  CHAR10 = 'XYCHARABCD'
> EVAL %SUBSTR(CHAR10 1 2) = %SUBSTR(CHAR10 7 4)
  CHAR10 = 'ABCHARABCD'

                                Bottom
Debug . . . _____
F3=Exit  F9=Retrieve  F12=Cancel  F19=Left  F20=Right  F21=Command entry

```

Figure 47. Displaying a Substring of a Character String Variable using the %SUBSTR operator

Displaying the address of a level-01 or level-77 data item

You can use the ILE source debugger %ADDR operator with the EVAL command to display the address of a level-01 or level-77 data item. To display a level-01 or level-77 variable's address, type:

```
EVAL %ADDR(variable-name)
```

There is no dereference operator in ILE COBOL, but you still can display the area where the pointer data item points to as hexadecimal values or character values. To display the target area of a pointer data item as hexadecimal values, type:

```
EVAL pointer-name: x size
```

To display the target area of a pointer data item as character values, type:

```
EVAL pointer-name: c size
```

Displaying Records, Group Items, and Arrays

You can use the EVAL debug command to display structures, records, and arrays. Unless the record, group item, or array is on the current line, you must first qualify the record, group item, or array that you want to display by identifying its line number using the QUAL debug command. Refer to “Displaying Variables and Expressions” on page 143 for a description of how to use the QUAL debug command. To display a record, group item, or array, type:

```
EVAL data-name
```

on the debug command line. *data-name* is the name of the record, group item, or array that you want to display. The value of the record, group item, or array will be shown on the Evaluate Expression display.

The following example shows you how to display the contents of an ILE COBOL group item.

```
01 ACCOUNT.  
  02 NUMBER          PIC 9(5).  
  02 FULL-NAME.  
    03 LAST-NAME     PIC X(20).  
    03 FIRST-NAME    PIC X(10).
```

To display the contents of the group item *ACCOUNT*, type:

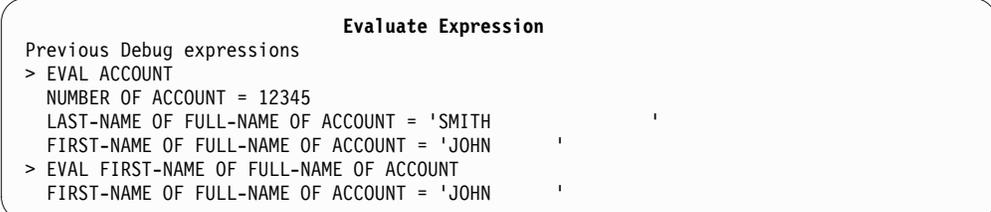
```
EVAL ACCOUNT
```

on the debug command line. The current contents of the group item *ACCOUNT* will be shown on the Evaluate Expression display as in Figure 48.

To display the contents of a single element of the group item *ACCOUNT*, such as element *FIRST-NAME OF FULL-NAME OF ACCOUNT*, type:

```
EVAL FIRST-NAME OF FULL-NAME OF ACCOUNT
```

on the debug command line. The current contents of the element *FIRST-NAME OF FULL-NAME OF ACCOUNT* will be shown on the Evaluate Expression display as in Figure 48. Press Enter to return to the Display Module Source display. You can also display elements using partially qualified names provided that the name is qualified sufficiently to resolve any name ambiguities.



The screenshot shows a window titled "Evaluate Expression". It contains the following text:

```
Previous Debug expressions  
> EVAL ACCOUNT  
  NUMBER OF ACCOUNT = 12345  
  LAST-NAME OF FULL-NAME OF ACCOUNT = 'SMITH'  
  FIRST-NAME OF FULL-NAME OF ACCOUNT = 'JOHN'  
> EVAL FIRST-NAME OF FULL-NAME OF ACCOUNT  
  FIRST-NAME OF FULL-NAME OF ACCOUNT = 'JOHN'
```

Figure 48. Displaying a Group Item using the EVAL Debug Command

The following example shows you how to display the contents of an ILE COBOL array.

```
05 A          PIC X(5) OCCURS 5 TIMES.
```

To display the contents of the array *A*, type:

```
EVAL A
```

on the debug command line. The current contents of the array *A* will be shown on the Evaluate Expression display as in Figure 49 on page 148.

To display the contents of a range of elements of the array *A*, type:

```
EVAL A(2..4)
```

on the debug command line. The current contents of elements *A(2)*, *A(3)*, and *A(4)* of the array *A* will be shown on the Evaluate Expression display as in Figure 49 on page 148.

To display the contents of a single element of the array *A*, such as element *A(4)*, type:

```
EVAL A(4)
```

on the debug command line. The current contents of the element *A(4)* will be shown on the Evaluate Expression display as in Figure 49. Press F3 (Exit) to return to the Display Module Source display.

Note: The subscript value specified on the EVAL debug command can only be a numeric value. For example, *A(4)* is accepted but *A(I+2)* or *A(2*3)* are not accepted.

```

                                     Evaluate Expression
Previous Debug expressions
> EVAL A
A(1) = 'ONE '
A(2) = 'TWO '
A(3) = 'THREE'
A(4) = 'FOUR '
A(5) = 'FIVE '
> EVAL A(2..4)
A(2) = 'TWO '
A(3) = 'THREE'
A(4) = 'FOUR '
> EVAL A(4)
A(4) = 'FOUR '
```

Figure 49. Displaying an Array using the EVAL Debug Command

Changing the Value of Variables

You can change the value of variables by using the EVAL command with an assignment operator. Unless the variable is on the current line, you must first qualify the variable that you want to change by identifying its line number using the QUAL debug command. Refer to “Displaying Variables and Expressions” on page 143 for a description of how to use the QUAL debug command. To change the value of the variable, type:

```
EVAL variable-name = value
```

on the debug command line. *variable-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 *variable-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.

You can use the EVAL debug command to assign numeric, alphabetic, alphanumeric, DBCS, boolean, floating-point, and date-time data to variables provided they match the definition of the variable.

Note: If the value that is assigned to the variable using the EVAL debug command does not match the definition of the variable, a warning message is issued and the value of the variable is not changed.

If the value that is assigned to a variable is a character string, the following rules apply:

- The length of the character string being assigned to the variable must be the same as the length of the variable.
- If the length of the character string being assigned to the variable is less than the length of the variable, then the character string is left justified in the variable and the remaining positions are filled with blanks.
- If the length of the character string being assigned to the variable is greater than the length of the variable, then the character string is truncated to the length of the variable.

The following are examples of how various type of data can be assigned to variables using the EVAL debug command.

```
EVAL COUNTER=3                (COUNTER is a numeric variable)
EVAL COUNTER=LIMIT            (LIMIT is another numeric variable)
EVAL END-OF-FILE='1'          (END-OF-FILE is a Boolean variable)
EVAL BOUNDARY=x'C9'           (BOUNDARY is an alphanumeric variable)
EVAL COMPUTER-NAME='ISERIES"   (COMPUTER-NAME is an alphanumeric variable)
EVAL INITIALS=%SUBSTR(NAME 17 3) (INITIALS and NAME are alphanumeric variables)
EVAL DBCS-NAME= G'0eK1K2K30r' (K1K2K3 are DBCS characters)

EVAL LONG-FLOAT(3) = -30.0E-3
    (LONG-FLOAT is an array of 3 double-precision floating-point
    data items - COMP-2)

EVAL SHORT-FLOAT = 10
    (SHORT-FLOAT is a single-precision floating-point data item -
    COMP-1)
```

Note: You cannot assign a figurative constant to a variable using the EVAL debug command. Figurative constants are not supported by the EVAL debug command. You may be able to change the value of a constant item in PROCEDURE DIVISION using the EVAL debug command. But the result is unpredictable.

Equating a Name with a Variable, Expression, or Command

You can use the EQUATE debug command to equate a name with a variable, 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 variable, 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 variable, expression, or debug command, and *definition* is the variable, 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 variable 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.

National Language Support for the ILE Source Debugger

When working with National Language Support for the ILE source debugger, the following conditions apply:

- When a view is displayed on the Display Module Source display, the ILE source debugger converts all data to the CCSID of the debug job.
- When assigning literals to variables, the ILE source debugger will not perform CCSID conversion on quoted literals (for example, 'abc'). Also, quoted literals are case sensitive.

When working with the source view, if the CCSID of the source file from which the source view is obtained is different from the CCSID of the module object, then the ILE source debugger may not recognize an ILE COBOL identifier containing invariant characters.

If either of the following conditions exist:

- The CCSID of the debug job is 290, 930, or 5026 (Japan Katakana)
- The code page of the device description used for debugging is 290, 930, or 5026 (Japan Katakana)

then debug commands, functions, and hexadecimal literals should be entered in uppercase. For example,

```
BREAK 16 WHEN var=X'A1B2'  
EVAL var:X
```

However, when debugging ILE COBOL, ILE RPG, or ILE CL module objects, identifier names in debug commands are converted to uppercase by the source debugger, and therefore may be displayed differently.

Changing and Displaying Locale-Based Variables

In ILE COBOL an item of class date-time, or a numeric-edited item, could be based in whole or in part on a locale. For example, a date item could be defined like:

```
01 group-item.  
   05 date1 FORMAT DATE SIZE 10 LOCALE is locale-french.
```

In this case the format of the date item and the CCSID of the characters that will form the contents of the date item will be based on the locale `locale-french`.

To create a locale, the locale must be described with a locale source member. Locale source is similar to COBOL source. It has a certain number of sections with predefined syntax and semantics, and just like COBOL source, must be compiled to form a locale object. To create a locale object, a CCSID must be specified, along

with the locale source member name, file, and library. For more information on creating locales, see “Creating Locales on the i5/OS” on page 197.

This means that the COBOL data item `date1` could have a CCSID different than the job CCSID. The ILE source debugger has no way to determine the CCSID of `date1`, so it converts the CCSID of the data item to the job CCSID. This may cause the contents of the data item to display incorrectly. To see the correct contents of these types of data items, you can display them in hexadecimal. For example, to see the contents of `date1` in hexadecimal, you would type:

```
EVAL date-1:x
```

Support for User-Defined Data Types

Defining a data item in the DATA DIVISION as a user-defined data type does not change how the data is interpreted by the debugger. Data items defined using the TYPE clause behave exactly as if they had been defined without using the TYPE clause.

Part 2. ILE COBOL Programming Considerations

Chapter 8. Working with Data Items

This chapter explains how to work with ILE COBOL numeric data, and how you can best represent numeric data and perform efficient arithmetic operations. Other topics include the use of intrinsic functions and working with items of class date-time. A list of topics are:

- “General ILE COBOL View of Numbers (PICTURE Clause)”
- “Computational Data Representation (USAGE Clause)” on page 156
- “Data Format Conversions” on page 165
- “Sign Representation and Processing” on page 166
- “Checking for Incompatible Data (Numeric Class Test)” on page 167
- “Performing Arithmetic” on page 168
- “Fixed-Point versus Floating-Point Arithmetic” on page 183
- “What is the Year 2000 Problem?” on page 186
- “Working with Date-Time Data Types” on page 189
- “Manipulating null-terminated strings” on page 208

General ILE COBOL View of Numbers (PICTURE Clause)

In general, you can view ILE COBOL numeric data in a way similar to character-string data—as a series of decimal digit positions. However, numeric items can have special properties, such as an arithmetic sign.

Defining Numeric Items

You define numeric items using the character “9” in the data description to represent the decimal digits of the number, instead of using an “X” as is done for alphanumeric items:

```
05 COUNT-X          PIC 9(4)  VALUE 25.  
05 CUSTOMER-NAME   PIC X(20) VALUE "Johnson".
```

You can code up to 18 digits in the PICTURE clause, as well as various other characters of special significance. The “S” in the following example makes the value signed.

```
05 PRICE           PIC S99V99.
```

The field can hold a positive or negative value. The “V” indicates the position of an implied decimal point. Neither “S” nor “V” are counted in the size of the item, nor do they require extra storage positions, unless the item is coded as USAGE DISPLAY with the SIGN IS SEPARATE clause. An exception is internal floating-point data (COMP-1 and COMP-2), for which there is no PICTURE clause. For example, an internal floating point data item is defined as follows:

```
05 GROMMET-SIZE-DEVIATION  USAGE COMP-1  VALUE 02.35E-5
```

For information on how you can control how the compiler handles floating-point data items, refer to the description of *FLOAT and *NOFLOAT under “CVTOPT Parameter” on page 35 and in “Using the PROCESS Statement to Specify Compiler Options” on page 51.

Separate Sign Position (For Portability)

If you plan to port your program or data to a different machine, you might want to code the sign as a separate digit in storage:

```
05 PRICE          PIC S99V9  SIGN IS LEADING, SEPARATE.
```

This ensures that the convention your machine uses for storing a non-separate sign will not cause strange results when you use a machine that uses a different convention.

Extra Positions for Displayable Symbols (Numeric Editing)

You can also define numeric items with certain editing symbols (such as decimal points, commas, and dollar signs) to make the data easier to read and understand when displayed or printed on reports. For example:

```
05 PRICE          PIC 9(5)V99.
05 EDITED-PRICE  PIC $$Z,ZZ9V99.
.
.
.
MOVE PRICE to EDITED-PRICE
DISPLAY EDITED-PRICE
```

If the contents of PRICE were 0150099 (representing the value 1,500.99), then \$ 1,500.99 would be displayed after the code is run.

How to Use Numeric-Edited Items as Numbers

Numeric-edited items are classified as alphanumeric data items, not as numbers. Therefore, they cannot be operands in arithmetic expressions or ADD, SUBTRACT, MULTIPLY, DIVIDE, and COMPUTE statements.

Numeric-edited items can be moved to numeric and numeric-edited items. In the following example, the numeric-edited item is *de-edited*, and its numeric value is moved to the numeric data item.

```
MOVE EDITED-PRICE to PRICE.
DISPLAY PRICE.
```

If these two statements were to immediately follow the statements shown in the previous example, then PRICE would be displayed as 0150099, representing the value 1,500.99.

Numeric-edited items can also be associated with a locale. When a MOVE is made to a numeric-edited item that is based on a locale, the result is edited according to that locale. The CCSID associated with a locale also affects the edited result, and when a program is run, the CCSIDs associated with the files, locales, and numeric-edited items used by a program are compared to see if conversion is necessary. For more information about how CCSIDs are treated at runtime, refer to "Runtime CCSID Considerations" on page 19.

For complete information on the data descriptions for numeric data, refer to *IBM Rational Development Studio for i: ILE COBOL Reference*.

Computational Data Representation (USAGE Clause)

You can control how the computer internally stores your numeric data items by coding the USAGE clause in your data description entries. The numeric data you use in your program will be one of the formats available with ILE COBOL:

- External decimal (USAGE DISPLAY)

- Internal decimal (USAGE PACKED-DECIMAL or COMP-3)
- Binary (USAGE BINARY or COMP-4)
- Native binary (USAGE COMP-5)
- External floating-point (USAGE DISPLAY)
- Internal floating-point (USAGE COMP-1, USAGE COMP-2)

COMP-4 is synonymous with BINARY, and COMP and COMP-3 are synonymous with PACKED-DECIMAL.

Regardless of which USAGE clause you use to control the computer's internal representation of the value, you use the same PICTURE clause conventions and decimal value in the VALUE clause, except for floating-point data.

External Decimal (USAGE DISPLAY) Items

When you code USAGE DISPLAY or omit the USAGE clause, each position (or byte) of storage contains one decimal digit. This corresponds to the format used for printing or displaying output, meaning that the items are stored in displayable form.

What USAGE DISPLAY Items Are For

External decimal items are primarily intended for receiving and sending numbers between your program and files, terminal, and printers. However, it is also acceptable to use external decimal items as operands and receivers in your program's arithmetic processing, and it is often convenient to program this way.

Should You Use Them for Arithmetic

If your program performs a lot of intensive arithmetic and efficiency is a high priority, you might want to use one of ILE COBOL's computational numeric data types for the data items used in the arithmetic.

The computer has to automatically convert displayable numbers to the *internal* representation of their numeric value before they can be used in arithmetic operations. Therefore, it is often more efficient to define your data items as computational items to begin with, rather than as DISPLAY items. For example:

```
05 COUNT-X          PIC S9V9(5)  USAGE COMP  VALUE 3.14159.
```

Internal Decimal (USAGE PACKED-DECIMAL or COMP-3)

Packed decimal format occupies 1 byte of storage for every two decimal places in the PICTURE description, except that the right-most byte contains only 1 digit and the sign. This format is most efficiently used when you code an odd number of digits in the PICTURE description, so that the left-most byte is fully used. Packed decimal format is handled as a fixed-point number for arithmetic purposes.

Why Use Packed Decimal

Packed decimal format:

- Requires less storage per digit than DISPLAY format requires.
- Is better suited for decimal alignment than binary format.
- Is converted to and from DISPLAY format more easily than binary format.
- Is well suited for containing arithmetic operands or results.

Binary (USAGE BINARY or COMP-4) Items

Binary format occupies 2, 4, or 8 bytes of storage, and is handled for arithmetic purposes as fixed-point number with the leftmost bit being the operational sign. For byte-reversed binary data, the sign bit is the leftmost bit of the rightmost byte.

How Much Storage BINARY Occupies

A PICTURE description with 4 or fewer decimal digits occupies 2 bytes; with 5 to 9 decimal digits, 4 bytes; with 10 to 18 decimal digits, 8 bytes.

Binary items are well suited for containing subscripts or reference modification start and length positions.

However, BINARY format is not as well suited for decimal alignment, so ILE COBOL converts BINARY numbers in arithmetic expressions to PACKED DECIMAL format. It is, therefore, preferable to use PACKED DECIMAL format for arithmetic expressions.

Using PACKED DECIMAL format over BINARY format is also preferable when converting numbers to display format. Converting a number from BINARY format to DISPLAY format is more difficult than converting a number from PACKED DECIMAL format to DISPLAY format.

Truncation of Binary Data (*STDTRUNC Compiler Option)

Use the *STDTRUNC and *NOSTDTRUNC compiler options (described in the "OPTION Parameter" on page "OPTION Parameter" on page 30). to indicate how BINARY and COMP-4 data is truncated.

Native Binary (USAGE COMP-5) Items

Native binary format is similar to binary format (USAGE BINARY or COMP-4) with the following differences:

- The *STDTRUNC and *NOSTDTRUNC compiler options do not apply to native binary items. The data items can contain values up to the capacity of the native binary representation (2, 4 or 8 bytes), rather than being limited to the value implied by the number of nines in the picture for the item (as is the case for USAGE BINARY data). When numeric data is moved or stored into a COMP-5 item, truncation occurs at the binary field size rather than at the COBOL picture size limit. When a COMP-5 item is referenced, the full binary field size is used in the operation.
- No bit is used for the operational sign when an item in COMP-5 native binary format is defined with no sign. Instead, all bits are used for numeric data, and the numeric value is never negative.

Internal Floating-Point (USAGE COMP-1 and COMP-2) Items

COMP-1 refers to short (single-precision) floating-point format, and COMP-2 refers to long (double-precision) floating-point format, which occupy 4 and 8 bytes of storage, respectively. The leftmost bit contains the sign; the next seven bits contain the exponent; the remaining 3 or 7 bytes contain the mantissa.

On IBM i, COMP-1 and COMP-2 data items are represented in IEEE format.

A PICTURE clause is not allowed in the data description of floating-point data items, but you can provide an initial value using a floating-point literal in the VALUE clause:

```
05 COMPUTE-RESULT      USAGE COMP-1  VALUE 06.23E-24.
```

The characteristics of conversions between floating-point format and other number formats are discussed in the section, "Data Format Conversions" on page 165.

Floating-point format is well-suited for containing arithmetic operands and results, and for maintaining the highest level of accuracy in arithmetic.

For complete information on the data descriptions for numeric data, see *IBM Rational Development Studio for i: ILE COBOL Reference*.

External Floating-Point (USAGE DISPLAY) Items

Displayable numbers coded in a floating-point format are called **external floating-point** items. Like external decimal items, you define external floating-point items explicitly with USAGE DISPLAY or implicitly by omitting the USAGE clause.

In the following example, COMPUTE-RESULT is implicitly defined as an external floating-point item. Each byte of storage contains one character (except for V).

```
05 COMPUTE-RESULT    PIC -9V(9)E-99.
```

The VALUE clause is not allowed in the data description for external floating-point items. Also, the minus signs (-) do not mean that the mantissa and exponent will always be negative numbers, but that when displayed the sign will appear as a blank for positive and a minus for negative. If a plus sign (+) were used, positive would be displayed as a plus sign and negative as a minus sign.

Just as with external decimal numbers, external floating-point numbers have to be converted (automatically by the compiler) to an internal representation of the numeric value before they can be operated on. External floating-point numbers are always converted to internal long floating-point format.

Creating User-Defined Data Types

In ILE COBOL, you can use the TYPEDEF clause to create **user-defined data types**. User-defined data types are not additions to the already available ILE COBOL data types, such as alphanumeric, numeric, boolean, and so on. User-defined data types (also known as **type definitions** or **type-names**) are actually entire elementary or group items that have been defined in the WORKING-STORAGE, LOCAL-STORAGE, LINKAGE or FILE section of a program, using the TYPEDEF clause. These type definitions act like templates that can then be used, using the TYPE clause, to define new data items. The new data item acquires all the characteristics of the user-defined data type. If the user-defined data type is a group item, then the new data item has subordinate elements of the same name, description, and hierarchy as those belonging to the user-defined data type.

User-defined data types can save you time and minimize source code because you don't have to redefine complex data structures that occur as part of the definition of two or more data items within your program. All you need to do is create one definition, and apply it to any subsequent definitions of the same type that you might need, by using the TYPE clause.

For example, imagine you are developing an inventory program for a small distributor, that distributes two types of items:

- Clothes. These come in many colors and sizes.
- Books. These come only with different titles.

Let's say the inventory program is going to count the amount on hand for each of the individual clothing items and books and store these in separate data items, and then also put the accumulated totals for the clothing and book inventories into separate data items.

Figure 50 on page 161 is an example of how you could use the `TYPDEF` and `TYPE` clauses to save time and minimize source code for the `WORKING-STORAGE` section of a program like this.

This example creates a user-defined data type for clothing and books. Then it creates separate data items for the three different clothing items and two different book items, based on the user-defined data types. This is much easier and more efficient than having to re-code the definitions for each of the inventory types. There's less chance of making a mistake, too.

```

          S o u r c e
STMT PL SEQNBR -A 1 B.+....2...+....3...+....4...+....5...+....6...+....7..IDENTFCN  S COPYNAME  CHG DATE
 1 000100 IDENTIFICATION DIVISION.
 2 000200 PROGRAM-ID.          SAMPTYPE.
   000300
   000400*****
   000500* The following program demonstrates some of the funcitons
   000600* available with the TYPE and TYPEDEF clauses.
   000700*****
   000800
 3 000900 ENVIRONMENT DIVISION.
 4 001000 CONFIGURATION SECTION.
 5 001100 SOURCE-COMPUTER.  IBM-ISERIES
 6 001200 OBJECT-COMPUTER.  IBM-ISERIES
 7 001300 INPUT-OUTPUT SECTION.
 8 001400 FILE-CONTROL.
 9 001500     SELECT DATA-IN
10 001600     ASSIGN TO Database-INVDATA
11 001700     ORGANIZATION IS INDEXED
12 001800     record key is inv-type
13 001900     with duplicates
14 002000     ACCESS MODE IS SEQUENTIAL.
15 002100
16 002200     SELECT PRINTER-FILE
17 002300     ASSIGN TO PRINTER-QPRINT
18 002400     ORGANIZATION IS SEQUENTIAL
19 002500     ACCESS MODE IS SEQUENTIAL.
20 002600
21 002700 DATA DIVISION.
22 002800 FILE SECTION.
23 002900 FD PRINTER-FILE.
24 003000 01 PRINTER-REC.
25 003100 05 PRINTER-RECORD    PIC X(132).
26 003200
27 003300*****
28 003400* define inventory type
29 003500*****
30 003600 01 INV-TYPE-T IS TYPEDEF PIC S9(3) VALUE 0.
31 003700 88 INV-TYPE-BOOK          VALUE 4, 5.
32 003800 88 INV-TYPE-BOOK-001      VALUE 4.
33 003900 88 INV-TYPE-BOOK-002      VALUE 5.
34 004000 88 INV-TYPE-CLOTHES     VALUE 1, 2, 3.
35 004100 88 INV-TYPE-CLOTHES-SWEATERS VALUE 1.
36 004200 88 INV-TYPE-CLOTHES-SOCKS VALUE 2.
37 004300 88 INV-TYPE-CLOTHES-PANTS VALUE 3.
38 004400
39 004500 FD DATA-IN.
40 004600 01 DATA-IN-REC.
41 004700 05 INV-TYPE          TYPE INV-TYPE-T.
42 004800 05 FILLER          PIC X(80).
43 004900
44 005000 WORKING-STORAGE SECTION.
45 005100*****
46 005200* Initialize END-OF-FILE flag to FALSE
47 005300*****

```

Figure 50. Example Showing How TYPEDEF and TYPE Clauses Can Be Used in a Program (Part 1 of 4)

```

STMT PL SEQNBR -A 1 B..+...2...+...3...+...4...+...5...+...6...+...7..IDENTFCN S COPYNAME CHG DATE
005400
36 005500 01 END-OF-FILE PIC 1 VALUE B"0".
37 005600 88 AT-END-OF-FILE VALUE B"1".
38 005700 01 ITEM-PRICE-T TYPEDEF PIC S9(4)V9(2) VALUE 0.
39 005800 01 ITEM-COLOR-T TYPEDEF PIC S9(2) VALUE 1.
40 005900 88 ITEM-COLOR-BLUE VALUE 1.
41 006000 88 ITEM-COLOR-RED VALUE 2.
42 006100 88 ITEM-COLOR-GREEN VALUE 3.
43 006200 01 ITEM-SIZE-T TYPEDEF PIC S9(2) VALUE 10.
44 006300 01 ITEM-COUNTER-T TYPEDEF PIC S9(6) VALUE 0.
006400
45 006500 01 ITEM-B-T TYPEDEF.
46 006600 05 ITEM-B-VALUE PIC S9(2).
47 006700 88 ITEM-B-BLUE VALUE 1.
48 006800 88 ITEM-B-RED VALUE 2.
49 006900 88 ITEM-B-GREEN VALUE 3.
50 007000 01 TEST-ITEM TYPE ITEM-B-T.
007100
51 007200 01 WORK-INV-TYPE TYPE INV-TYPE-T.
007300*****
007400* User-defined data type for items of clothing.
007500* Items of clothing are INVENTORY-TYPE 1 through 3.
007600*****
007700
52 007800 01 CLOTHING-ITEM IS TYPEDEF.
53 007900 05 CLOTHING-TYPE TYPE INV-TYPE-T.
54 008000 05 PRICE TYPE ITEM-PRICE-T.
55 008100 05 COLOR TYPE ITEM-COLOR-T.
56 008200 05 CLOTHING-SIZE TYPE ITEM-SIZE-T.
57 008300 05 FILLER PIC X(70).
008400
58 008500 01 SWEATERS TYPE CLOTHING-ITEM.
59 008600 01 SOCKS TYPE CLOTHING-ITEM.
60 008700 01 PANTS TYPE CLOTHING-ITEM.
008800
008900*****
009000* User-defined data type for books.
009100* Books are INVENTORY-TYPE 4 through 5.
009200*****
009300
61 009400 01 BOOK-ITEM IS TYPEDEF.
62 009500 05 BOOK-TYPE TYPE INV-TYPE-T.
63 009600 05 PRICE TYPE ITEM-PRICE-T.
64 009700 05 FILLER PIC X(20).
65 009800 05 BOOK-TITLE PIC X(40).
66 009900 05 FILLER PIC X(14).
010000
67 010100 01 BOOK-001 TYPE BOOK-ITEM.
68 010200 01 BOOK-002 TYPE BOOK-ITEM.
010300
010400*****
010500* Initialize all of the inventory counters.
010600*****
010700
69 010800 01 sweaters-count TYPE item-counter-t.

```

Figure 50. Example Showing How TYPEDEF and TYPE Clauses Can Be Used in a Program (Part 2 of 4)

```

STMT PL SEQNBR -A 1 B.+...2...+...3...+...4...+...5...+...6...+...7..IDENTFCN S COPYNAME CHG DATE
70 010900 01 socks-count TYPE item-counter-t.
71 011000 01 pants-count TYPE item-counter-t.
   011100
72 011200 01 book-001-count TYPE item-counter-t.
73 011300 01 book-002-count TYPE item-counter-t.
   011400
74 011500 01 clothes-count TYPE item-counter-t.
75 011600 01 book-count TYPE item-counter-t.
   011700
   011800*****
   011900* Declare report variables.
   012000*****
   012100
76 012200 01 header-line.
77 012300 05 FILLER pic x(40) value spaces.
78 012400 05 FILLER pic x(52) value "Detailed Inventory Report".
79 012500 05 FILLER pic x(40) value spaces.
   012600
80 012700 01 DETAIL-LINE.
81 012800 05 FILLER pic x(10) value spaces.
82 012900 05 ITEM-DESCRIPTION pic x(25) value spaces.
83 013000 05 ITEM-QUANTITY pic 9(6) blank when zero.
84 013100 05 FILLER pic x(92) value spaces.
   013200
   013300
85 013400 PROCEDURE DIVISION.
   013500 MAIN-PAR.
86 013600 OPEN INPUT DATA-IN
   013700 OUTPUT PRINTER-FILE.
   013800
   013900
   014000*****
   014100* Read the first record.
   014200*****
   014300
87 014400 READ DATA-IN
   014500 AT END
88 014600 SET AT-END-OF-FILE TO TRUE
   014700 NOT AT END
89 014800 MOVE INV-TYPE TO WORK-INV-TYPE
   014900 END-READ.
   015000
   015100*****
   015200* Tally each of the inventory types and move the amounts into
   015300* separate totals.
   015400*****
   015500
90 015600 PERFORM UNTIL AT-END-OF-FILE
91 015700 EVALUATE TRUE
   015800 WHEN INV-TYPE-CLOTHES-SWEATERS OF WORK-INV-TYPE
92 015900 ADD 1 TO sweaters-count
93 016000 ADD 1 TO clothES-count
   016100 WHEN INV-TYPE-CLOTHES-SOCKS OF WORK-INV-TYPE
94 016200 ADD 1 TO socks-count
95 016300 ADD 1 TO clothES-count

```

Figure 50. Example Showing How TYPEDEF and TYPE Clauses Can Be Used in a Program (Part 3 of 4)

```

5722WDS V5R4M0 060210 LN  IBM ILE COBOL          CBLGUIDE/SAMPTYPE      AISERIES 06/02/15 13:31:06      Page 5
STMT PL SEQNBR -A 1 B..+...2...+...3...+...4...+...5...+...6...+...7..IDENTFCN  S COPYNAME  CHG DATE
016400      WHEN INV-TYPE-CLOTHES-PANTS OF WORK-INV-TYPE
96 016500      ADD 1 TO pants-count
97 016600      ADD 1 TO clothES-COUNT
016700      WHEN INV-TYPE-BOOK-001 OF WORK-INV-TYPE
98 016800      ADD 1 TO book-001-count
99 016900      ADD 1 TO book-count
017000      WHEN INV-TYPE-BOOK-002 OF WORK-INV-TYPE
100 017100      ADD 1 TO book-002-count
101 017200      ADD 1 TO book-count
017300      END-EVALUATE
017400
102 017500      READ DATA-IN
017600      AT END
103 017700      SET AT-END-OF-FILE TO TRUE
017800      NOT AT END
104 017900      MOVE INV-TYPE TO WORK-INV-TYPE
018000      END-READ
018100      END-PERFORM.
018200
018300*****
018400* Write report.
018500*****
018600
105 018700      PERFORM REPORT-WRITE.
106 018800      CLOSE DATA-IN
018900      PRINTER-FILE.
107 019000      STOP RUN.
019100
019200*****
019300* Procedure to write report.
019400*****
019500
019600 REPORT-WRITE.
108 019700      WRITE PRINTER-REC FROM HEADER-LINE AFTER ADVANCING PAGE.
019800
109 019900      MOVE "BOOKS:"          TO ITEM-DESCRIPTION.
110 020000      MOVE ZEROS          TO ITEM-QUANTITY.
111 020100      WRITE PRINTER-REC FROM DETAIL-LINE AFTER ADVANCING 2 LINES.
020200
112 020300      MOVE "Best-seller Number 1:" TO ITEM-DESCRIPTION.
113 020400      MOVE BOOK-001-COUNT      TO ITEM-QUANTITY.
114 020500      WRITE PRINTER-REC FROM DETAIL-LINE AFTER ADVANCING 2 LINES.
020600
115 020700      MOVE "Best-seller Number 2:" TO ITEM-DESCRIPTION.
116 020800      MOVE BOOK-002-COUNT      TO ITEM-QUANTITY.
117 020900      WRITE PRINTER-REC FROM DETAIL-LINE AFTER ADVANCING 2 LINES.
021000
118 021100      MOVE "Total Books:"      TO ITEM-DESCRIPTION.
119 021200      MOVE BOOK-COUNT          TO ITEM-QUANTITY.
120 021300      WRITE PRINTER-REC FROM DETAIL-LINE AFTER ADVANCING 2 LINES.
021400
121 021500      WRITE PRINTER-REC FROM HEADER-LINE AFTER ADVANCING PAGE.
021600
122 021700      MOVE "CLOTHES:"        TO ITEM-DESCRIPTION.
123 021800      MOVE ZEROS          TO ITEM-QUANTITY.
124 021900      WRITE PRINTER-REC FROM DETAIL-LINE AFTER ADVANCING 2 LINES.
022000
125 022100      MOVE "Sweaters:"        TO ITEM-DESCRIPTION.
126 022200      MOVE SWEATERS-COUNT    TO ITEM-QUANTITY.
127 022300      WRITE PRINTER-REC FROM DETAIL-LINE AFTER ADVANCING 2 LINES.
022400
128 022500      MOVE "Socks:"           TO ITEM-DESCRIPTION.
129 022600      MOVE SOCKS-COUNT       TO ITEM-QUANTITY.
130 022700      WRITE PRINTER-REC FROM DETAIL-LINE AFTER ADVANCING 2 LINES.
022800
131 022900      MOVE "Pants:"           TO ITEM-DESCRIPTION.
132 023000      MOVE PANTS-COUNT       TO ITEM-QUANTITY.
133 023100      WRITE PRINTER-REC FROM DETAIL-LINE AFTER ADVANCING 2 LINES.
023200
134 023300      MOVE "Total Clothes:"    TO ITEM-DESCRIPTION.
135 023400      MOVE CLOTHES-COUNT     TO ITEM-QUANTITY.
136 023500      WRITE PRINTER-REC FROM DETAIL-LINE AFTER ADVANCING 2 LINES.
023600
          * * * * *   E N D   O F   S O U R C E   * * * * *

```

Figure 50. Example Showing How TYPEDEF and TYPE Clauses Can Be Used in a Program (Part 4 of 4)

Data Format Conversions

When the code in your program involves the interaction of items with different data formats, the compiler converts these items:

- Temporarily, for comparisons and arithmetic operations
- Permanently, for assignment to the receiver in a MOVE or COMPUTE statement.

What Conversion Means

A conversion is actually a move of a value from one data item to another. The compiler performs any conversions that are required during the execution of the arithmetic and comparison statements with the same rules that are used for MOVE and COMPUTE statements. The rules for moves are defined in *IBM Rational Development Studio for i: ILE COBOL Reference*. When possible, the compiler performs the move to preserve the numeric *value* as opposed to a direct digit-for-digit move. (For more information on truncation and predicting the loss of significant digits, refer to “Conversions and Precision.”)

Conversion Takes Time

Conversion generally requires additional storage and processing time because data is moved to an internal work area and converted before the operation is performed. The results might also have to be moved back into a work area and converted again.

Conversions and Precision

Conversion between fixed-point data formats (external decimal, packed decimal, and binary) are completed without loss of precision, as long as the target fields can contain all of the digits of the source operand.

Conversions Where Loss of Data is Possible

A loss of precision is possible in conversions between fixed-point data formats and floating-point data formats (short floating-point, long floating-point, and external floating-point). These conversions happen during arithmetic evaluations that have a mixture of both fixed-point and floating-point operands. (Because fixed-point and external floating-point items both have decimal characteristics, reference to fixed-point items in the following examples includes external floating-point items as well, unless stated otherwise.)

When converting from fixed-point to internal floating-point format, fixed-point numbers in base 10 are converted to the numbering system used internally, base 16.

Although the compiler converts short form to long form for comparisons, zeros are used for padding the short number.

When a USAGE COMP-1 data item is moved to a fixed-point data item with more than 6 digits, the fixed-point data item will receive only 6 significant digits, and the remaining digits will be zero.

Conversions that Preserve Precision: If a fixed-point data item with 6 or fewer digits is moved to a USAGE COMP-1 data item and then returned to the fixed-point data item, the original value is recovered.

If a USAGE COMP-1 data item is moved to a fixed-point data item of 6 or more digits and then returned to the USAGE COMP-1 data item, the original value is recovered.

If a fixed-point data item with 15 or fewer digits is moved to a USAGE COMP-2 data item and then returned to the fixed-point data item, the original value is recovered.

If a USAGE COMP-2 data item is moved to a fixed-point (not external floating-point) data item of 18 digits and then returned to the USAGE COMP-2 data item, the original value is recovered.

Conversions that Result In Rounding: If a USAGE COMP-1 data item, a USAGE COMP-2 data item, an external floating-point data item, or a floating-point literal is moved to a fixed-point data item, rounding occurs in the low-order position of the target data item.

If a USAGE COMP-2 data item is moved to a USAGE COMP-1 data item, rounding occurs in the low-order position of the target data item.

If a fixed-point data item is moved to an external floating-point data item where the PICTURE of the fixed-point data item contains more digit positions than the PICTURE of the external floating-point data item, rounding occurs in the low-order position of the target data item.

It is possible that when external floating-point data is DISPLAYed or ACCEPTed, or when an external floating-point literal is MOVEed to an external floating-point data item, the external floating-point data item displayed, accepted, or received can be an inaccurate value. This is because the floating-point data type is an approximation. When an external floating-point literal is accepted, displayed, or moved, it is first converted to a true floating-point value (IEEE), which can also affect its accuracy. For example, consider the following MOVE:

```
77 external-float-1 PIC +9(3).9(13)E+9(3).  
   MOVE +123455779012.34523E+297 to external-float-1.  
   DISPLAY "EXTERNAL-FLOAT-1=" external-float-1.
```

The displayed result of the MOVE is:

```
EXTERNAL-FLOAT-1=+123.4557790123452E+306
```

Sign Representation and Processing

Sign representation affects the processing and interaction of your numeric data.

Given X'sd', where s is the sign representation and d represents the digit, the valid sign representations for external decimal (USAGE DISPLAY) without the SIGN IS SEPARATE clause) are :

Positive: A, C, E, and F.

Negative: B and D.

Signs generated internally are F for positive and unsigned, and D for negative.

Given X'ds', where d represents the digit and s is the sign representation, the valid sign representations for internal decimal (USAGE PACKED-DECIMAL) ILE COBOL data are:

Positive: A, C, E, and F.

Negative: B and D.

Signs generated internally are F for positive and unsigned, and D for negative.

With the *CHGPOSSN Compiler Option

The ILE COBOL compiler option *CHGPOSSGN affects sign processing for external decimal and internal decimal data. *CHGPOSSGN has no effect on binary data or floating-point data. For more information, refer to the discussion of *CHGPOSSN on page “*NOCHGPOSSGN and *CHGPOSSGN” on page 34.

Positive signs generated by the compiler which are normally F become C on MOVE and arithmetic statements as well as the VALUE clause.

The *CHGPOSSGN compiler option is less effective than the default, *NOCHGPOSSGN, and should only be used when sharing data with MVS®, VM, or other systems with different preferred signs.

Checking for Incompatible Data (Numeric Class Test)

The compiler assumes that the values you supply for a data item are valid for the item’s PICTURE and USAGE clause, and assigns the value you supply without checking for validity. When an item is given a value that is incompatible with its data description, references to that item in the PROCEDURE DIVISION will be undefined, and your results will be unpredictable.

Frequently, values are passed into your program and are assigned to items that have incompatible data descriptions for those values. For example, non-numeric data might be moved or passed into a field in your program that is defined as an unsigned number. In either case, these fields contain invalid data. Ensure that the contents of a data item conforms to its PICTURE and USAGE clauses before using the data item in any further processing steps.

How to Do a Numeric Class Test

You can use the numeric class test to perform data validation. For example:

```
LINKAGE SECTION.  
01  COUNT-X      PIC 999.  
.  
.  
.  
PROCEDURE DIVISION USING COUNT-X.  
    IF COUNT-X IS NUMERIC THEN DISPLAY "DATA IS GOOD".  
.  
.  
.
```

The numeric class test checks the contents of a data item against a set of values that are valid for the particular PICTURE and USAGE of the data item. For example, a packed decimal item would be checked for hexadecimal values X'0' through X'9' in the digit positions, and for a valid sign value in the sign position (whether separate or non-separate).

Performing Arithmetic

If ILE COBOL together with ILE run-time provide various features to perform arithmetic:

- ADD, SUBTRACT, MULTIPLY, DIVIDE, and COMPUTE statements (discussed in “COMPUTE and Other Arithmetic Statements”).
- Arithmetic expressions (discussed in “Arithmetic Expressions” on page 169).
- Intrinsic functions (discussed in “Numeric Intrinsic Functions” on page 169).
- ILE callable services (APIs)

ILE provides several groups of bindable APIs which are available to every ILE compiler. The math APIs include CEE4SIFAC to compute factorials, and CEE5DCOS to compute cosine.

For more information about the bindable APIs that you can use, refer to the *CL*
and APIs section of the *Programming* category in the **i5/OS Information Center** at
this Web site -<http://www.ibm.com/systems/i/infocenter/>.

For the complete details of syntax and usage for ILE COBOL language constructs, refer to the *IBM Rational Development Studio for i: ILE COBOL Reference*.

COMPUTE and Other Arithmetic Statements

The general practice is to use the COMPUTE statement for most arithmetic evaluations rather than the ADD, SUBTRACT, MULTIPLY, and DIVIDE statements. This is because one COMPUTE statement can often be coded instead of several individual statements.

The COMPUTE statement assigns the result of an arithmetic expression to a data item:

```
COMPUTE Z = A + B / C ** D - E
```

or to many data items:

```
COMPUTE X Y Z = A + B / C ** D - E
```

When to Use Other Arithmetic Statements

Some arithmetic might be more intuitive using the other arithmetic statements. For example:

```
ADD 1 TO INCREMENT.
```

instead of:

```
COMPUTE INCREMENT = INCREMENT + 1.
```

Or,

```
SUBTRACT OVERDRAFT FROM BALANCE.
```

instead of:

```
COMPUTE BALANCE = BALANCE - OVERDRAFT.
```

Or,

```
ADD 1 TO INCREMENT-1, INCREMENT-2, INCREMENT-3.
```

instead of:

```
COMPUTE INCREMENT-1 = INCREMENT-1 + 1  
COMPUTE INCREMENT-2 = INCREMENT-2 + 1  
COMPUTE INCREMENT-3 = INCREMENT-3 + 1
```

You might also prefer to use the DIVIDE statement (with its REMAINDER phrase) for division in which you want to process a remainder. The REM Intrinsic Function also provides the ability to process a remainder.

Arithmetic Expressions

In the examples of COMPUTE shown above, everything to the right of the equal sign represents an arithmetic expression. Arithmetic expressions can consist of a single numeric literal, a single numeric data item, or a single Intrinsic Function reference. They can also consist of several of these items connected by arithmetic operators. These operators are evaluated in a hierarchic order.

Table 11. Operator Evaluation

Operator	Meaning	Order of Evaluation
Unary + or -	Algebraic sign	First
**	Exponentiation	Second
/ or *	Division or multiplication	Third
Binary + or -	Addition or subtraction	Last

Operators at the same level are evaluated from left to right; however, you can use parentheses with these operators to change the order in which they are evaluated. Expressions in parentheses are evaluated before any of the individual operators are evaluated. Parentheses, necessary or not, make your program easier to read.

In addition to using arithmetic expressions in COMPUTE statements, you can also use them in other places where numeric data items are allowed. For example, you can use arithmetic expressions as comparands in relation conditions:

```
IF (A + B) > (C - D + 5) THEN...
```

Numeric Intrinsic Functions

Intrinsic functions can return an alphanumeric, DBCS, numeric, boolean or date-time value.

Numeric intrinsic functions:

- Return a signed numeric value.
- Are considered to be temporary numeric data items.
- Can be used only in the places in the language syntax where expressions are allowed.
- Can save you time because you do not have to provide the arithmetic for the many common types of calculations that these functions cover. For more information on the practical application of intrinsic functions, refer to “Intrinsic Function Examples” on page 171.

Types of Numeric Functions

Numeric functions are classified into these categories:

Integer Those that return an integer.

Floating-Point Those that return a long floating-point value.

Argument Dependent

Return type depends on the arguments specified

The numeric functions available in ILE COBOL under these categories are listed in Table 12

Table 12. Types of Data that Numeric Functions Return

Integer	Floating-point	Argument Dependent
DATE-OF-INTEGERS	ACOS	MAX *
DATE-TO-YYYYMMDD	ANNUITY	MIN *
DAY-OF-INTEGERS	ASIN	RANGE
DAY-TO-YYYYDDD	ATAN	SUM
EXTRACT-DATE-TIME	COS	
FACTORIAL	LOG	
FIND-DURATION	LOG10	
INTEGER	MEAN	
INTEGER-OF-DATE	MEDIAN	
INTEGER-OF-DAY	MIDRANGE	
INTEGER-PART	NUMVAL	
LENGTH	NUMVAL-C	
MOD	PRESENT-VALUE	
ORD	RANDOM	
ORD-MAX	REM	
ORD-MIN	SIN	
YEAR-TO-YYYY	SQRT	
	STANDARD-DEVIATION	
	TAN	
	VARIANCE	

Note: * MAX and MIN can be alphanumeric.

Nesting Functions and Arithmetic Expressions

Numeric functions can be nested; you can reference one function as the argument of another. A nested function is evaluated independently of the outer function.

Because numeric functions and arithmetic expressions hold similar syntactic status, you can also nest an arithmetic expression as an argument to a numeric function:

```
COMPUTE X = FUNCTION MEAN (A, B, C / D).
```

In this example, there are only three function arguments: A, B and the arithmetic expression (C / D).

All Subscripting and Special Registers

Two other useful features of Intrinsic Functions are the ALL subscript and special registers.

You can reference all the elements of an array as function arguments by using the ALL subscript. This feature is used with tables.

The integer-type special registers are allowed as arguments wherever integer arguments are allowed.

Intrinsic Function Examples

You can use Intrinsic Functions to perform several different kinds of arithmetic as outlined in the table below:

Table 13. Types of Arithmetic that Numeric Functions Handle

Number Handling	Date/Time	Finance	Mathematics	Statistics
LENGTH	CURRENT-DATE	ANNUITY	ACOS	MEAN
MAX	DATE-OF-INTEGERS	PRESENT	ASIN	MEDIAN
MIN	DAY-TO-YYYYDDD	-VALUE	ATAN	MIDRANGE
NUMVAL	DATE-TO-YYYYMM		COS	RANDOM
NUMVAL-C	DD		FACTORIAL	RANGE
ORD-MAX	DAY-OF-INTEGERS		INTEGER	STANDARD
ORD-MIN	EXTRACT-DATE-TIME		INTEGER-PART	-DEVIATION
	FIND-DURATION		LOG	VARIANCE
	INTEGER-OF-DATE		LOG10	
	INTEGER-OF-DAY		MOD	
	WHEN-COMPILED		REM	
	YEAR-TO-YYYY		SIN	
			SQRT	
			SUM	
			TAN	

The following examples and accompanying explanations show intrinsic functions in each of the categories listed in the preceding table.

General Number-Handling: Suppose you want to find the mean value of three prices (represented as alphanumeric items with dollar signs), put this value into a numeric field in an output record, and determine the length of the output record. You could use NUMVAL-C (a function that returns the numeric value of an alphanumeric string) and the MEAN function to do this:

```
01 X                PIC 9(2).
01 PRICE1           PIC X(8)    VALUE "$8000".
01 PRICE2           PIC X(8)    VALUE "$4000".
01 PRICE3           PIC X(8)    VALUE "$6000".
01 OUTPUT-RECORD.
   05 PRODUCT-NAME  PIC X(20).
   05 PRODUCT-NUMBER PIC 9(9).
   05 PRODUCT-PRICE PIC 9(6).
.
.
.
PROCEDURE DIVISION.
  COMPUTE PRODUCT-PRICE =
    FUNCTION MEAN (FUNCTION NUMVAL-C(PRICE1)
                  FUNCTION NUMVAL-C(PRICE2)
                  FUNCTION NUMVAL-C(PRICE3)).
  COMPUTE X = FUNCTION LENGTH(OUTPUT-RECORD).
```

Additionally, to ensure that the contents in PRODUCT-NAME are in uppercase letters, you could use the following statement:

```
MOVE FUNCTION UPPER-CASE(PRODUCT-NAME) TO PRODUCT-NAME.
```

Date and Time: The following example shows how to calculate a due date that is 90 days from today. The first eight characters returned by the CURRENT-DATE function represent the date in a 4-digit year, 2-digit month, and 2-digit day format

(YYYYMMDD). In the example, this date is converted to its integer value. Then 90 is added to this value, and the integer is converted back to the YYYYMMDD format.

```
01 YYYYMMDD          PIC 9(8).
01 INTEGER-FORM      PIC S9(9).
.
.
.
MOVE FUNCTION CURRENT-DATE(1:8) TO YYYYMMDD.
COMPUTE INTEGER-FORM = FUNCTION INTEGER-OF-DATE(YYYYMMDD).
ADD 90 TO INTEGER-FORM.
COMPUTE YYYYMMDD = FUNCTION DATE-OF-INTEGER(INTEGER-FORM).
DISPLAY 'Due Date: ' YYYYMMDD.
```

You can also calculate a due date as a category date-time data item. For an example of this type of calculation, refer to “Example of Calculating a Due Date” on page 182.

Finance: Business investment decisions frequently require computing the present value of expected future cash inflows to evaluate the profitability of a planned investment. The present value of money is its value today. The present value of an amount that you expect to receive at a given time in the future is that amount which, if invested today at a given interest rate, would accumulate to that future amount.

For example, assume a proposed investment of \$1,000 produces a payment stream of \$100, \$200, and \$300 over the next three years, one payment per year respectively. The following ILE COBOL statements show how to calculate the present value of those cash inflows at a 10% interest rate.

```
01 SERIES-AMT1      PIC 9(9)V99    VALUE 100.
01 SERIES-AMT2      PIC 9(9)V99    VALUE 200.
01 SERIES-AMT3      PIC 9(9)V99    VALUE 300.
01 DISCOUNT-RATE  PIC S9(2)V9(6) VALUE .10.
01 TODAYS-VALUE     PIC 9(9)V99.
.
.
.
COMPUTE TODAYS-VALUE =
FUNCTION
PRESENT-VALUE(DISCOUNT-RATE SERIES-AMT1 SERIES-AMT2
SERIES-AMT3).
```

The ANNUITY function can be used in business problems that require you to determine the amount of an installment payment (annuity) necessary to repay the principal and interest of a loan. The series of payments is characterized by an equal amount each period, periods of equal length, and an equal interest rate each period. The following example shows how you could calculate the monthly payment required to repay a \$15,000 loan at 12% annual interest in three years (36 monthly payments, interest per month = .12/12):

```
01 LOAN             PIC 9(9)V99.
01 PAYMENT          PIC 9(9)V99.
01 INTEREST         PIC 9(9)V99.
01 NUMBER-PERIODS  PIC 99.
.
.
.
COMPUTE LOAN = 15000.
COMPUTE INTEREST = .12
COMPUTE NUMBER-PERIODS = 36.
COMPUTE PAYMENT =
LOAN * FUNCTION ANNUITY((INTEREST / 12) NUMBER-PERIODS).
```

Mathematics: The following ILE COBOL statement demonstrates how intrinsic functions can be nested, how arguments can be arithmetic expressions, and how previously complex mathematical calculations can be simply performed:

```
COMPUTE Z = FUNCTION LOG(FUNCTION SQRT (2 * X + 1))
+ FUNCTION REM(X 2)
```

Here, the remainder of dividing X by 2 is found with an intrinsic function, instead of using a DIVIDE statement with a REMAINDER clause.

Statistics: Intrinsic Functions also make calculating statistical information on data easier. Assume you are analyzing various city taxes and want to calculate the mean, median, and range (the difference between the maximum and minimum taxes):

```
01 TAX-S          PIC 99V999 VALUE .045.
01 TAX-T          PIC 99V999 VALUE .02.
01 TAX-W          PIC 99V999 VALUE .035.
01 TAX-B          PIC 99V999 VALUE .03.
01 AVE-TAX        PIC 99V999.
01 MEAN-TAX       PIC 99V999.
01 TAX-RANGE      PIC 99V999.
.
.
.
COMPUTE AVE-TAX = FUNCTION MEAN(TAX-S TAX-W TAX-B)
COMPUTE MEDIAN-TAX = FUNCTION MEDIAN(TAX-S TAX-W TAX-B)
COMPUTE TAX-RANGE = FUNCTION RANGE(TAX-S TAX-W TAX-B)
```

Converting Data Items (Intrinsic Functions)

Intrinsic Functions are available to convert character-string data items to the following:

- Uppercase or lower case
- Reverse order
- Numbers
- Date-time data items
- UTF-8

You can use the NATIONAL-OF and DISPLAY-OF intrinsic functions to convert to and from national (Unicode) strings.

Use TRIM, TRIML TRIMR intrinsic functions to remove leading and/or trailing characters from a string.

Besides using Intrinsic Functions to convert characters, you can also use the INSPECT statement.

Converting to Uppercase or Lowercase (UPPER-CASE, LOWER-CASE)

This code:

```
01 ITEM-1          PIC X(30)    VALUE "Hello World!".
01 ITEM-2          PIC X(30).
.
.
.
DISPLAY ITEM-1.
DISPLAY FUNCTION UPPER-CASE(ITEM-1).
DISPLAY FUNCTION LOWER-CASE(ITEM-1).
MOVE FUNCTION UPPER-CASE(ITEM-1) TO ITEM-2.
DISPLAY ITEM-2.
```

would display the following messages on the terminal:

```
Hello World!  
HELLO WORLD!  
hello world!  
HELLO WORLD!
```

The DISPLAY statements do not change the actual contents of ITEM-1 and only affect how the letters are displayed. However, the MOVE statement causes uppercase letters to be moved to the actual contents of ITEM-2.

Converting to Reverse Order (REVERSE)

The following code:

```
MOVE FUNCTION REVERSE(ORIG-CUST-NAME) TO ORIG-CUST-NAME.
```

would reverse the order of the characters in ORIG-CUST-NAME. For example, if the starting value were JOHNSON, the value after the statement is performed would be NOSNHOJ.

Converting to Numbers (NUMVAL, NUMVAL-C)

The NUMVAL and NUMVAL-C functions convert character strings to numbers. Use these functions to convert alphanumeric data items that contain free format character representation numbers to numeric form, and process them numerically. For example:

```
01 R          PIC X(20)  VALUE "- 1234.5678".  
01 S          PIC X(20)  VALUE "-$12,345.67CR".  
01 TOTAL      USAGE IS COMP-2.  
.  
.  
.  
COMPUTE TOTAL = FUNCTION NUMVAL(R) + FUNCTION NUMVAL-C(S).
```

The difference between NUMVAL and NUMVAL-C is that NUMVAL-C is used when the argument includes a currency symbol or comma, as shown in the example. You can also place an algebraic sign in front or in the rear, and it will be processed. The arguments must not exceed 18 digits (not including the editing symbols). For exact syntax rules, see the *IBM Rational Development Studio for i: ILE COBOL Reference* manual.

Note: Both NUMVAL and NUMVAL-C return a long (double-precision) floating-point value. A reference to either of these functions, therefore, represents a reference to a numeric data item.

Why Use NUMVAL and NUMVAL-C?: When you use NUMVAL or NUMVAL-C, you do not need to statically declare numeric data in a fixed format and input data in a precise manner. For example, for this code:

```
01 X          PIC S999V99  LEADING SIGN IS SEPARATE.  
.  
.  
.  
ACCEPT X FROM CONSOLE.
```

the user of the application must enter the numbers exactly as defined by the PICTURE clause. For example:

```
+001.23  
-300.00
```

However, using the NUMVAL function, you could code:

```

01 A          PIC X(10).
01 B          PIC S999V99.
.
.
ACCEPT A FROM CONSOLE.
COMPUTE B = FUNCTION NUMVAL(A).

```

and the input could be:

```

1.23
-300

```

Converting to Date-Time Data Items (CONVERT-DATE-TIME)

The CONVERT-DATE-TIME function takes an alphanumeric, numeric, or date-time item, and converts it to a date-time data item. The intrinsic functions can be used to:

- Convert dates, times, or timestamps from an alphanumeric (string) item to a date-time item
- Convert an item of category date in one format to another category date item, whose format is based on a locale.

For example, the following statement converts a non-numeric literal (an alphanumeric constant) to a category date data item:

```

MOVE FUNCTION CONVERT-DATE-TIME ('98/08/09' DATE '%y/%m/%d')
  TO DATE-1.

```

Conversion also occurs when comparing or moving numeric data items containing dates to date-time data items. For more information about the considerations for these types of moves, refer to “MOVE Considerations for Date-Time Data Items” on page 192.

When moving alphanumeric data items containing dates to date-time data items, no conversion is done: whatever characters are contained in the alphanumeric data item are moved to the date-time data item. It is the responsibility of the programmer to ensure that dates contained in alphanumeric data items are in the correct format before they are moved to date-time data items.

Converting to UTF-8 (UTF8STRING)

The UTF8STRING function converts character strings to UTF-8 (UCS Transformation Format 8). The UTF-8 coded form is represented by CCSID 1208. For example:

```

01 STR1 PIC X(3) VALUE "ABC".
01 VRR-X3 PIC X(3).
.
.
MOVE FUNCTION UTF8STRING(STR1) TO VRR-X3.

```

The contents of VRR-X3 would become X"414243".

Converting alphanumeric or DBCS to national data (NATIONAL-OF)

Use the NATIONAL-OF intrinsic function to convert an alphabetic, alphanumeric, or DBCS item to a character string represented in Unicode (UCS-2). Specify the source code page as an argument if the source is encoded in a different code page than is in effect with the CCSID compiler option.

Converting national to alphanumeric or DBCS data (DISPLAY-OF)

Use the DISPLAY-OF intrinsic function to convert a national item to a character string represented in the code page that you specify as an argument or with the CCSID compiler option. If you specify an EBCDIC code page that combines SBCS and DBCS characters, the returned string might contain a mixture of SBCS and DBCS characters, with DBCS substrings delimited by shift-in and shift-out characters.

Overriding the default code page

In some cases, you might need to convert data to or from a CCSID that differs from the CCSID specified as the CCSID option value. To do this, use a conversion function in which you specify the code page for the item explicitly.

If you specify a code page as an argument to DISPLAY-OF and it differs from the code page that you specify with the CCSID compiler option, do not use the DISPLAY-OF function result in any operations that involve implicit conversion (such as an assignment to, or comparison with, a national data item). Such operations assume the EBCDIC code page that is specified with the CCSID compiler option.

Conversion exceptions

Implicit or explicit conversion between national and alphanumeric data could fail and generate a severity-40 error. Failures could occur if any of the following occur:

- The code page that you specified (implicitly or explicitly) is not a valid code page
- The combination of the CCSID that you specified explicitly or implicitly (such as by using the CCSID compiler option) and the UCS-2 Unicode CCSID (specified explicitly or implicitly such as by using the National CCSID compiler option) is not supported by the operating system.

#

A character that does not have a counterpart in the target CCSID does not result in a conversion exception. Such a character is converted to a substitution character of the target code page.

The following example shows the use of the NATIONAL-OF and DISPLAY-OF intrinsic functions and the MOVE statement for converting to and from Unicode strings. It also demonstrates the need for explicit conversions when you operate on strings encoded in multiple code pages in the same program.

```
PROCESS CCSID(37)
*...
01 Data-in-Unicode          pic N(100) usage national.
01 Data-in-Greek           pic X(100).
01 other-data-in-US-English pic X(12) value "PRICE in $=".
*...
    Read Greek-file into Data-in-Greek
    Move function National-of(Data-in-Greek, 00875)
      to Data-in-Unicode
*...process Data-in-Unicode here ...
    Move function Display-of(Data-in-Unicode, 00875)
      to Data-in-Greek
    Write Greek-record from Data-in-Greek
```

The above example works correctly: Data-in-Greek is converted as data represented in CCSID 00875 (Greek) explicitly. However, the following statement would result

in an incorrect conversion (unless all the characters in the item happen to be among those with a common representation in the Greek and the English code pages):

```
Move Data-in-Greek to Data-in-Unicode
```

Data-in-Greek is converted to Unicode by this MOVE statement based on the CCSID 00037 (U.S. English) to UCS-2 conversion. This conversion would fail because Data-in-Greek is actually encoded in CCSID 00875.

If you can correctly set the CCSID compiler option to CCSID 00875 (that is, the rest of your program also handles EBCDIC data in Greek), you can code the same example correctly as follows:

```
PROCESS CCSID(00875)
*...
01 Data-in-Unicode pic N(100) usage national.
01 Data-in-Greek pic X(100).
   Read Greek-file into Data-in-Greek
*... process Data-in-Greek here ...
*... or do the following (if need to process data in Unicode)
   Move Data-in-Greek to Data-in-Unicode
*... process Data-in-Unicode
   Move function Display-of(Data-in-Unicode) to Data-in-Greek
   Write Greek-record from Data-in-Greek
```

Removing leading and/or trailing characters (TRIM, TRIML, TRIMR)

The TRIM, TRIML, TRIMR functions remove blanks or specified characters from a string. For example:

```
01 ADDR.
  05 STREET-NO PIC X(5) VALUE "120".
  05 STREET-NAME PIC X(50) VALUE "Young Street".
  05 CITY PIC X(20) VALUE "Toronto".
  05 STATE PIC X(15) VALUE "Ontario".
  05 ZIP PIC X(6) VALUE "M1C5D9".
01 ADDRESS-LINE PIC X(80).
  STRING FUNCTION TRIM(STREET-NO) " "
  FUNCTION TRIM(STREET-NAME) ", "
  FUNCTION TRIM(CITY) ", " FUNCTION TRIM(STATE) " "
  FUNCTION TRIM(ZIP) DELIMITED BY SIZE
  INTO ADDRESS-LINE.
DISPLAY ADDRESS-LINE.
```

The output would be:

```
120 Young Street, Toronto, Ontario M1C5D9
```

Evaluating Data Items (Intrinsic Functions)

Several Intrinsic Functions can be used in evaluating data items:

- CHAR and ORD for evaluating integers and single alphanumeric characters with respect to the collating sequence used in your program.
- MAX, MIN, ORD-MAX, and ORD-MIN for finding the largest and smallest items in a series of data items.
- LENGTH for finding the length of data items.
- WHEN-COMPILED for finding the date and time the program was compiled.
- TEST-DATE-TIME for determining if a date-time, alphanumeric, numeric packed, or zoned item is a valid date, time, or timestamp.

Evaluating Single Characters for Collating Sequence (CHAR, ORD)

If you want to know the ordinal position of a certain character in the collating sequence, reference the ORD function using the character in question as the argument, and ORD will return an integer representing that ordinal position. One convenient way to do this is to use the substring of a data item as the argument to ORD:

```
IF FUNCTION ORD (CUSTOMER-RECORD(1:1)) IS > 194 THEN ...
```

On the other hand, if you know what position in the collating sequence you want but do not know what character it corresponds to, then reference the CHAR function using the integer ordinal position as the argument, and CHAR will return the desired character:

```
INITIALIZE CUSTOMER-NAME REPLACING ALPHABETIC BY FUNCTION CHAR(65).
```

Returning Variable-Length Results with Alphanumeric Functions

The results of alphanumeric functions might be of varying lengths and values depending on the function arguments.

In the following example, the amount of data moved to R3 and the results of the COMPUTE statement depend on the values and sizes of R1 and R2:

```
01 R1          PIC X(10) VALUE "e".
01 R2          PIC X(05) VALUE "f".
01 R3          PIC X(05) VALUE "g".
01 R4          PIC X(20) VALUE SPACES.
01 L           PIC 99.
.
.
.
MOVE FUNCTION MEAN(R1 R2 R3) TO R4.
COMPUTE L = FUNCTION LENGTH(FUNCTION MEAN(R1 R2 R3)).
```

Here R2 is evaluated to the mean value. Therefore, assuming that the symbol b represents a blank space, the string "fbbbb" would be moved to R4 (the unfilled character positions in R4 are padded with spaces), and L evaluates to the value of 5. If R1 were the value "f", then R1 would be the mean value, and the string "fbbbbbbbbbb" would be moved to R4 (the unfilled character positions in R4 would be padded with spaces); the value 10 would be assigned to L.

You might be dealing with variable-length output from alphanumeric functions. Plan your program code accordingly. For example, you might need to think about using variable-length record files when it is possible that the records you will be writing might be of different lengths:

```

FILE SECTION.
FD OUTPUT-FILE.
01 CUSTOMER-RECORD      PIC X (80).
WORKING-STORAGE SECTION.
01 R1                    PIC X (50).
01 R2                    PIC X (70).
.
.
.
WRITE CUSTOMER-RECORD FROM FUNCTION MEAN(R1 R2 R3).

```

Finding the Largest or Smallest Data Item (MAX, MIN, ORD-MAX, ORD-MIN)

If you have two or more alphanumeric data items and want to know which data item contains the largest value (evaluated according to the collating sequence), use the MAX or ORD-MAX function, supplying the data items in question as arguments. If you want to know which item contains the smallest value, you would use the MIN or ORD-MIN function.

MAX and MIN: The MAX and MIN functions simply return the contents of one of the variables you supply. For example, with these data definitions:

```

05 Arg1 Pic x(10) Value "THOMASSON ".
05 Arg2 Pic x(10) Value "THOMAS ".
05 Arg3 Pic x(10) Value "VALLEJO ".

```

the following statement;

```
Move Function Max(Arg1 Arg2 Arg3) To Customer-record(1:10)
```

would assign "VALLEJObbb" to the first ten character positions of Customer-record.

Note: We are representing a blank with "b".

If MIN were used instead, then "THOMASbbbb" would be returned.

ORD-MAX and ORD-MIN: The functions ORD-MAX and ORD-MIN return an integer that represents the ordinal position of the argument with the largest or smallest value in the list of arguments you have supplied (counting from the left). If the ORD-MAX function were used in the example above, you would receive a syntax error message at compile time, because you would be attempting to reference a numeric function in an invalid place (see *IBM Rational Development Studio for i: ILE COBOL Reference*). The following is a valid example of the ORD-MAX function:

```
Compute x = Function Ord-max(Arg1 Arg2 Arg3)
```

This would assign the integer 3 to x, if the same arguments were used as in the previous example. If ORD-MIN were used instead, the integer 2 would be returned.

Note: This group of functions can also be used for numbers, in which case the algebraic values of the arguments are compared. For more information, see "Arithmetic Expressions" on page 169. The above examples would probably be more realistic if Arg1, Arg2 and Arg3 were instead successive elements of an array (table). For information on using table elements as function arguments, see "Processing Table Items" on page 185.

Returning Variable-Length Results with Alphanumeric Functions: The results of alphanumeric functions might be of varying lengths and values depending on the

function arguments. In the following example, the amount of data moved to R3 and the results of the COMPUTE statement depend on the values and sizes of R1 and R2:

```
01 R1 Pic x(10) value "e".
01 R2 Pic x(05) value "f".
01 R3 Pic x(20) value spaces.
01 L Pic 99.
.
.
Move Function Max(R1 R2) to R3
Compute L = Function Length(Function Max(R1 R2))
```

Here, R2 is evaluated to be larger than R1. Therefore, assuming that the symbol `␣` represents a blank space, the string "f␣␣␣␣" would be moved to R3 (the unfilled character positions in R3 are padded with spaces), and L evaluates to the value 5. If R1 were the value "g" then R1 would be larger than R2, and the string "g␣␣␣␣␣␣␣␣␣␣␣␣" would be moved to R3 (the unfilled character positions in R3 would be padded with spaces); the value 10 would be assigned to L.

You might be dealing with variable-length output from alphanumeric functions. Plan your program code accordingly. For example, you might need to think about using variable-length record files when it is possible that the records you will be writing might be of different lengths:

```
File Section.
FD Output-File.
01 Customer-Record Pic X(80).
Working-Storage Section.
01 R1 Pic x(50).
01 R2 Pic x(70).
.
.
Write Customer-Record from Function Max(R1 R2)
```

Finding the Length of Data Items (LENGTH)

The LENGTH function is useful in many programming contexts for determining the length of string items. The following ILE COBOL statement shows moving a data item, such as a customer name, into the particular field in a record that is for customer names:

```
MOVE CUSTOMER-NAME TO CUSTOMER-RECORD(1:FUNCTION LENGTH(CUSTOMER-NAME)).
```

Note: The LENGTH function can also be used on a numeric data item or a table entry.

LENGTH OF Special Register: In addition to the LENGTH function, another technique to find the length of a data item is to use the LENGTH OF special register.

There is a fundamental difference between the LENGTH OF special register and the LENGTH Intrinsic Function. FUNCTION LENGTH returns the length of an item in character positions, whereas LENGTH OF returns the length of an item in bytes. In most cases, this makes little difference except for items with a class of DBCS.

For example:

```
77 CUSTOMER-NAME PIC X(30).
77 CUSTOMER-LOCATION-ASIA PIC G(50).
```

Coding either `FUNCTION LENGTH(CUSTOMER-NAME)` or `LENGTH OF CUSTOMER-NAME` will return 30; however coding `FUNCTION LENGTH(CUSTOMER-LOCATION-ASIA)` will return 50, whereas `LENGTH OF CUSTOMER-LOCATION-ASIA` will return 100.

Whereas the `LENGTH` function can only be used where arithmetic expressions are allowed, the `LENGTH OF` special register can be used in a greater variety of contexts. For example, the `LENGTH OF` special register can be used as an argument to an Intrinsic Function that allows integer arguments. (An Intrinsic Function cannot be used as an operand to the `LENGTH OF` special register.) The `LENGTH OF` special register can also be used as a parameter in a `CALL` statement.

Finding the Date of Compilation (WHEN-COMPILED)

If you want to know the date and time the program was compiled as provided by the system on which the program was compiled, you can use the `WHEN-COMPILED` function. The result returned has 21 character positions with the first 16 positions in the format:

```
YYYYMMDDhhmmsshh
```

to show the four-digit year, month, day, and time (in hours, minutes, seconds, and hundredths of seconds) of compilation.

WHEN-COMPILED Special Register: The `WHEN-COMPILED` special register is another technique you can use to find the date and time of compilation. It has the format:

```
MM/DD/YYhh.mm.ss
```

The `WHEN-COMPILED` special register supports only a two-digit year and carries the time out only to seconds. The special register can only be used as the sending field in a `MOVE` statement.

Testing for Date-Time Data Items (TEST-DATE-TIME)

If you want to know if a date-time, alphanumeric, numeric packed, or zoned item is a valid date, time, or timestamp data item, you can use the `TEST-DATE-TIME` intrinsic function. It can be useful to test for valid date-time data items before completing a move or calculation using another date-time intrinsic function, such as `ADD-DURATION`, or `SUBTRACT-DURATION`. The following example shows how to test for date-time data items:

```
IF FUNCTION TEST-DATE-TIME (date-3 DATE) = B'1'  
  MOVE DATE-3 TO CUTOFF-DATE.
```

Working with Date and Time Durations (ADD-DURATION, FIND-DURATION, SUBTRACT-DURATION)

You can use intrinsic functions to work with durations between dates, times, and timestamps. For example, if you have two dates, and you want to know how many months fall in between the two dates, you can use the `FIND-DURATION` function to calculate this. You can also calculate due dates and stale dates (dates that have passed) using the `ADD-DURATION` and `SUBTRACT-DURATION` intrinsic functions.

For more information on using date-time data items, refer to “Working with Date-Time Data Types” on page 189.

Example of Finding the Duration Between Two Dates: The following example shows how to calculate how many days fall between two dates in date-time format:

```

01 YYYYMMDD          FORMAT DATE "@Y%m%d".
01 EXPIRY-DATE       FORMAT DATE "%m/%d/@Y" VALUE "10/31/1997".
01 DURATION          PIC S9(5).
.
.
.
MOVE FUNCTION CURRENT-DATE(1:8) TO YYYYMMDD.
COMPUTE DURATION = FUNCTION FIND-DURATION (YYYYMMDD EXPIRY-DATE DAYS).
IF DURATION <= 0 THEN
    DISPLAY 'Expiry date, ' EXPIRY-DATE ' has passed.'
END-IF.

```

The FIND-DURATION intrinsic function above subtracts YYYYMMDD from EXPIRY-DATE. If the date in YYYYMMDD becomes later than October 31, 1997, then the duration will be returned as a negative value. A duration of zero days or a negative number of days would indicate an expiry.

Assuming that the current date is November 1, 1997, the output of the above program would be:

Expiry date 10/31/1997 has passed.

Example of Calculating a Due Date: The following example shows how to calculate a due date in a date-time format:

```

01 YYYYMMDD          FORMAT DATE "@Y%m%d".
01 DATE-TIME-FORM    FORMAT DATE "%m/%d/@Y".
.
.
.
MOVE FUNCTION CURRENT-DATE(1:8) TO YYYYMMDD.
MOVE FUNCTION ADD-DURATION (YYYYMMDD DAYS 90) TO DATE-TIME-FORM.
DISPLAY 'Due Date: ' DATE-TIME-FORM.

```

Assuming that the current date is October 8, 1997, the output of the above program would be:

Due Date: 01/06/1998

Example of Calculating a Stale Date: To calculate if a date is so far in the past that it invalidates the dated piece (such as a cheque), you could use the SUBTRACT-DURATION intrinsic function as follows:

```

01 YYYYMMDD          FORMAT DATE "@Y%m%d".
01 STALE-DATE        FORMAT DATE "%m/%d/@Y".
01 cheque-date       FORMAT DATE "%m/%d/@Y" VALUE "03/09/1997".
.
.
.
MOVE FUNCTION CURRENT-DATE(1:8) TO YYYYMMDD.
MOVE FUNCTION SUBTRACT-DURATION (YYYYMMDD DAYS 180) TO STALE-DATE.
IF STALE-DATE > cheque-date THEN
    DISPLAY 'Cheque date, ' cheque-date ', is stale-dated.'
    DISPLAY 'The stale-date is: ' STALE-DATE
END-IF.

```

Assuming that the current date is October 8, 1997, the output of the above program would be:

Cheque date, 03/09/1997, is stale-dated.
The stale date is: 04/11/1997

Formatting Dates and Times Based On Locales (LOCALE-DATE, LOCALE-TIME)

A date or time can be formatted in a culturally appropriate way by using LOCALE functions. In the example below locale object (type *LOCALE) EN_US must be created in library QSYSLOCALE before running the COBOL program. For more information about how to create a locale object, refer to "Creating Locales on the i5/OS" on page 197.

The LOCALE functions take an alphanumeric item (a character string) in the format of a date, for the LOCALE-DATE intrinsic function, or in the format of a time, for the LOCALE-TIME intrinsic function and return another alphanumeric item with the date or time formatted in a culturally appropriate way.

The argument for LOCALE-DATE must be an 8-byte character string in a date format specified by the CURRENT-DATE intrinsic function. The argument for LOCALE-TIME must be an 13-byte character string in a time format specified by the CURRENT-DATE intrinsic function, positions 9 through 21.

For example:

```
SPECIAL-NAMES.  
    LOCALE "EN_US" IN LIBRARY "QSYSLOCALE" IS usa.  
:  
DISPLAY "Date is:" FUNCTION LOCALE-DATE("19970908" usa).  
DISPLAY "Time is:" FUNCTION LOCALE-TIME("06345200+0000" usa).
```

would display:

```
Date is: 08/09/1997  
Time is: 06:34:52
```

Note: To get the above result, locale USA must be in the GMT time zone.

In the above example, argument-1 for the LOCALE-DATE function, 19970908, represents the 4-digit year, followed by the month, followed by the day of the month. Argument-1 for the LOCALE-TIME function, 06345200+0000, represents the following:

- The first six digits are the hours, followed by the minutes, followed by the seconds.
- The seventh and eighth characters are the hundredths of seconds.
- The ninth character can be a plus or minus.
- The tenth and eleventh digits are the difference in hours from Greenwich Mean Time (GMT). (These two digits are not used by the LOCALE-TIME function.)
- The 12th and 13th digits are minutes.

Fixed-Point versus Floating-Point Arithmetic

Many statements in your program might involve arithmetic. For example, each of the following COBOL statements requires some kind of arithmetic evaluation:

- General arithmetic.

```
COMPUTE REPORT-MATRIX-COL = (EMP-COUNT ** .5) + 1  
ADD REPORT-MATRIX-MIN TO REPORT-MATRIX-MAX GIVING  
REPORT-MATRIX-TOT.
```

- Expressions and functions.

```
COMPUTE REPORT-MATRIX-COL = FUNCTION SQRT(EMP-COUNT) + 1  
COMPUTE CURRENT-DAY = FUNCTION DAY-OF-INTEGER(NUMBER-OF-DAYS + 1)
```

- Arithmetic comparisons.

```
IF REPORT-MATRIX-COL < FUNCTION SQRT(EMP-COUNT) + 1
IF CURRENT-DAY not = FUNCTION DAY-OF-INTEGER(NUMBER-OF-DAYS + 1)
```

For each arithmetic evaluation in your program—whether it is a statement, an Intrinsic Function, an expression, or some combination of these nested within each other—how you code the arithmetic determines whether it will be floating-point or fixed-point evaluation.

The following discussion explains when arithmetic and arithmetic comparisons are evaluated in fixed-point and floating-point. For details on the precision of arithmetic evaluations, see “Conversions and Precision” on page 165.

Floating-Point Evaluations

In general, if your arithmetic evaluation has either of the characteristics listed below, it will be evaluated by the compiler in floating-point arithmetic:

- An operand or result field is floating-point.
 - A data item is floating-point if you code it as a floating-point literal, or if you define it as USAGE COMP-1, USAGE COMP-2, or as external floating-point (USAGE DISPLAY with a floating-point PICTURE).
 - An operand that is a nested arithmetic expression or a reference to numeric Intrinsic Function results in floating-point when:
 - An argument in an arithmetic expression results in floating-point.
 - The function is a floating-point function.
 - The function is a mixed-function with one or more floating-point arguments.
- It is an argument to a floating-point function.
 - Functions like COS and SIN are floating-point functions that expect one argument. Since these functions are floating-point functions, the argument will be calculated in floating-point.

Fixed-Point Evaluations

In general, if your arithmetic operation contains neither of the characteristics listed above for floating-point, it will be evaluated by the compiler in fixed-point arithmetic. In other words, your arithmetic evaluations will be handled by the compiler as fixed-point only if all your operands are given in fixed-point, and your result field is defined to be fixed-point. Nested arithmetic expression and function references must represent fixed-point values.

Arithmetic Comparisons (Relation Conditions)

If your arithmetic is a comparison (contains a relational operator), then the numeric expressions being compared—whether they are data items, arithmetic expressions, function references, or some combination of these—are really operands (comparands) in the context of the entire evaluation. This is also true of abbreviated comparisons; although one comparand might not explicitly appear, both are operands in the comparison. When you use expressions that contain comparisons in ILE COBOL, the expression is evaluated as floating-point if at least one of the comparands is, or resolves to, floating-point; otherwise, the expression is calculated as fixed-point.

For example, consider the following statement:

```
IF (A + B) = C or D = (E + F)
```

In the preceding example there are two comparisons, and therefore four comparands. If any of the four comparands is a floating-point value or resolves to a floating-point value, all arithmetic in the IF statement will be done in floating-point; otherwise all arithmetic will be done in fixed-point.

In the case of the EVALUATE statement:

```
EVALUATE (A + D)
  WHEN (B + E) THRU C
  WHEN (F / G) THRU (H * I)
  .
  .
  .
END-EVALUATE.
```

An EVALUATE statement can be rewritten into an equivalent IF statement, or series of IF statements. In this example, the equivalent IF statements are:

```
if ( (A + D) >= (B + E) ) AND
   ( (A + D) <= C )
if ( (A + D) >= (F / G) ) AND
   ( (A + D) <= (H * I) )
```

Thus, following these rules for the IF statement above, each IF statement's comparands must be looked at to determine if all the arithmetic in that IF statement will be fixed-point or floating-point.

Examples of Fixed-Point and Floating-Point Evaluations

For the examples shown in "Fixed-Point versus Floating-Point Arithmetic" on page 183, if you define the data items in the following manner:

```
01 EMPLOYEE-TABLE.
   05 EMP-COUNT          PIC 9(4).
   05 EMPLOYEE-RECORD OCCURS 1 TO 1000 TIMES
                        DEPENDING ON EMP-COUNT.
       10 HOURS          PIC +9(5)E+99.
   .
   .
   .
01 REPORT-MATRIX-COL    PIC 9(3).
01 REPORT-MATRIX-MIN    PIC 9(3).
01 REPORT-MATRIX-MAX    PIC 9(3).
01 REPORT-MATRIX-TOT    PIC 9(3).
01 CURRENT-DAY          PIC 9(7).
01 NUMBER-OF-DAYS       PIC 9(3).
```

- These evaluations would be done in floating-point arithmetic:

```
COMPUTE REPORT-MATRIX-COL = FUNCTION SQRT(EMP-COUNT) + 1
IF REPORT-MATRIX-TOT < FUNCTION SQRT(EMP-COUNT) + 1
```

- These evaluations would be done in fixed-point arithmetic:

```
ADD REPORT-MATRIX-MIN TO REPORT-MATRIX-MAX GIVING REPORT-MATRIX-TOT.
IF CURRENT-DAY NOT = FUNCTION DAY-OF-INTEGGER((NUMBER-OF-DAYS) + 1)
COMPUTE REPORT-MATRIX-MAX =
FUNCTION MAX(REPORT-MATRIX-MAX REPORT-MATRIX-TOT)
```

Processing Table Items

You can process alphanumeric or numeric table items using intrinsic functions as long as the table item's data description is compatible with the function's argument requirements. The *IBM Rational Development Studio for i: ILE COBOL Reference* describes the required data formats for the arguments of the various Intrinsic Functions.

Use a subscript or index to reference an individual data item as a function argument. Assuming TABLE-ONE is a 3X3 array of numeric items, you can find the square root of the middle element with a statement such as:

```
COMPUTE X = FUNCTION SQRT(TABLE-ONE(2,2)).
```

Processing Multiple Table Items (ALL Subscript)

You might often need to process the data in tables iteratively. For intrinsic functions that accept multiple arguments, you can use the ALL subscript to reference all the items in the table or single dimension of the table. The iteration is handled automatically, making your code shorter and simpler.

Example 1:

This example sums a cross section of Table-Two:

```
Compute Table-Sum = FUNCTION SUM (Table-Two(ALL, 3, ALL))
```

Assuming that Table2 is a 2x3x2 array, the above statement would cause these elements to be summed:

```
Table-Two(1,3,1)  
Table-Two(1,3,2)  
Table-Two(2,3,1)  
Table-Two(2,3,2)
```

Example 2:

This example computes values for all employees.

```
01 Employee-Table.  
05 Emp-Count Pic s9(4) usage binary.  
05 Emp-Record occurs 1 to 500 times  
depending on Emp-Count.  
10 Emp-Name Pic x(20).  
10 Emp-Idme Pic 9(9).  
10 Emp-Salary Pic 9(7)v99.  
.  
.  
Procedure Division.  
Compute Max-Salary = Function Max(Emp-Salary(ALL))  
Compute I = Function Ord-Max(Emp-Salary(ALL))  
Compute Avg-Salary = Function Mean(Emp-Salary(ALL))  
Compute Salary-Range = Function Range(Emp-Salary(ALL))  
Compute Total-Payroll = Function Sum(Emp-Salary(ALL))
```

Example 3:

Scalars and array arguments can be mixed for functions that accept multiple arguments:

```
Compute Table-Median = Function Median(Arg1 Table-One(ALL))
```

What is the Year 2000 Problem?

The year 2000 problem involves using two digits to represent the year. If the date fields in your program only have the last 2 digits of the year, on 1/1/2000 the current year will be represented as 00. That means the current year will be less than the previous year because 00 is less than 99.

Century support for the 21st Century has been added to ILE COBOL. This means that if you are retrieving a year with the last 2 digits in the range of 40 – 99, the digits “19” will be added as the prefix, and you will retrieve a four-digit year in the range of 1940 – 1999. Contrastingly, if you are retrieving a year with the last 2 digits in the range of 00 – 39, the digits “20” will be added as the prefix, and you will retrieve a four-digit year in the range of 2000 – 2039.

Long-Term Solution

To take your programs through the year 9999, you must eventually:

1. Change applications to retrieve a 4-digit year instead of a 2-digit year, using one of the following methods:
 - Using the new YYYYMMDD and YYYYDDD phrases of the ACCEPT statement to obtain a 4-digit year *or*
 - Using Intrinsic Functions to get 4-digit year date (such as CURRENT-DATE, DATE-OF-INTEGGER and DAY-OF-INTEGGER) *or*
 - Using Integrated Language Environment callable services to get 4-digit year dates
2. Increase the size of the data items that contain dates so that they can store a 4-digit year, or change the data items into date data items that hold a 4-digit year.
3. Rebuild databases with 4-digit years.

However, there is a short-term solution that is easier.

Short-Term Solution

If you cannot change all of your applications and data before the year 2000 you can leave your data alone and change your application to interpret 2-digit years as 4-digit years. This type of technique is generally referred to as **windowing**. With this technique you can take a 2-digit year and determine a 4-digit year based on a predefined 100-year window. For example, given the window 1940 to 2039:

- A 2-digit year of 92 would be 1992
- A 2-digit year of 02 would be 2002.

There are two ways to do windowing in ILE COBOL. You can perform the windowing yourself with the aid of the ILE COBOL intrinsic functions, or you can let ILE COBOL perform the windowing by changing your numeric or character dates into date data items.

If you want to do the windowing yourself, ILE COBOL provides a set of century window Intrinsic Functions, which allow 2-digit years to be interpreted in a 100-year window (because each 2-digit number can only occur once in any 100-year period). You select the period, give the Intrinsic Function a 2-digit year, or a date or day with a two-digit year, and the Intrinsic Function will return the appropriate value with a 4-digit year in that 100-year window.

The ILE COBOL compiler provides three century window intrinsic functions: YEAR-TO-YYYY, DAY-TO-YYYYDDD, and DATE-TO-YYYYMMDD. The YEAR-TO-YYYY Intrinsic Function takes a 2-digit year and returns a 4-digit year in a specified 100-year window. The other two Intrinsic Functions take a date that contains a 2-digit year and returns a date with a 4-digit year in a specified 100-year window. For the DAY-TO-YYYYDDD Intrinsic Function, the date taken is a 5-digit number with a format like that of the ACCEPT FROM DAY statement. Similarly, the DATE-TO-YYYYMMDD Intrinsic Functions takes a 6-digit number with a format like that of the ACCEPT FROM DATE statement.

Form more information about the century window Intrinsic Functions, refer to the *IBM Rational Development Studio for i: ILE COBOL Reference*.

In order for ILE COBOL to perform the windowing for you, you must change your character or numeric dates into date data items. In the code fragment below there

are two numeric items that represent dates. The code is going to display a message if the current date is past the expiration date.

```
01 my-dates.  
* expiration-date is year 1997, month 10, day 9  
05 expiration-date PIC S9(6) VALUE 971009  
   USAGE PACKED-DECIMAL.  
* current-date-1 is year 2002, month 8, day 5  
05 current-date-1 PIC S9(6) VALUE 020805  
   USAGE PACKED-DECIMAL.  
IF current-date-1 > expiration-date THEN  
  DISPLAY "items date is past expiration date"  
END-IF.
```

In the above code even though 2002 is greater than 1997, the numeric values 020805 is not greater than 971009, so the IF will evaluate to FALSE, and the DISPLAY statement will not be run. However, by changing the numeric dates to date data items the DISPLAY statement will run. Notice that the size (in bytes) of both expiration-date and current-date-1 has not changed:

```
01 my-dates.  
* expiration-date is year 1997, month 10, day 9  
05 expiration-date FORMAT DATE "%y%m%d" VALUE "971009"  
   USAGE PACKED-DECIMAL.  
* current-date-1 is year 2002, month 8, day 5  
05 current-date-1 FORMAT DATE "%y%m%d" VALUE "020805"  
   USAGE PACKED-DECIMAL.  
IF current-date-1 > expiration-date THEN  
  DISPLAY "items date is past expiration date"  
END-IF.
```

Advantage of Short-Term Solution

The advantage of the short-term solution is that you need to change only a few programs, and you do not need to change your databases. This approach is cheaper, quicker, and easier than the long-term solution.

However, you can use the century window Intrinsic Functions to convert your databases or files from 2-digit year dates to 4-digit year dates. You can do this by reading in the 2-digit year dates, interpreting them to get 4-digit years, and then rewriting the data into a copy of the original that has been expanded to hold the 4-digit year data. All new data would then go into the new file or database.

Disadvantages of the Short-Term Solution

This approach buys you only a few years, depending on the application. You still must eventually change all date programs and databases.

You cannot use the century window forever because a 2-digit year can only be unique in a given 100-year period. Over time you will need more than 100 years for your data window—in fact, many companies need more than 100 years now.

The reason that the century window buys you more time is that you know in any given section of ILE COBOL code whether you are trying to figure out if a date is old (the date is in the past) or if a due date has not yet been reached (the date is in the future). You can then use this knowledge to determine how to set your century window.

There are limitations, though. For example, the century window cannot solve the problem of trying to figure out how long a customer has been with your company, if the time-span is greater than 100 years and you only have 2-digit years in your dates. Another example is sorting. All of the records that you want to sort by date

must have 4-digit year dates. For these problems and others, you need to use ACCEPT statements, Intrinsic Functions, or ILE date services which return a 4-digit year.

Working with Date-Time Data Types

Items of COBOL class date-time, include date, time, and timestamp items. These items are declared with the FORMAT clause of a data description entry. For example:

```
01 group-item.
   05 date1 FORMAT DATE "%m/%d/@Y".
   05 date2 FORMAT DATE.
   05 time1 FORMAT TIME SIZE 8 LOCALE german-locale.
   05 time2 FORMAT TIME "%H:%M:%S".
   05 time3 FORMAT TIME.
   05 timestamp1 FORMAT TIMESTAMP.
```

For items of class date-time the FORMAT clause is used in place of a PICTURE clause. In the example above, after the keyword FORMAT one of the words DATE, TIME, or TIMESTAMP appears. These words identify the category of the date-time item.

Note: The words DATE and TIME are reserved words; whereas, the word TIMESTAMP is a context-sensitive word.

After the reserved word or context-sensitive word that dictates the category of the date-time item a **format literal** may appear. A format literal is a non-numeric literal that describes the format of a date or time item.

In the case of data item date1 the %m stands for months, %d for days, and the @Y for year (including a 2-digit century). The % and @ character begin a specifier. The three specifiers used in the format literal of date1 are part of a set of specifiers documented in *IBM Rational Development Studio for i: ILE COBOL Reference*. A format literal is a combination of specifiers and separators. So again looking at date1 there are two separators, both of which are the character /.

In the above example each specifier has a pre-determined size. For example data item time2 has three specifiers: %H, %M, and %S, which stand for hours (2 digits), minutes (2 digits), and seconds (2 digits); as well as two specifiers both of which are the character :. Thus the total size of time2 is 8 characters.

Separators and specifiers may come in any order in a format literal; and must obey the following rules:

- The total size of specifiers and separators must not exceed 256 bytes.
- Separators may be of any size and can be repeated.
- Each specifier can only appear once in a format literal.
- Specifier's are restricted to certain date-time categories. For example the specifier %H (hours) can not be used for a date item.
- Specifier's can not overlap. For example you can not specify @C (single digit century) with @Y a year with a two digit century.

In the above example neither date2 nor timestamp1 have format literals specified. Items of category timestamp can not have user defined format literals; however, they do have a default format literal of @Y-%m-%d-%H.%M.%S.@Sm. For an item of

category date or time, if a format literal is not explicitly specified in the data description entry one can be specified in the SPECIAL-NAMES paragraph. An example is shown below:

```
SPECIAL-NAMES.  FORMAT OF DATE IS "@C:%y:%j",  
                FORMAT OF TIME IS "%H:%M:%S:@Sm".
```

If the above SPECIAL-NAMES paragraph had been specified in the same program as group item `group-item`, the date format of `date2` would have been `@C:%y:%j`. On the other hand, if a SPECIAL-NAMES paragraph did not exist, the format of the date item would default to ISO. An ISO date has the format `@Y-%m-%d`. The only item of category time without a format literal (implicitly or explicitly defined) is `time3`, so if the above SPECIAL-NAMES paragraph did exist, `time3` would have the time format `%H:%M:%S:@Sm`. On the other hand, if no FORMAT OF TIME clause appeared in the SPECIAL-NAMES paragraph, the format would default to ISO. An ISO time has the format `%H.%M.%S`.

By default when COPY DDS declares items of class date-time it generates a PICTURE clause for an alphanumeric item. In order to change the PICTURE clause into a FORMAT clause, several new CVTOPT parameter values have been defined, these are:

- *DATE
- *TIME
- *TIMESTAMP.

When *DATE has been specified, any DDS date data types are converted to COBOL date items, for example, a FORMAT clause is generated instead of a PICTURE clause.

In DDS to specify the format of a date field, the DATFMT keyword can be specified. The DATFMT keyword can also be specified on zoned, packed, and character fields. For these types of fields, COPY DDS would normally generate a PICTURE clause for a numeric zoned, numeric packed, and alphanumeric data item, respectively. You can force COPY DDS to generate a FORMAT clause for these items by specifying the *CVTTODATE value of the CVTOPT parameter.

For a list of the DATFMT parameters allowed for zoned, packed, and character DDS fields, and their equivalent ILE COBOL format that is generated from COPY DDS when the CVTOPT(*CVTTODATE) conversion parameter is specified, refer to "Class Date-Time" on page 440.

As mentioned above, the FORMAT clause of a data description entry defines an item of class date-time. This data description entry can also contain one or more of the following clauses:

- OCCURS
- REDEFINES
- RENAMES
- SYNCHRONIZED
- TYPEDEF
- USAGE
- VALUE.

This same data description entry can have one or more 88 (condition-names) associated with it. The VALUE clause of the condition-name can contain a THRU phrase. The following clauses can refer to a class date-time data description entry:

- LIKE
- REDEFINES

- RENAME
- TYPE.

The following code fragment shows various definitions of class date-time items:

```

01 TimestampT IS TYPEDEF
    FORMAT TIMESTAMP VALUE "1997-12-01-05.52.50.000000".
01 group-item.
    05 date1 FORMAT DATE OCCURS 3 TIMES VALUE "1997-05-08".
    05 date2 FORMAT DATE "@Y-%m-%d" VALUE "2001-09-08".
    05 date3 REDEFINES date2 FORMAT DATE.
        88 date3-key-dates VALUE "1997-05-01" THRU "2002-05-01".
    05 time1 FORMAT TIME "%H:%M" VALUE "14:10".
    05 time2 LIKE time1.
    05 timestamp1 TYPE TimestampT.

```

Each of the above clauses has various rules when used with an item of class date-time.

The SYNCHRONIZED clause can be specified for a date-time item; however, it is just treated as documentation (it does not align the item).

The USAGE clause for a date-time item can be DISPLAY or PACKED-DECIMAL for a date or time item; however, timestamps can only be USAGE DISPLAY. If a date-time item has a USAGE of PACKED-DECIMAL, then the format literal must contain specifiers only (no separators) and the specifiers must result in numeric digits.

The VALUE clause for a date-time item should be a non-numeric literal in the format of the date-time item. No checks are made at compile time to verify that the format of the VALUE clause non-numeric literal matches the FORMAT clause. It is up to the programmer to make sure the VALUE clause non-numeric literal is correct.

The level 88 (condition-names) associated with a date-time item can have a THRU phrase. The VALUE clause of a level-88 item associated with a date-time item should contain a non-numeric literal whose format matches the parent item. Level-88 items used in relational conditions result in a date-time comparison.

A LIKE clause that refers to a date-time item cannot modify its size. The LIKE clause causes the new item to inherit all the attributes of the FORMAT clause, including the SIZE and LOCALE clauses.

Date-time data items can be used with the following statements, clauses, and intrinsic functions:

- MOVE
- Implicit moves in the:
 - READ INTO
 - WRITE FROM
 - REWRITE FROM
 - RETURN INTO
 - RELEASE FROM
- Relation condition
- ACCEPT (Format 2)
- DISPLAY (All formats except extended DISPLAY)
- CALL USING and CALL GIVING
- Procedure Division USING and GIVING

- As a key in the OCCURS clause
- As a key in the SORT/MERGE
- RECORD KEY clause
- Expressions using ADDRESS OF, LENGTH OF, FORMAT OF, LOCALE OF
- The following intrinsic functions:
 - ADD-DURATION
 - CONVERT-DATE-TIME
 - EXTRACT-DATE-TIME
 - FIND-DURATION
 - SUBTRACT-DURATION
 - TEST-DATE-TIME
 - LENGTH.

Date-time data types can also be used in SORT (and MERGE) operations, however, some restrictions apply. For more information about these restrictions, refer to “Date-Time Data Type Considerations” on page 431.

MOVE Considerations for Date-Time Data Items

This section describes some of the considerations for using date-time data items in the MOVE statement and, statements where there is an implicit move (READ INTO, WRITE FROM, REWRITE FROM, RETURN INTO, and RELEASE FROM), and relation conditions:

- Translation of @p (am or pm) to upper-case (AM or PM)
- Conversion of 2-digit years to 4-digit years or centuries
- Overriding the default date window using the DATTIM process statement option
- Conversion of times to microseconds
- Time Zones.

Translation of @p to Uppercase

Time items can be defined with the @p conversion specifier. This specifier will be replaced with either AM or PM. However, the AM and PM can be any mix of upper and lower case characters. This means that in a statement that contains both source and receivers with the @p conversion specifier, the source can contain a mix of upper and lower case characters, but the receiver will always be translated into upper case characters. For example:

```
01 group-item.
  05 time1 FORMAT TIME "%I:%M @p" VALUE "06:20 am".
  05 time2 LIKE time1.
  MOVE time1 TO time2.
  DISPLAY "time2 = " time2.
```

In the above code, time1 is the source for the MOVE statement, so its @p specifier can be a mix of upper and lower case, in this example it is lowercase am. The receiver, time2, which has a format identical to time1, will contain @p in upper case. Thus the output of the above program would be:

```
time2 = 06:20 AM
```

Conversion of 2-Digit Years to 4-Digit Years or Centuries

When you are moving or comparing 2-digit years to 4-digit years or centuries, or comparing 4-digit years or centuries to 2-digit years, ILE COBOL first converts the 2-digit year using a windowing algorithm. The default windowing algorithm used is as follows:

- If a 2-digit year is moved to a 4-digit year, the century (1st 2 digits of the year) are chosen as follows:

- If the 2-digit year is greater than or equal to 40, the century used is 1900. In other words, 19 becomes the first 2 digits of the 4-digit year.
- If the 2-digit year is less than 40, the century used is 2000. In other words, 20 becomes the first 2 digits of the 4-digit year.
- If a data item with a 4-digit year or century is moved to a 2-digit year, the first 2 digits of the year (the century) are truncated. If later, the date is modified and that 2-digit year is moved back to a 4-digit year, then the algorithm just described for a 2-digit to 4-digit year move is used and inaccuracy can result. The programmer must ensure that when these types of moves are made, that inaccuracy does not result. In other words, if there is a chance that inaccuracy can result, just move 2-digit years to 2-digit years and 4-digit years to 4-digit years.

Note: When an alphanumeric data item containing a date is moved to a date-time data item, no checking or conversion is done. The programmer must ensure that the alphanumeric date being moved is in the correct format.

To show you how this works, three date moves are done in this program:

```

ID DIVISION.
PROGRAM-ID. datmoves.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SPECIAL-NAMES.
    FORMAT DATE IS '%m/%d/%y'.
DATA DIVISION.
WORKING-STORAGE SECTION.
77 date1 format date Value '07/12/39'.
77 date2 format date '@Y/%m/%d'.
77 date3 format date '%y/%m/%d'.
01 ALPHA_USA_S PIC X(08).
PROCEDURE DIVISION.
PARA1.
    move date1 to date2. 1
    display "date2 =" date2.
*
    move date2 to date3. 2
    display "date3 =" date3.
*
    move FUNCTION ADD-DURATION (date3 YEARS 1) to date2. 3
    display "date2 =" date2.

```

The output from this program is:

```

date2 =2039/07/12
date3 =39/07/12
date2 =1940/07/12

```

In move **1**, date1 (containing the value 07/12/39) is moved to date2. Because date1 contains a 2-digit year that is less than 40, and it is moved to date2, which has a 4-digit year, the century used is 2000 and the 4-digit year becomes 2039.

In move **2**, a 4-digit year is moved to a 2-digit year, and the century (1st 2 digits of the year) is just truncated.

In move **3**, a year is added to date3 and it is then moved back to a 4-digit year. Because the year that was added moved the 2-digit year of the date out of the 21st century, the date becomes a 20th century date and is inaccurate. This move shows you how the windowing algorithm works, and how inaccuracies can result when moving dates between 4-digit and 2-digit year formats.

Overriding the Default Date Window Using the DATTIM PROCESS Statement

Option: Sometimes you may not be able to avoid moving dates between 4-digit and 2-digit years, and you know that inaccuracy will result based on the default windowing algorithm that ILE COBOL uses. You can use the DATTIM process statement option to change the default date window.

The syntax of the DATTIM process statement is:

Syntax

►—DATTIM—(—4-digit base century—2-digit base year—)—————►

4-digit base century

This must be the first argument. Defines the base century that ILE COBOL uses for its windowing algorithm. If the DATTIM process statement option is not specified, 1900 is used.

The 4-digit-base-century also affects the interpretation of the @C conversion specifier. The @C conversion specifier represents a 1-digit century, whose value ranges between 0 and 9. A 0 for a 1-digit century represents a base century of 1900, 1 = 2000, ... 9 = 2800. So, a date data item whose format is @C/%y/%m and whose value is 1/12/05, represents year 2012, the first day of month 5 (May). However, 0 of @C is really equal to the 4-digit base century. Thus, a DATTIM(2200, 40) would cause 0 = 2200, 1 = 2300 ..., 9 = 3100.

2-digit base year

This must be the second argument. Defines the base year that ILE COBOL uses for its windowing algorithm. If the DATTIM process statement option is not specified, 40 is used.

The default DATTIM(1900, 40) results in a 100-year window of 1940 through 2039. To solve the problem encountered in our previous example, we could set the DATTIM option in either of the following ways:

- Specifying DATTIM(1900 70) would result in a 100-year window of 1970 through 2069
- If we assume that all 2-digit years are in the 21st century we could specify DATTIM(2000 00), which would result in a 100-year window of 2000 through 2099.

Given either of these options on a PROCESS statement in the previous example, the output would be:

```
date2 =2039/07/12
date3 =39/07/12
date2 =2040/07/12
```

The only change in the output is the result of move **3**. If you remember from the previous example, the output was date2 =1940/07/12, which was an inaccurate result from moving an updated 2-digit date to a 4-digit date based on the default ILE COBOL windowing algorithm (base century of 1900 and base year of 40).

Performance Considerations for Date-Time Relational Conditions: When a date-time item is or compared to another date-time item in ILE COBOL, it is first stripped of all separators and converted to ISO format (no separators). The two date-time items are then compared, digit by digit. In the case of a date to date comparison, both dates are converted to @Y%m%d and then compared. Times would

be converted to 8/1/07M%S, and then compared. This means that to get the best performance for date-time compares, date-time items should be in ISO format.

Performance Considerations for Date-Time MOVES: The format of a date-time item affects the amount of time a date-time move will take to complete. The three basic formats in order of best to worst performance are:

1. A date-time item whose format is equivalent to a i5/OS DDS supported date, time, or timestamp format. Within this group, ISO formats are generally the best.
2. A non-locale based date-time item whose format is not part of 1.
3. A locale-based date-time item.

#

Conversion of Times to Microseconds

A time data item can contain one or more specifiers that contain fractions of a second. The four conversion specifiers that hold fractions of a second are:

- @Sh Hundredths of a second
- @Sm Millionths of a second (microseconds)
- @So Thousandths of a second (milliseconds)
- @St Tenths of a second

For example, a time data item might have the format %H:%M:%S @So/@Sm.

If there are two time data items and one is moved to the other, the specifiers that hold fractions of a second in the sending data item are all converted to one number, representing the number of microseconds. The microseconds are then converted to the appropriate fractions of a second in the receiving time data item.

#

Time Zones: Category time and timestamp data items are affected by time zones. ILE COBOL retrieves time zone information from System i value QUTCOFFSET (Coordinated Universal Time Offset, also known as Greenwich Mean Time offset), and from LC_TOD locale category. A time data item associated with a locale uses the time zone in the tzdiff keyword of the LC_TOD locale category. A time data item which is not associated with a locale and a timestamp data item are assumed to be in the time zone of the System i. That is, in the time zone specified by the QUTCOFFSET system value.

So, for example:

```
SPECIAL-NAMES.  
    LOCALE "EN_US" IN LIBRARY "QSYSLOCALE" IS USA.  
  
:  
01 GROUP-ITEM.  
    05 SYSTEM-TIME-1 FORMAT TIME "%H:%M:%S" VALUE "14:32:10".  
    05 LOCALE-TIME-1 FORMAT TIME SIZE 8 IS LOCALE USA.  
  
:  
    MOVE SYSTEM-TIME-1 TO LOCALE-TIME-1.
```


#

The locale source for EN_US that is shipped with a i5/OS has a default tzdiff value of 0. However, this can be changed by the user by copying the locale source to a different source physical file. In the MOVE statement above, the data item SYSTEM-TIME-1 is not associated with any locale, so its time zone is dictated by the QUTCOFFSET system value. The data item LOCALE-TIME-1 is associated with locale EN_US in library QSYSLOCALE. This means that its time zone is dictated by the

LC_TOD locale category of this locale. During the MOVE statement, the resulting
time in LOCALE-TIME-1 will be adjusted by the difference in the time zones between
SYSTEM-TIME-1 and LOCALE-TIME-1.

ILE COBOL does not take into consideration Daylight Savings Time. To accommodate this, you would have to update the Coordinated Universal Time Offset in the LC_TOD locale category and in QUTCOFFSET to account for this time difference.

Other intrinsic functions that take time zones into consideration are: FIND-DURATION, LOCALE-TIME, and CONVERT-DATE-TIME.

Working With Locales

A **locale** identifies formatting information that is culturally specific. For a specific cultural region, this information describes the valid alphabetic characters, collating sequence, number formats and currency amounts, and date and time formats.

Locale information is grouped into locale categories that control specific aspects of the runtime of a program. These locale categories are:

Locale-Category Name	Behavior Affected
LC_CTYPE	Defines character types, such as upper-case, lower-case, space, digit, and punctuation. Affects the behavior of locale-based numeric-edited, date, and time items, as well as locale-based intrinsic functions.
LC_COLLATE	Defines the collating sequence.
LC_TIME	Defines the date and time conventions, such as calendar used, time zone, and days of the week. Affects the behavior of date and time data items whose format is based on a locale, and intrinsic functions that return date and time items.
LC_NUMERIC	Defines numeric formats.
LC_MONETARY	Defines the monetary names, symbols, punctuation, and other details. Affects locale-based numeric-edited items.
LC_MESSAGES	Defines the format for informative and diagnostic messages, and interactive responses.
# LC_TOD	Defines time zone difference, time zone name, and Daylight Savings Time start and end (i5/OS-specific). It also affects the behavior of locale-based time data items, intrinsic functions that return time items, and intrinsic functions that format times based on locales.
#	
#	
#	
#	
#	
LC_ALL	All locale categories, including all of those previously defined in this list. This category may include categories and cultural elements not used by ILE COBOL.

The locale categories LC_MESSAGES, LC_COLLATE, and LC_NUMERIC are not used directly by ILE COBOL. However, these categories can be SET and queried, in order that applications can use it.

#

Creating Locales on the i5/OS

On i5/OS, *LOCALE objects are created with the CRTLOCALE command, specifying the name of the file member containing the locale's definition source, and the CCSID to be used for mapping the characters of the locale's character set to their hexadecimal values.

A locale definition source member contains information about a language environment. This information is divided into a number of distinct categories which are described in the previous section. One locale definition source member characterizes one language environment.

Characters are represented in a locale definition source member with their symbolic names. The mapping between the symbolic names, the characters they represent and their associated hexadecimal values are based on the CCSID value specified on the CRTLOCALE command. The locale definition source members can be found on the i5/OS system in library QSYSLOCALE.

For more information about how locales of type *LOCALE are created, see "Using Coded Character Set Identifiers" on page 16.

Setting a Current Locale for Your Application

All ILE COBOL applications running on the AS/400 and using locales of type *LOCALE have a current locale that is scoped to the activation group of the program. The current locale determines the behavior of locale-based numeric-edited, locale-based date and time data items, and locale intrinsic functions, that do not specify a locale mnemonic-name. The current locale can be set explicitly using the SET LOCALE statement. See the *IBM Rational Development Studio for i: ILE COBOL Reference* for more information on using the SET LOCALE statement.

If the current locale is not set explicitly using SET LOCALE, it is implicitly set by the ILE COBOL runtime at program activation time. This is the same default locale that you can set using the DEFAULT keyword in Format 8 of the SET statement. Here is how the ILE COBOL runtime sets the current locale when a program is activated:

- If the user profile has a value for the LOCALE parameter other than *NONE (the default) or *SYSVAL, that value will be used for the application's current locale.
- If the value of the LOCALE parameter in the user profile is *NONE, the default ILE COBOL locale will become the current locale.
- If the value of the LOCALE parameter in the user profile is *SYSVAL, the locale associated with the system value QLOCALE will be used for the program's current locale.
- If the value of QLOCALE is *NONE, the default ILE COBOL locale will become the current locale.

The current locale used by ILE COBOL is shared with ILE C compiler and ILE C++ compiler. This means that the ILE C compiler setlocale function that changes the current locale for ILE C compiler programs also affects the current locale for ILE COBOL programs, and the other way around.

For more information about how locales of type *LOCALE are enabled, see “Using Coded Character Set Identifiers” on page 16.

Identification and Scope of Locales

The times at which a locale is identified, and the scope of its effect after being identified are:

- When a run unit is activated, the default locale is identified and remains the current locale for that run unit until it is changed within the run unit by a SET statement. After the locale has been changed from the default, the default can be made the current locale again by using the DEFAULT keyword in Format 8 of the SET statement.
- For the LOCALE-DATE and LOCALE-TIME intrinsic functions, the current locale is identified at the beginning of each statement that references any of these functions, and is used for the evaluation of the function during that statement. For more information about these intrinsic functions, refer to the *IBM Rational Development Studio for i: ILE COBOL Reference*.
- When a LOCALE phrase is used in a PICTURE clause or a FORMAT clause, and the mnemonic-name-1 is *not* specified, the current locale is identified once at the start of each statement that edits or de-edits the data item.

Note: Switching locales between the editing and de-editing of a given data item can result in unpredictable behavior. You are responsible for ensuring that the locale used for de-editing is the same as the locale used for editing.

- When a LOCALE phrase is used in a PICTURE clause or a FORMAT clause, and mnemonic-name-1 *is* specified, the current locale is the one associated with the mnemonic-name in the SPECIAL-NAMES paragraph. It must be identified anytime before the first reference in a source unit to a data item requiring its use. Its scope is that source unit.
- For a SET statement, the locale specified in the FROM phrase becomes the current locale for the run unit, until it is changed again by another SET statement.

LC_MONETARY Locale Category

In ILE COBOL, there is a subset of PICTURE-editing symbols for locale-based numeric-edited data items that correspond to definitions that can be made in the LC_MONETARY locale category. These symbols are: 9, ., \$, and cs (currency symbol). For more information about these locale-based PICTURE-editing symbols, refer to the *IBM Rational Development Studio for i: ILE COBOL Reference*. This section describes the LC_MONETARY locale category and relates each of the ILE COBOL locale-based PICTURE-editing symbols to the keywords used to define this locale category.

The LC_MONETARY category of a locale definition source file defines rules and symbols for formatting monetary numeric information. This category begins with an LC_MONETARY category header and ends with an END LC_MONETARY category trailer.

All operands for the LC_MONETARY category keywords are defined as string or integer values. String values are bounded by double-quotation marks ("). All values are separated from the keyword they define by one or more spaces. Two adjacent double-quotation marks indicate an undefined string value. A -1 indicates an undefined integer value. The following keywords are recognized in the LC_MONETARY category:

int_curr_symbol

Specifies the string used for the international currency symbol. The operand for the `int_curr_symbol` keyword is a four-character string. The first three characters contain the alphabetic international-currency symbol. The fourth character specifies a character separator between the international currency symbol and a monetary quantity. Specifies the string used for the local currency symbol. This keyword is not used by ILE COBOL.

currency_symbol

Specifies the string used for the local currency symbol. In ILE COBOL, this keyword is used along with several other keywords to format the `cs` locale-based PICTURE-editing symbol. Refer to "`p_cs_precedes`", "`p_sep_by_space`", "`n_cs_precedes`", and "`n_sep_by_space`".

mon_decimal_point

Specifies the string used for the decimal delimiter used to format monetary quantities. In ILE COBOL, this corresponds to the `.` locale-based PICTURE-editing symbol.

mon_thousands_sep

Specifies the string used for grouping digits to the left of the decimal delimiter in formatted monetary quantities.

mon_grouping

Defines the size of each group of digits in formatted monetary quantities. The operand for the `mon_grouping` keyword consists of a sequence of semicolon-separated integers. Each integer specifies the number of digits in a group. The initial integer defines the size of the group immediately to the left of the decimal delimiter. The following integers define succeeding groups to the left of the previous group. If the last digit is not `-1`, subsequent grouping is performed using the previous digit. If the last digit is `-1`, grouping is only performed for the number of groups specified.

The following is an example of the interpretation of the `mon_grouping` keyword. Assuming the value to be formatted is `123456789` and the operand for the `mon_thousands_sep` keyword is comma (`,`), the following results occur:

mon_grouping Value	Formatted Value
<code>3;-1</code>	<code>123456,789</code>
<code>3</code>	<code>123,456,789</code>
<code>3;2</code>	<code>12,34,56,789</code>
<code>3;2;-1</code>	<code>134,56,789</code>

positive_sign

Specifies the string used to indicate a nonnegative-valued formatted monetary quantity. In ILE COBOL, this corresponds to the `+` locale-based PICTURE-editing symbol.

Note: In ILE COBOL, this keyword is used along with several other keywords to format the `+` locale-based PICTURE-editing symbol. Refer to "`negative_sign`", "`p_sign_posn`", and "`n_sign_posn`".

negative_sign

Specifies the string used to indicate a negative-valued formatted monetary quantity.

Note: In ILE COBOL, this keyword is used along with several other keywords to format the + locale-based PICTURE-editing symbol. Refer to "positive_sign", "p_sign_posn", and "n_sign_posn".

int_frac_digits

Specifies an integer value representing the number of fractional digits (those after the decimal delimiter) to be displayed in a formatted monetary quantity using the int_curr_symbol value. This keyword is not used by ILE COBOL.

frac_digits

Specifies an integer value representing the number of fractional digits (those after the decimal delimiter) to be displayed in a formatted monetary quantity using the currency_symbol value. This keyword is not used by ILE COBOL.

p_cs_precedes

Specifies an integer value indicating whether the int_curr_symbol or currency_symbol string precedes or follows the value for a non-negative formatted monetary quantity. The following integer values are recognized:

- 0 Indicates that the currency symbol follows the monetary quantity.
- 1 Indicates that the currency symbol precedes the monetary quantity.

Note: In ILE COBOL, this keyword is used along with several other keywords to format the cs locale-based PICTURE-editing symbol. Refer to "currency_symbol", "p_sep_by_space", "n_cs_precedes", and "n_sep_by_space".

p_sep_by_space

Specifies an integer value indicating whether the int_curr_symbol or currency_symbol string is separated by a space from a non-negative formatted monetary quantity. The following integer values are recognized:

- 0 Indicates that no space separates the currency symbol from the monetary quantity.
- 1 Indicates that a space separates the currency symbol from the monetary quantity.
- 2 Indicates that a space separates the currency symbol and the positive_sign string, if adjacent.

Note: In ILE COBOL, this keyword is used along with several other keywords to format the cs locale-based PICTURE-editing symbol. Refer to "currency_symbol", "p_cs_precedes", "n_cs_precedes", and "n_sep_by_space".

n_cs_precedes

Specifies an integer value indicating whether the int_curr_symbol or currency_symbol string precedes or follows the value for a negative formatted monetary quantity. The following integer values are recognized:

- 0 Indicates that the currency symbol follows the monetary quantity.
- 1 Indicates that the currency symbol precedes the monetary quantity.

Note: In ILE COBOL, this keyword is used along with several other keywords to format the cs locale-based PICTURE-editing symbol. Refer to "currency_symbol", "p_cs_precedes", "p_sep_by_space", and "n_sep_by_space".

n_sep_by_space

Specifies an integer value indicating whether the `int_curr_symbol` or `currency_symbol` string is separated by a space from a negative formatted monetary quantity. The following integer values are recognized:

- 0 Indicates that no space separates the currency symbol from the monetary quantity.
- 1 Indicates that a space separates the currency symbol from the monetary quantity.
- 2 Indicates that a space separates the currency symbol and the `negative_sign` string, if adjacent.

Note: In ILE COBOL, this keyword is used along with several other keywords to format the `cs` locale-based PICTURE-editing symbol. Refer to "`currency_symbol`", "`p_cs_precedes`", "`p_sep_by_space`", and "`n_cs_precedes`".

p_sign_posn

Specifies an integer value indicating the positioning of the `positive_sign` string for a non-negative formatted monetary quantity. The following integer values are recognized:

- 0 Indicates that parenthesis enclose both the monetary quantity and the `int_curr_symbol` or `currency_symbol` string.
- 1 Indicates that the `positive_sign` string precedes the quantity and the `int_curr_symbol` or `currency_symbol` string.
- 2 Indicates that the `positive_sign` string follows the quantity and the `int_curr_symbol` or `currency_symbol` string.
- 3 Indicates that the `positive_sign` string immediately precedes the `int_curr_symbol` or `currency_symbol` string.
- 4 Indicates that the `positive_sign` string immediately follows the `int_curr_symbol` or `currency_symbol` string.

Note: In ILE COBOL, this keyword is used along with several other keywords to format the `+` locale-based PICTURE-editing symbol. Refer to "`positive_sign`", "`negative_sign`", and "`n_sign_posn`".

n_sign_posn

Specifies an integer value indicating the positioning of the `negative_sign` string for a negative formatted monetary quantity. The following integer values are recognized:

- 0 Indicates that parenthesis enclose both the monetary quantity and the `int_curr_symbol` or `currency_symbol` string.
- 1 Indicates that the `negative_sign` string precedes the quantity and the `int_curr_symbol` or `currency_symbol` string.
- 2 Indicates that the `negative_sign` string follows the quantity and the `int_curr_symbol` or `currency_symbol` string.
- 3 Indicates that the `negative_sign` string immediately precedes the `int_curr_symbol` or `currency_symbol` string.
- 4 Indicates that the `negative_sign` string immediately follows the `int_curr_symbol` or `currency_symbol` string.

Note: In ILE COBOL, this keyword is used along with several other keywords to format the + locale-based PICTURE-editing symbol. Refer to "positive_sign", "negative_sign", and "p_sign_posn".

Producing Unique Monetary Formats—Example

A unique customized monetary format can be produced by changing the value of a single statement. For example, the following table shows the results of using all combinations of defined values for the p_cs_precedes, p_sep_by_space, and p_sign_posn statements:

Table 14. Results of Various Locale Variable Value Combinations

		p_sep_by_space		
		2	1	0
p_cs_precedes = 1	p_sign_posn = 0	(\$1.25)	(\$ 1.25)	(\$1.25)
	p_sign_posn = 1	+ \$1.25	+\$ 1.25	+\$1.25
	p_sign_posn = 2	\$1.25 +	\$ 1.25+	\$1.25+
	p_sign_posn = 3	+ \$1.25	+\$ 1.25	+\$1.25
	p_sign_posn = 4	\$ +1.25	+\$ 1.25	+\$1.25
p_cs_precedes = 0	p_sign_posn = 0	(1.25 \$)	(1.25 \$)	(1.25\$)
	p_sign_posn = 1	+1.25 \$	+1.25 \$	+1.25\$
	p_sign_posn = 2	1.25\$ +	1.25 \$+	1.25\$+
	p_sign_posn = 3	1.25+ \$	1.25 +\$	1.25+\$
	p_sign_posn = 4	1.25\$ +	1.25 \$+	1.25\$+

LC_MONETARY—Example

The following is an example of the LC_MONETARY category listed in a locale definition source file:

```
LC_MONETARY
#
int_curr_symbol    "<U><S><D>"
currency_symbol    "<dollar-sign>"
mon_decimal_point  "<period>"
mon_thousands_sep "<comma>"
mon_grouping       3;-1
positive_sign      "<plus-sign>"
negative_sign      "<hyphen>"
int_frac_digits    2
frac_digits        2
p_cs_precedes      1
p_sep_by_space     2
n_cs_precedes      1
n_sep_by_space     2
p_sign_posn        3
n_sign_posn        3
#
END LC_MONETARY
```

LC_TIME Category

In ILE COBOL the LC_TIME category is used to format date and time items that are based on a locale. Like other locale categories, LC_TIME consists of a series of keywords followed by their operands. The LC_TIME keyword "d_fmt" specifies the format of locale based date data items. The LC_TIME keyword "t_fmt" specifies the format of locale based time data items.

The following section gives a more detailed description of all the LC_TIME category keywords, including those not currently used by ILE COBOL. The descriptions below mention several conversion specifiers such as %a and %c that are currently not supported by ILE COBOL.

The LC_TIME category of a locale definition source file defines rules and symbols for formatting time and date information. This category begins with an LC_TIME category header and terminates with an END LC_TIME category trailer.

All operands for the LC_TIME category keywords are defined as string or integer values. String values are bounded by double quotation marks (""). All values are separated from the keyword they define by one or more spaces. Two adjacent double quotation marks indicate an undefined string value. A -1 indicates an undefined integer value. Field descriptors are used by commands and subroutines that query the LC_TIME category to represent elements of time and date formats. The following keywords are recognized in the LC_TIME category:

abday Defines the abbreviated weekday names corresponding to the %a field descriptor. Recognized values consist of seven semicolon-separated strings. The first string corresponds to the abbreviated name for the first day of the week (Sun), the second to the abbreviated name for the second day of the week, and so on.

day Defines the full spelling of the weekday names corresponding to the %A field descriptor. Recognized values consist of seven semicolon-separated strings. The first string corresponds to the full spelling of the name of the first day of the week (Sunday), the second to the name of the second day of the week, and so on. This keyword is not used by ILE COBOL.

abmon Defines the abbreviated month names corresponding to the %b field descriptor. Recognized values consist of 12 semicolon-separated strings. The first string corresponds to the abbreviated name for the first month of the year (Jan), the second to the abbreviated name for the second month of the year, and so on. This keyword is not used by ILE COBOL.

mon Defines the full spelling of the month names corresponding to the %B field descriptor. Recognized values consist of 12 semicolon-separated strings. The first string corresponds to the full spelling of the name for the first month of the year (January), the second to the full spelling of the name for the second month of the year, and so on. This keyword is not used by ILE COBOL.

d_t_fmt Defines the string used for the standard date and time format corresponding to the %c field descriptor. The string can contain any combination of characters, field descriptors, or escape sequences. This keyword is not used by ILE COBOL.

d_fmt Defines the string used for the standard date format corresponding to the %x field descriptor. The string can contain any combination of characters, field descriptors, or escape sequences. Following is an example of how the d_fmt keyword can be constructed:

%D The %D indicates a %m/%d/%y date format.

%d-%m-%y

%m/%d/%Y

t_fmt Defines the string used for the standard time format corresponding to the

%X field descriptor. The string can contain any combination of characters, field descriptors, or escape sequences. Following is an example of how the `t_fmt` keyword can be constructed:

`%H:%M:%S`

`%H.%M.%S`

am_pm

Defines the strings used to represent ante meridian (before noon) and post meridian (after noon) corresponding to the `%p` field descriptor. Recognized values consist of two strings separated by a ; (semicolon). The first string corresponds to the ante meridian designation, the last string to the post meridian designation.

t_fmt_ampm

Defines the string used for the standard 12-hour time format that includes an `am_pm` value (`%p` field descriptor). This statement corresponds to the `%r` field descriptor. The string can contain any combination of characters and field descriptors. This keyword is not used by ILE COBOL.

era Defines how the years are counted and displayed for each era in a locale, corresponding to the `%E` field descriptor modifier. For each era, there must be one string in the following format:

direction:offset:start_date:end_date:era_name:era_format

This keyword is not used by ILE COBOL.

The variables for the era-string format are defined as follows:

direction

Specifies a - (minus sign) or + (plus sign) character. The plus character indicates that years count in the positive direction when moving from the start date to the end date. The minus character indicates that years count in the negative direction when moving from the start date to the end date.

offset Specifies a number representing the first year of the era.

start_date

Specifies the starting date of the era in the `yyyy/mm/dd` format, where `yyyy`, `mm`, and `dd` are the year, month, and day, respectively. Years prior to the year AD 1 are represented as negative numbers. For example, an era beginning March 5th in the year 100 BC would be represented as `-100/03/05`.

end_date

Specifies the ending date of the era in the same form used for the `start_date` variable or one of the two special values `-*` or `+`. A `-*` value indicates that the ending date of the era extends backward to the beginning of time. A `+` value indicates that the ending date of the era extends forward to the end of time. Therefore, the ending date can be chronologically before or after the starting date of the era. For example, the strings for the Christian eras AD and BC would be entered as follows:

```
+:0:0000/01/01:+:AD:%0 %N
+:1:-0001/12/31:-*:BC:%0 %N
```

era_name

Specifies a string representing the name of the era that is substituted for the `%EC` field descriptor.

era_format

Specifies a string for formatting the %EY field descriptor.

An **era** value consists of one string for each era. If more than one era was specified, each era string is separated by a ; (semicolon).

era_d_fmt

Defines the string used to represent the date in alternate-era format corresponding to the %Ex field descriptor. The string can contain any combination of characters and field descriptors.

era_t_fmt

Defines the string used to represent the time in alternate-era format corresponding to the %EX field descriptor. The string can contain any combination of characters and field descriptors.

era_d_t_fmt

Defines the string used to represent the date and time in alternate-era format corresponding to the %Ec field descriptor. The string can contain any combination of characters and field descriptors.

alt_digits

Defines alternate strings for digits corresponding to the %O field descriptor. Recognized values consist of a group of strings separated by ; (semicolons). The first string represents the alternate string for zero, the second string represents the alternate string for one, and so on. A maximum of 100 alternate strings can be specified.

Escape Sequences

The following are escape sequences allowed for the d_t_fmt, d_fmt, and t_fmt keyword values:

- `\\` Represents the backslash character.
- `\a` Represents the alert character.
- `\b` Represents the backspace character.
- `\f` Represents the form-feed character.
- `\n` Represents the newline character.
- `\r` Represents the carriage-return character.
- `\t` Represents the tab character.
- `\v` Represents the vertical-tab character.

LC_TIME Example

The following is an example of a LC_TIME category in a locale definition source file:

```
LC_TIME
#
#Abbreviated weekday names (%a)
abday  "<S><u><n>"; "<M><o><n>"; "<T><u><e>"; "<W><e><d>"; \
        "<T><h><u>"; "<F><r><i>"; "<S><a><t>"
#
#Full weekday names (%A)
day    "<S><u><n><d><a><y>"; "<M><o><n><d><a><y>"; \
        "<T><u><e><s><d><a><y>"; "<W><e><d><n><e><s><d><a><y>"; \
        "<T><h><u><r><s><d><a><y>"; "<F><r><i><d><a><y>"; \
        "<S><a><t><u><r><d><a><y>"
#
#Abbreviated month names (%b)
abmon  "<J><a><n>"; "<F><e><b>"; "<M><a><r>"; "<A><p><r>"; \
```

```

        "<M><a><y>"; "<J><u><n>"; "<J><u><l>"; "<A><u><g>"; \
        "<S><e><p>"; "<O><c><t>"; "<N><o><v>"; "<D><e><c>"
#
#Full month names (%B)
mon      "<J><a><n><u><a><r><y>"; "<F><e><b><r><u><a><r><y>"; \
        "<M><a><r><c><h>"; "<A><p><r><i><l>"; "<M><a><y>"; \
        "<J><u><n><e>"; "<J><u><l><y>"; "<A><u><g><u><s><t>"; \
        "<S><e><p><t><e><m><b><e><r>"; "<O><c><t><o><b><e><r>"; \
        "<N><o><v><e><m><b><e><r>"; "<D><e><c><e><m><b><e><r>"
#
#Date and time format (%c)
d_t_fmt  "%a_%bf%d %H:%M:%S %Y"
#
#Date format (%x)
d_fmt    "%m/%d/%y"
#
#Time format (%X)
t_fmt    "%H:%M:%S"
#
#Equivalent of AM/PM (%p)
am_pm    "<A><M>"; "<P><M>"
#
#12-hour time format (%r)
t_fmt_ampm "%I:%M:%Sm%p"
#
era      "+:0:0000/01/01:++:AD:%EC"; \
        "+:1:-0001/12/31:-*:BC:%Ey";
era_d_fmt ""
alt_digits "<0><t><h>"; "<1><s><t>"; "<2><n><d>"; "<3><r><d>"; \
        "<4><t><h>"; "<5><t><h>"; "<6><t><h>"; "<7><t><h>"; \
        "<8><t><h>"; "<9><t><h>"; "<1><0><t><h>"
#
END LC_TIME

```

LC_TOD Category

In ILE COBOL, the LC_TOD locale category dictates the timezone for a locale based time item. In particular the tzdiff keyword specifies the difference between the local time and Greenwich mean time. This information is used when moving or comparing a locale based time item to another time (locale or non-locale based). The tzdiff keyword is the only LC_TOD keyword currently used by ILE COBOL.

The LC_TOD category defines the rules used to define the start and end time of daylight savings time, the difference between local time and Greenwich Mean time, the time zone name, and the daylight savings time name. This category is an IBM extension and must appear after all other category definitions in the source file.

All the operands for the LC_TOD category are defined as string or integer values. String values are bounded by double-quotation marks (""). All values are separated from the keyword they define by one or more spaces. Two adjacent double-quotation marks indicate an undefined string value. A 0 (zero) indicates an undefined integer value. The following keywords are recognized in the LC_TOD category.

tzdiff Specifies an integer value representing the time zone difference in minutes. It is the difference between the local time and Greenwich mean time.

tname Specifies the string used for the time zone name.

dstname Specifies the string used for the daylight savings time name.

dststart Specifies a set of four integers representing the start date for the daylight

savings time. The operand for the `dststart` keyword consists of a sequence of four comma-separated integers in the following format:

`month,week,day,time`

The variables for the `dststart` format are defined as:

- month* Specifies an integer value representing the month of the year when Daylight Savings Time (DST) takes effect. This value ranges from 1 to 12, with 1 corresponding to January, and 12 corresponding to December.
- week* Specifies an integer value representing the week of the month when DST takes effect. This value ranges from -4 to 4, with -4 corresponding to the fourth week of the month counting from the end of the month and 4 corresponding to the fourth week of the month counting from the beginning of the month.
- day* Specifies an integer value representing the day of the month when DST takes effect or if the `week` keyword is not 0 (zero), then this is the day of the week when DST takes effect. This value ranges from 1 to the last day of the month or 1 to the last day of the week.
- time* Specifies an integer value representing the number of seconds after 12 midnight, local standard time, when DST takes effect. This value ranges from 0 to 86399.

dstend

Specifies a set of four integers representing the end date for the daylight savings time. The operand for the `dstend` keyword consists of a sequence of four comma-separated integers in the following format:

`month,week,day,time`

The variables for the `dstend` format are defined as:

- month* Specifies an integer value representing the month of the year when Daylight Savings Time (DST) ends. This value ranges from 1 to 12, with 1 corresponding to January, and 12 corresponding to December.
- week* Specifies an integer value representing the week of the month when DST ends. This value ranges from -4 to 4, with -4 corresponding to the fourth week of the month counting from the end of the month and 4 corresponding to the fourth week of the month counting from the beginning of the month.
- day* Specifies an integer value representing the day of the month when DST ends or if the `week` keyword is not 0 (zero), then this is the day of the week when DST ends. This value ranges from 1 to the last day of the month or 1 to the last day of the week.
- time* Specifies an integer value representing the number of seconds after 12 midnight, local standard time, when DST takes effect. This value ranges from 0 to 86399.

dstshift

Specifies an integer value representing the daylight savings time shift in seconds.

LC_TOD Example

The following is an example of a `LC_TOD` category in a locale definition source file:

```

LC_TOD
#
tzdiff      360
tname       "<C><e><n><t><r><a><l>"
dstname     "<P><D><T>"
#Set daylight savings time to start on 3rd week of October at
#midnight on Saturday.
dststart    10,3,6,0
#Set daylight savings time to end on April 23, at midnight.
dstend      4,0,23,0
dstshift    3600
#
END LC_TOD

```

Manipulating null-terminated strings

You can construct and manipulate null-terminated strings (for example, strings that are passed to or from a C program) by various mechanisms.

For example, you can:

- Use null-terminated literal constants (Z". . . ").
- Use an INSPECT statement to count the number of characters in a null-terminated string:

```

MOVE 0 TO char-count
INSPECT source-field TALLYING char-count
      FOR CHARACTERS
      BEFORE X"00"

```

- Use an UNSTRING statement to move characters in a null-terminated string to a target field, and get the character count:

```

WORKING-STORAGE SECTION.
01 source-field          PIC X(1001).
01 char-count           COMP PIC 9(4).
01 target-area.
   02 individual-char OCCURS 1 TO 1000 TIMES DEPENDING ON char-count
                        PIC X.
. . .
PROCEDURE DIVISION.
. . .
   UNSTRING source-field DELIMITED BY X"00"
           INTO target-area
           COUNT IN char-count
   ON OVERFLOW
       DISPLAY "source not null terminated or target too short"
. . .
END-UNSTRING

```

- Use a SEARCH statement to locate trailing null or space characters. Define the string being examined as a table of single characters.
- Check each character in a field in a loop (PERFORM). You can examine each character in the field by using a reference modifier such as source-field (I:1).

"Example: null-terminated strings"

RELATED REFERENCES

Null-terminated nonnumeric literals (*ILE COBOL Language Reference*)

Example: null-terminated strings

The following example shows several ways in which you can process null-terminated strings.

```

01 L pic X(20) value z'ab'.
01 M pic X(20) value z'cd'.
01 N pic X(20) value z'xyz'.
01 N-Length pic 99 value zero.
01 X pic X(20) value z'xyz'.
01 X pic X(20).
01 Y pic X(13) value 'Hello, World!'.
. . .
* Display null-terminated string
  Inspect N tallying N-length
  for characters before initial x'00'
  Display 'N: ' N(1:N-Length) ' Length: ' N-Length
. . .
* Move null-terminated string to alphanumeric, strip null
  Unstring N delimited by X'00' into X
. . .
* Create null-terminated string
  String Y delimited by size
  X'00' delimited by size
  into N.
. . .
* Concatenate two null-terminated strings to produce another
  String L delimited by x'00'
  M delimited by x'00'
  X'00' delimited by size
  into N.

```

Chapter 9. Calling and Sharing Data Between ILE COBOL Programs

Sometimes an application is simple enough to be coded as a single, self-sufficient program. In many cases, however, an application's solution will consist of several, separately compiled programs used together.

The IBM i system provides communication between ILE COBOL programs, and between ILE COBOL and non-ILE COBOL programs.

This chapter describes:

- Various methods used to call another ILE COBOL program
- How control is transferred back to the calling program once the called program has finished running
- How to pass data between the calling program and called program
- How to cancel an ILE COBOL program.

Run Time Concepts

A program object is created from one or more module objects. Each program object has one and only one module object designated as the main entry point when the program object is activated. When a module object is created by the ILE COBOL compiler, a PEP is generated which calls the outermost ILE COBOL program contained in the compilation unit. When you bind multiple module objects together to create a program object, you must specify which module object contains the PEP of the program object being created. You do this by identifying the module object in the ENTMOD parameter of the CRTPGM command. The PEP of this module object becomes the PEP for the program object.

When a program object is activated using a dynamic program call, the PEP is given control. The PEP then calls the UEP which is the outermost ILE COBOL program in the module object that is to be performed first. Refer to the *ILE Concepts* book for a discussion on PEPs and UEPs.

Activation and Activation Groups

The process of getting a program object or service program ready to run is called **activation**. Activation is done by the system when a program object is called. Because service programs are not called in their entirety, they are activated during the call to a program object that directly or indirectly requires their services. ILE procedures within service programs are called using static procedure calls; they cannot be called using dynamic program calls.

Activation does the following functions:

- It uniquely allocates the static data needed by the program object or service program
- It changes the symbolic links to used service programs into links to physical addresses.

When activation allocates the storage necessary for the static variables used by a program object, the space is allocated from an **activation group**. Each activation

group has a name. The name of the activation group is supplied by the user (or by the system when *NEW is specified). You can specify, at the time the program object or service program is created using CRTPGM or CRTSRVPGM, the activation group in which the program object or service program is to be activated. Refer to *ILE Concepts* for a more detailed discussion on activation and activation groups.

COBOL Run Unit

A COBOL **run unit** is a set of one or more programs that function as a unit at run time to provide a problem solution. A COBOL run unit is an independent entity that can be executed without communicating with, or being coordinated with, any other run unit except that it can process data files and messages or set and test switches that are used by other run units. A run unit can also contain program objects and service programs created from module objects that are created from the compilation of programs written in languages other than ILE COBOL.

In ILE, a COBOL run unit is composed of program objects and service programs that all run in a single ILE activation group. To preserve OPM COBOL/400 compatible run unit semantics, your ILE COBOL application must meet the following conditions:

- Each ILE COBOL compilation unit must be compiled and then bound into a single program object.
- All run unit participants (ILE COBOL or other ILE programs/procedures) must run in a single ILE activation group.

Note: You should use a named ILE activation group in which to run your application in order to properly maintain COBOL run unit semantics. By using a named ILE activation group for all participating program objects, you need not specify a particular ILE COBOL program object to be the main program before your application is run.

On the other hand, if a particular ILE COBOL program object is known to be main program before your application is run, you can specify the *NEW attribute for the ACTGRP option when creating a *PGM object using the ILE COBOL program as the UEP. All other participating program objects should specify the *CALLER attribute for the ACTGRP option.

- The oldest invocation of the ILE activation group (corresponding to the run unit) must be that of ILE COBOL. This is the main program of the run unit.

If these conditions are not met, there may be a control boundary that binds the scope of the STOP RUN so that the state of the entire application is not refreshed.

Note: The above condition dictates that an ILE COBOL program running in the *DFTACTGRP is generally run in a run unit that is not compatible with an OPM COBOL/400 run unit.

Control Boundaries

All ILE languages, including ILE COBOL, use a common mechanism called the **call stack** for transferring control to and from called ILE procedures or OPM program objects. The call stack consists of a last-in, first-out list of call stack entries, one entry for each called ILE procedure or program object. Each call stack entry has

information about the automatic variables for the ILE procedure, and other resources scoped to the call stack entry such as condition handlers and cancel handlers.

In ILE COBOL, each ILE COBOL program or nested program that is called has one call stack entry. Each declarative that is called also has its own call stack entry.

A call adds a new entry on the stack for the called ILE procedure or OPM program object and passes control to the called object. A return removes the call stack entry and passes control back to the called ILE procedure or program object in the previous call stack entry.

In ILE, you can create an application that runs program objects in multiple activation groups. You can call an ILE COBOL program object that is running in a different activation group from that of the calling program. In this case, the call stack entry for the called program object is known as a **control boundary**. A control boundary is defined as any ILE call stack entry for which the immediately preceding call stack entry is for an ILE procedure or program object in a different activation group. An ILE call stack entry for which the immediately preceding call stack entry is for an OPM program object is also a control boundary.

If the called program object is the first program object to be activated in a particular activation group, then its call stack entry is known as a **hard control boundary**. If the called program object, which is a control boundary, is not the first program object to be activated in an activation group, then its call stack entry is known as a **soft control boundary**. The main program of a run unit that is compatible with and OPM COBOL/400 run unit is found at the hard control boundary of the activation group.

When a STOP RUN statement (or a GOBACK statement in a main ILE COBOL program) is encountered in a called ILE COBOL program, control is transferred to the caller of the control boundary. In a run unit that is compatible with an OPM COBOL/400 run unit, STOP RUN will end the run unit.

An implicit COMMIT operation is performed on files under commitment control if commitment control is scoped to the activation group and the activation ends normally with no errors closing the files. A ROLLBACK operation is performed if the activation group ends abnormally or there are errors closing the files. Nothing happens if commitment control is scoped to the job.

The control boundary is also where an unhandled error is turned into a function check. When the function check is again unhandled, then, at the control boundary, it will be changed to the generic ILE failure condition, CEE9901, and sent to the caller of the control boundary.

Main Programs and Subprograms

The first program in the activation group to be activated begins the COBOL run unit, and is the **main program**. The main program is at the hard control boundary of the activation group. No specific source statements or options identify an ILE COBOL program as a main program or a subprogram.

A **subprogram** is a program in the run unit below the main program in the call stack. For more information about program stacks and other terms concerning interprogram communication, see the *CL Programming* manual.

Initialization of Storage

The first time an ILE COBOL program in a run unit is called, its storage is initialized. Storage is initialized again under the following conditions:

- The PROGRAM-ID paragraph of the ILE COBOL program possesses the INITIAL clause. Storage is reinitialized each time the program is called.
- The run unit is ended, then reinitiated.
- The program is canceled (using the CANCEL statement for ILE COBOL) and then called again.
- The end of section-name and paragraph-name branching addresses (set by previous PERFORM statements) are always re-initialized each time the program is called.

Transferring Control to Another Program

In the Procedure Division, a program can call another program (generally called a subprogram in COBOL terms), and this called program may itself call another program. The program that calls another program is referred to as the **calling** program, and the program it calls is referred to as the **called** program.

The called ILE COBOL program starts running at the top of the non-declarative part of the Procedure Division. If a called ILE COBOL program does not have a Procedure Division or does not have a non-declarative part in the Procedure Division, it will simply return to the calling ILE COBOL program.

When the called program processing is completed, the program can either transfer control back to the calling program or end the run unit. The run unit is ended after STOP RUN is issued and the nearest control boundary is a hard control boundary. If the nearest control boundary is a soft control boundary, then control returns to the caller of the control boundary but the run unit remains active.

A called program must not directly or indirectly call its caller (such as program X calling program Y; program Y calling program Z; and program Z then calling program X). This is called a **recursive** call. ILE COBOL does **not** allow recursion in non-recursive main programs or subprograms. Recursive calls are only allowed if you code the RECURSIVE clause on the recursively invoked program's PROGRAM-ID paragraph. If you try to recursively call a COBOL program that does not have the RECURSIVE clause coded on its PROGRAM-ID paragraph, a run time error message is generated.

Calling an ILE COBOL Program

To call another ILE COBOL program, you can use one of the following methods:

- Calls to nested programs
- Static procedure calls
- Dynamic program calls.

Calls to nested programs allow you to create applications using structured programming techniques. They can also be used in place of PERFORM procedures to prevent unintentional modification of data items. Calls to nested programs can be made using either the CALL *literal* or CALL *identifier* statement. For more information on nested programs, see "Calling Nested Programs" on page 217.

A **static procedure call** transfers control to a called ILE COBOL program that is bound by copy or by reference into the same program object as the calling ILE

COBOL program. Static procedure calls can be made using the *CALL literal* or *CALL procedure-pointer* statements. A static procedure call can be used to call any of the following:

- An ILE procedure within the same module object
- A nested ILE COBOL program (using *CALL literal*)
- An ILE procedure in a separate module object that has been bound to the calling ILE COBOL program
- An ILE procedure in a separate service program.

A **dynamic program call** transfers control to a called ILE COBOL program that has been bound into a separate program object from the calling ILE COBOL program. The called ILE COBOL program must be the UEP of the program object. Only the ILE COBOL program that is the UEP of the program object can be called from another ILE COBOL program that is in a different program object. ILE COBOL programs, other than the one designated as the UEP, are only visible within the program object. With a dynamic program call, the called program object is activated the first time it is called within the activation group. Dynamic program calls can be made using the *CALL literal*, *CALL identifier*, or *CALL procedure-pointer-data-item* statements. Use the *SET procedure-pointer-data-item TO ENTRY program-object-name* statement to set the *procedure-pointer-data-item* before using the *CALL procedure-pointer-data-item* statement.

For additional information on static procedure calls and dynamic program calls, see “Using Static Procedure Calls and Dynamic Program Calls” on page 220.

Identifying the Linkage Type of Called Programs and Procedures

When calling another ILE COBOL program that is not in the same module object as the calling program and the call is made through a *CALL literal* statement, you must specify whether the called program is an ILE program object or an ILE procedure.

You identify whether you are calling a program object or a procedure by specifying the linkage type of the call.

The LINKAGE type of call can be specified explicitly or it can be forced by specifying a phrase that is associated with a particular linkage. For example, the IN LIBRARY phrase forces a call to be a LINKAGE program. In the instances where there is not a phrase that forces a linkage, there are three ways to explicitly specify a linkage. They are listed in order of precedence:

1. The LINKAGE phrase of the CALL, CANCEL, or SET...ENTRY statements
 - To call or cancel a program object, specify LINKAGE TYPE IS PROGRAM in the CALL, CANCEL, or SET...ENTRY statement.

```
PROCEDURE DIVISION.  
  
:  
CALL LINKAGE TYPE IS PROGRAM literal-1  
  
:  
CALL LINKAGE PROGRAM literal-2 IN LIBRARY literal-3  
  
:  
CANCEL LINKAGE PROGRAM literal-2 IN LIBRARY literal-3
```


3. the LINKLIT parameter of the CRTCBMOD and CRTBNDCBL commands, or the associated PROCESS statement option.
 - The LINKLIT parameter of the CRTCBMOD and CRTBNDCBL commands allows you to specify, at compile time, the linkage type for all external CALL *literal-1*, CANCEL *literal-1*, or SET *procedure-pointer-data-item* TO ENTRY *literal-1* statements in the ILE COBOL program. You do not need to specify the LINKAGE TYPE clause in the SPECIAL-NAMES paragraph or the LINKAGE TYPE phrase with the CALL, CANCEL, or SET...ENTRY statement when the linkage has been defined by the LINKLIT parameter of CRTCBMOD or CRTBNDCBL.
 - To create a module that calls program objects, type:


```
CRTCBMOD MODULE(MYLIB/XMPLE1)
SRCFILE(MYLIB/QCBLLSRC) SRCMBR(XMPLE1)
LINKLIT(*PGM)
```
 - To create a module which calls procedures, type:


```
CRTCBMOD MODULE(MYLIB/XMPLE1)
SRCFILE(MYLIB/QCBLLSRC) SRCMBR(XMPLE1)
LINKLIT(*PRC)
```
 - You code the CALL and CANCEL statements as follows when using the LINKLIT parameter of CRTCBMOD to specify linkage type:


```
PROCEDURE DIVISION.
:
:           CALL literal-1.
:
:           CANCEL literal-1.
```

Calling Nested Programs

Nested programs give you a method to create modular functions for your application and maintain structured programming techniques. Nested programs allow you to define multiple separate functions, each with its own controlled scope, within a single compilation unit. They can be used like PERFORM procedures with the additional ability to protect **local** data items.

Nested programs are contained in the same module as their calling program when they are compiled. Therefore, nested programs always run in the same activation group as their calling programs.

Structure of Nested Programs

An ILE COBOL program may **contain** other ILE COBOL programs. The **contained** programs may themselves contain yet other programs. A contained program may be **directly** or **indirectly** contained within a program.

Figure 51 on page 218 describes a nested program structure with directly and indirectly contained programs.

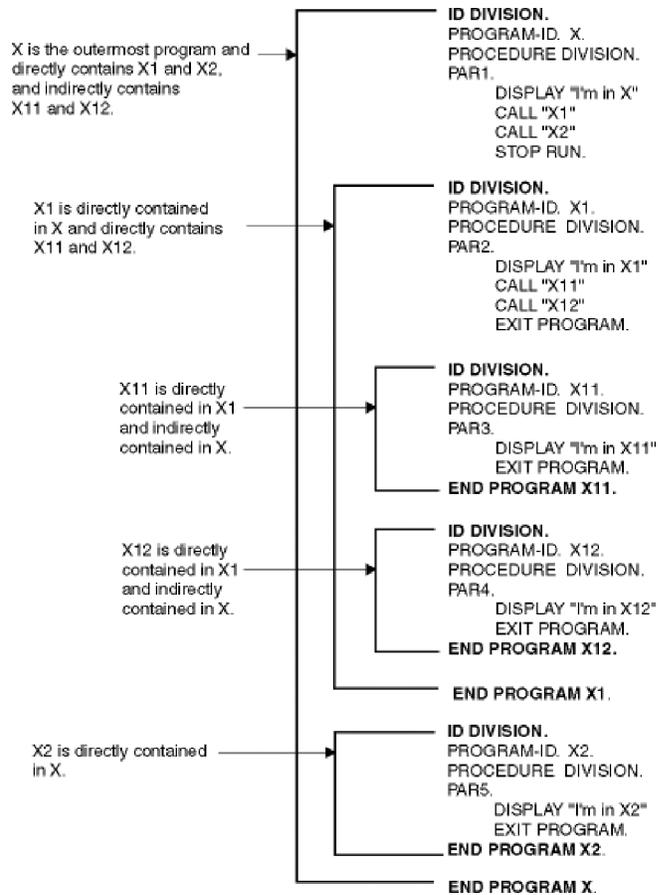


Figure 51. Nested Program Structure with Directly and Indirectly Contained Programs

Conventions for Using Nested Program Structure

There are several conventions that apply when using nested program structures.

1. The Identification Division is required in each program. All other divisions are optional.
2. Program name in the PROGRAM-ID paragraph must be unique.
3. Names of nested programs can be any valid COBOL word or a nonnumeric literal.
4. Nested programs can not have a Configuration Section. The outermost program must specify any Configuration Section options that may be required.
5. Each nested program is included in the containing program immediately before its END PROGRAM header (see Figure 51).
6. Each ILE COBOL program must be terminated by an END PROGRAM header.
7. Nested programs can only be called or canceled from an ILE COBOL program in the same module object.
8. Calls to nested programs can only be made using either a CALL *literal* or CALL *identifier* statement. Calls to nested programs cannot be made using CALL *procedure-pointer*. Calls to nested programs follow the same rules as static procedure calls.

Calling Hierarchy for Nested Programs

A nested program may only be called by its directly containing program, unless the nested program is identified as COMMON in its PROGRAM-ID paragraph. In that case, the COMMON program may also be called by any program that is contained (directly or indirectly) within the same program as the one directly containing the COMMON program. Recursive calls are only allowed for nested programs that have the RECURSIVE clause, or when the nested program's direct or indirect containing program has the RECURSIVE clause.

Figure 52 shows the outline of a nested structure with some contained programs identified as COMMON.

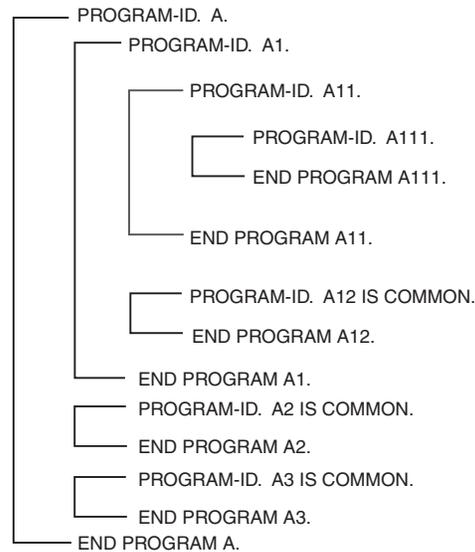


Figure 52. Nested Program Structure with Directly and Indirectly Contained Programs

The following table describes the **calling hierarchy** for the structure that is shown in Figure 52. Notice that A12, A2, and A3 are identified as COMMON and the resulting differences in calls associated with them.

Table 15. Calling Hierarchy for Nested Structures with COMMON Programs

This Program	Can call these programs	And can be called by these programs
A	A1, A2, A3	None
A1	A11, A12, A2, A3	A
A11	A111, A12, A2, A3	A1
A111	A12, A2, A3	A11
A12	A2, A3	A1, A11, A111
A2	A3	A, A1, A11, A111, A12, A3
A3	A2	A, A1, A11, A111, A12, A2

You should note that:

- A2 cannot call A1 because A1 is not COMMON and is not directly contained in A2
- A111 cannot call A11 because that would be a recursive call, unless A11, or A1, or A has a RECURSIVE clause in its PROGRAM-ID paragraph.
- A1 can call A2 because A2 is COMMON

- A1 can call A3 because A3 is COMMON.

Scope of Names within a Nested Structure

There are two classes of names within nested structures—**local** and **global**. The class will determine whether a name is known beyond the scope of the program which declares it.

Local Names: Names are local unless declared to be GLOBAL (except the program name). These local names are not visible or accessible to any program outside of the one where they were declared; this includes both contained and containing programs.

Global Names: A name that is specified as global (by using the GLOBAL clause) is visible and accessible to the program in which it is declared, and to all the programs that are directly and indirectly contained within the program. This allows the contained programs to share common data and files from the containing program, simply by referencing the name of the item.

Any item that is subordinate to the global item (including condition names and indexes) is automatically global.

The same name may be declared with the GLOBAL clause multiple times, providing that each declaration occurs in a different program. Be aware that masking, or hiding, a name within a nested structure is possible by having the same name occur within different programs of the same containing structure.

Searching for Name Declarations: When a name is referenced within a program, a search is made to locate the declaration for that name. The search begins within the program that contains the reference and continues **outward** to containing programs until a match is found. The search follows this process:

1. Declarations within the program are searched first.
2. If no match is found, then **only** global declarations are searched in successive outer containing programs.
3. The search ends when the first matching name is found, otherwise an error exists if no match is found.

Using Static Procedure Calls and Dynamic Program Calls

The following discussion applies to separately compiled subprograms only, not to nested programs. For information about calls within a nested program structure, see “Calling Nested Programs” on page 217.

The binding process differs, depending on whether your ILE COBOL program uses static procedure calls or dynamic program calls. When a static procedure call is used to call an ILE COBOL subprogram, it must first be compiled into a module object and then bound, by copy or by reference, into the same program object as the calling ILE COBOL program. When a dynamic program call is used to call an ILE COBOL subprogram, the ILE COBOL subprogram must be compiled and bound as a separate program object. For more information on the binding process, see the *ILE Concepts* book.

Static procedure calls offer performance advantages over dynamic program calls.

When an ILE COBOL subprogram is called using a static procedure call, it is already activated, since it is bound in the same program object as the calling program, and it is performed immediately upon receiving control from the calling ILE COBOL program.

When an ILE COBOL subprogram is called using a dynamic program call, many other tasks may need to be performed before the called ILE COBOL program is actually performed. These tasks include the following:

- If the activation group in which the called ILE COBOL program is to be activated does not exist, it must first be created before the called ILE COBOL program can be activated in it.
- If the called ILE COBOL program has not been previously activated, it must first be activated before it can be performed. Activating the called ILE COBOL program also implies activating all service programs bound (directly or indirectly) to it. Activation involves performing the following functions:
 - Uniquely allocating the static data needed by the program object or service program
 - changing the symbolic links to used service programs into links to physical addresses.

Thus, a dynamic program call is slower than a static procedure call due to the cost of activation the first time it is performed in an activation group.

Dynamic program calls and static procedure calls also differ in the number of operands that can be passed from the calling ILE COBOL program to the called ILE COBOL program. You can pass up to 255 operands using a dynamic program call. With a static procedure call, you can pass up to 400 operands.

Arguments that are designated as OMITTED or as having associated operational descriptors can only be passed using a static procedure call. These arguments cannot be passed using dynamic program calls.

Performing Static Procedure Calls using CALL literal

You can perform a static procedure call by using the CALL *literal* statement (where *literal* is the name of a subprogram). There are three ways to specify that the call is to be a static procedure call. They are listed in order of precedence:

Note: The IN LIBRARY phrase is incompatible with a static procedure call.

1. Use the LINKAGE phrase of the CALL statement.
 - You specify LINKAGE TYPE IS PROCEDURE in the CALL statement to ensure that the called program will be invoked using a static procedure call.

```
PROCEDURE DIVISION.  
  
:  
CALL LINKAGE TYPE IS PROCEDURE literal-1
```

2. Use the LINKAGE TYPE clause of the SPECIAL-NAMES paragraph.
 - You specify LINKAGE TYPE IS PROCEDURE FOR *literal-1* in the SPECIAL-NAMES paragraph where *literal-1* is the name of the ILE COBOL program you are calling. You do not need to specify the LINKAGE TYPE phrase with the CALL statement when the linkage has been specified in the SPECIAL-NAMES paragraph.

```

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.

:
SPECIAL-NAMES.
LINKAGE TYPE IS PROCEDURE FOR literal-1.

:
PROCEDURE DIVISION.

:
CALL literal-1.

```

3. Use the LINKLIT parameter of the CRTCBMOD and CRTBNDCBL commands, or the associated PROCESS statement option.
 - You specify *PRC with the LINKLIT parameter of the CRTCBMOD and CRTBNDCBL commands, at compile time, to indicate that static procedure calls are to take place for all external CALL *literal-1* statements in the ILE COBOL program. You do not need to specify the LINKAGE TYPE clause in the SPECIAL-NAMES paragraph or the LINKAGE TYPE phrase with the CALL or CANCEL statement when the linkage has been defined by the LINKLIT parameter of CRTCBMOD.

```

CRTCBMOD MODULE(MYLIB/XMPLE1)
SRCFILE(MYLIB/QCBLLESRC) SRCMBR(XMPLE1)
LINKLIT(*PRC)

```

- You code the CALL statements as follows when using the LINKLIT parameter of CRTCBMOD to specify linkage type:

```
PROCEDURE DIVISION.
```

```

:
CALL literal-1.

```

Performing Dynamic Program Calls using CALL literal

You can perform a dynamic program call by using the CALL *literal* statement (where *literal* is the name of a subprogram) or the CALL *identifier* statement. Refer to “Using CALL identifier” on page 223 for more information about CALL *identifier*. There are three ways, using CALL *literal*, to specify that the call is to be a dynamic program call. They are listed in order of precedence:

1. Use the LINKAGE phrase of the CALL statement.
 - You specify LINKAGE TYPE IS PROGRAM in the CALL statement to ensure that the called program will be invoked using a dynamic program call.

```

PROCEDURE DIVISION.

:
CALL LINKAGE TYPE IS PROGRAM literal-1

```

2. Use the LINKAGE TYPE clause of the SPECIAL-NAMES paragraph.
 - You specify LINKAGE TYPE IS PROGRAM FOR *literal-1* in the SPECIAL-NAMES paragraph where *literal-1* is the name of the ILE COBOL program you are calling. You do not need to specify the LINKAGE TYPE phrase with the CALL statement when the linkage has been specified in the SPECIAL-NAMES paragraph.

```

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.

:
SPECIAL-NAMES.
LINKAGE TYPE IS PROGRAM FOR literal-1.

```

```
⋮  
PROCEDURE DIVISION.
```

```
⋮  
CALL literal-1.
```

3. Use the LINKLIT parameter of the CRTCBMOD and CRTBNDCBL commands, or the associated PROCESS statement option.
 - You specify *PGM with the LINKLIT parameter of the CRTCBMOD and CRTBNDCBL commands, at compile time, to indicate that dynamic program calls are to take place for all external CALL *literal-1* statements in the ILE COBOL program. You do not need to specify the LINKAGE TYPE clause in the SPECIAL-NAMES paragraph or the LINKAGE TYPE phrase with the CALL or CANCEL statement when the linkage has been defined by the LINKLIT parameter of CRTCBMOD.

```
CRTCBMOD MODULE(MYLIB/XMPLE1)  
SRCFILE(MYLIB/QCBLLSRC) SRCMBR(XMPLE1)  
LINKLIT(*PGM)
```

- You code the CALL statements as follows when using the LINKLIT parameter of CRTCBMOD to specify linkage type:

```
PROCEDURE DIVISION.
```

```
⋮  
CALL literal-1.
```

A dynamic program call activates the subprogram at run time. Use a dynamic call statement when:

- You want to simplify maintenance tasks and take advantage of code re-usability. When a subprogram is changed, all module objects, except for service programs, that call it statically and are bound by copy must be re-bound. If they are bound by reference, they do not need to be re-bound provided that the interface between the subprogram and the module objects is unchanged. If the changed subprogram is called dynamically, then only the changed subprogram needs to be re-bound. Thus, dynamic calls make it easier to maintain one copy of a subprogram with a minimum amount of binding.

- The subprograms called with the CALL *literal* are used infrequently or are very large.

If the subprograms are called only on a few conditions, dynamic calls can activate the subprograms only when needed.

If the subprograms are very large or there are many of them, use of static calls might require a larger working set size in main storage.

Using CALL identifier

You can use CALL *identifier* (where *identifier* is not a procedure-pointer) to call a nested ILE COBOL program or to call a program object. The contents of the identifier determine, at run time, whether a nested program is called or a program object is called. If the contents of the identifier match the name of a visible nested program, then the call is directed to the nested program. Otherwise, a dynamic program call is made to a program object with the name specified in the contents of the identifier.

An IN LIBRARY phrase specified on a CALL identifier forces the call to be to a program object.

An open pointer that associates a CALL identifier (and any associated IN LIBRARY item) with an object is set the first time you use the identifier in a CALL statement.

If you carry out a call by an identifier to a program object that you subsequently delete or rename, you must use the CANCEL statement to null the open pointer associated with the identifier. This ensures that when you next use the identifier to call your program object, the associated open pointer will be set again.

The following example shows how to apply the CANCEL statement to an identifier:

```
MOVE "ABCD" TO IDENT-1.  
CALL IDENT-1.  
CANCEL IDENT-1.
```

If you apply the CANCEL statement directly to the literal "ABCD", you do *not* null the open pointer associated with IDENT-1. Instead, you can continue to call program ABCD simply by using IDENT-1 in your CALL statement.

The value of the open pointer also changes if you change the value of the CALL identifier and perform a call using this new value. The value of the open pointer is also affected by any associated IN LIBRARY item. If a different library is specified for a CALL to IDENT-1 than on a previous call to IDENT-1, the open pointer is reset.

Using CALL procedure-pointer

You can perform a static procedure call or a dynamic program call using the CALL *procedure-pointer* statement.

Before using the CALL *procedure-pointer* statement, you must set the *procedure-pointer* data item to an address value. The *procedure-pointer* data item can be set to the outermost COBOL program (an ILE procedure), an ILE procedure in another compilation unit, or a program object. You use the Format 6 SET Statement to set the value of the *procedure-pointer* data item.

You specify LINKAGE TYPE IS PROCEDURE in the SET statement to set the *procedure-pointer* data item to an ILE procedure.

You specify LINKAGE TYPE IS PROGRAM in the SET statement to set the *procedure-pointer* data item to a program object.

You can also use the LINKAGE TYPE clause of the SPECIAL-NAMES paragraph or the LINKLIT parameter of the CRTCBMOD and CRTBNDCBL commands to determine the type of the object to which the *procedure-pointer* data item is set. Refer to "Identifying the Linkage Type of Called Programs and Procedures" on page 215 for more information on setting the linkage type using the LINKAGE TYPE clause of the SPECIAL-NAMES paragraph or the LINKLIT parameter of the CRTCBMOD and CRTBNDCBL commands.

You code the SET statement and CALL statement as follows when using CALL *procedure-pointer* to perform a static procedure call:

```
PROCEDURE DIVISION.  
  
:  
:  
:  
SET procedure-pointer  
TO ENTRY LINKAGE TYPE IS PROCEDURE literal-1.
```

```

:
CALL procedure-pointer.

```

You code the SET statement and CALL statement as follows when using CALL *procedure-pointer* to perform a dynamic program call:

```

PROCEDURE DIVISION.

:
SET procedure-pointer
  TO ENTRY LINKAGE TYPE IS PROGRAM literal-1.

:
CALL procedure-pointer.

```

Using Recursive Calls

Code the RECURSIVE clause on the PROGRAM-ID clause so your program can be recursively reentered while a previous invocation is still active. Below is an example of how you could use the RECURSIVE clause to make a program a recursive program, and how a Local-Storage Section data item can be used in a recursive program.

```

5722WDS V5R4M0 060210 LN  IBM ILE COBOL           MYLIB/FACTORIAL           ISERIES   06/02/15 17:25:51           Page     2
                               S o u r c e
STMT PL SEQNBR -A 1 B.+...2....3....4....5....6....7..IDENTFCN  S COPYNAME  CHG DATE
  1   000100 IDENTIFICATION DIVISION.
  2   000200 PROGRAM-ID. FACTORIAL RECURSIVE.
      000300
  3   000400 ENVIRONMENT DIVISION.
  4   000500 CONFIGURATION SECTION.
  5   000600 SOURCE-COMPUTER. IBM-ISERIES.
  6   000700 OBJECT-COMPUTER. IBM-ISERIES.
      000800
  7   000900 DATA DIVISION.
  8   001000 WORKING-STORAGE SECTION.
  9   001100 01 NUMB PIC 9(4) VALUE 5.
 10   001200 01 FACT PIC 9(8) VALUE 0.
      001300
 11   001400 LOCAL-STORAGE SECTION.
 12   001500 01 NUM PIC 9(4).
      001600
 13   001700 PROCEDURE DIVISION.
 14   001800     MOVE NUMB TO NUM.
 15   001900     IF NUMB = 0
 16   002000         MOVE 1 TO FACT
      002100     ELSE
 17   002200         SUBTRACT 1 FROM NUMB
 18   002300         CALL "FACTORIAL"
 19   002400         MULTIPLY NUM BY FACT
      002500     END-IF.
 20   002600     DISPLAY NUM "!" = " FACT.
 21   002700     GOBACK.
 22   002800 END PROGRAM FACTORIAL.
                               * * * * * E N D   O F   S O U R C E   * * * * *

```

Figure 53. Example of a recursive call to calculate the factorial of a number

Returning from an ILE COBOL Program

You can issue a STOP RUN, EXIT PROGRAM, or GOBACK statement to return control from a called ILE COBOL program.

You must know if an ILE COBOL program is a main program or a subprogram to determine how control is returned from a called program when an error occurs, or a program ends. See “Main Programs and Subprograms” on page 213 for a description of main programs and subprograms.

Returning from a Main Program

To return control from a main program, you use either `STOP RUN`, `GOBACK`, or `EXIT PROGRAM` with the `CONTINUE` phrase. The `STOP RUN` and `GOBACK` statements end the run unit, and control is returned to the caller of the main program. `EXIT PROGRAM` without the `CONTINUE` phrase cannot be used to return control from a main program. When `EXIT PROGRAM` without the `CONTINUE` phrase is encountered in a main program, no operation is performed and processing continues at the next statement in the main program.

Returning from a *NEW Activation Group

When the `STOP RUN`, `GOBACK`, or an `EXIT PROGRAM` with the `CONTINUE` phrase are performed from a called main ILE COBOL program in a `*NEW` activation group, the activation group is ended when control is returned to the calling program. The activation group will close all files and return all resources] back to the system.

As a result of the activation group ending, the called ILE COBOL program is placed in its initial state.

Returning from a Named Activation Group

When an `EXIT PROGRAM` with the `CONTINUE` phrase is performed from a called main ILE COBOL program in a named activation group, the activation group remains active and control is returned to the calling program. All files and resources in the activation group are left in their last used state.

When the `STOP RUN` or `GOBACK` statements are performed from a called main ILE COBOL program in a named activation group, the activation group is ended when control is returned to the calling program. The activation group will close all files and return all resources back to the system.

Returning from the Default (*DFACTGRP) Activation Group

When the `STOP RUN` or `GOBACK` statements are performed from a called main ILE COBOL program in the default (`*DFACTGRP`) activation group, the activation group remains active and control is returned to the calling program. All files and resources used in the activation group are left in their last used state.

Returning from a Subprogram

To return control from a subprogram, the subprogram may end with an `EXIT PROGRAM`, a `GOBACK`, or a `STOP RUN` statement. If the subprogram ends with an `EXIT PROGRAM` or a `GOBACK` statement, control returns to its immediate caller without ending the run unit. An implicit `EXIT PROGRAM` statement is generated if there is no next executable statement in a called program. If the subprogram ends with a `STOP RUN` statement, all programs in the run unit up to the nearest control boundary are ended, and control returns to the program prior to the control boundary.

A subprogram is usually left in its **last-used state** when it ends with `EXIT PROGRAM` or `GOBACK`. The next time it is called in the run unit, its internal values will be as they were left, except that all `PERFORM` statements are considered to be complete and will be reset to their initial values. In contrast, a main program is initialized each time it is called. There are two exceptions:

- A subprogram that is dynamically called and then canceled will be in the initial state the next time it is called.
- A program, which has the `INITIAL` clause specified in its `PROGRAM-ID` paragraph, will be in the initial state each time it is called.

Maintaining OPM COBOL/400 Run Unit Defined STOP RUN Semantics

To have the STOP RUN statement behave in a manner which is compatible with an OPM COBOL/400 run unit, your ILE COBOL application must be created using specific conditions. Refer to “COBOL Run Unit” on page 212 for a description of these conditions.

Examples of Returning from an ILE COBOL Program

The following examples illustrate the behavior of EXIT PROGRAM, STOP RUN, and GOBACK in various combinations of Named, *NEW, and *DFTACTGP activation groups.

Figure 54 shows an activation group containing programs A, B, C, D, and E. A calls B and C; C calls D and E.

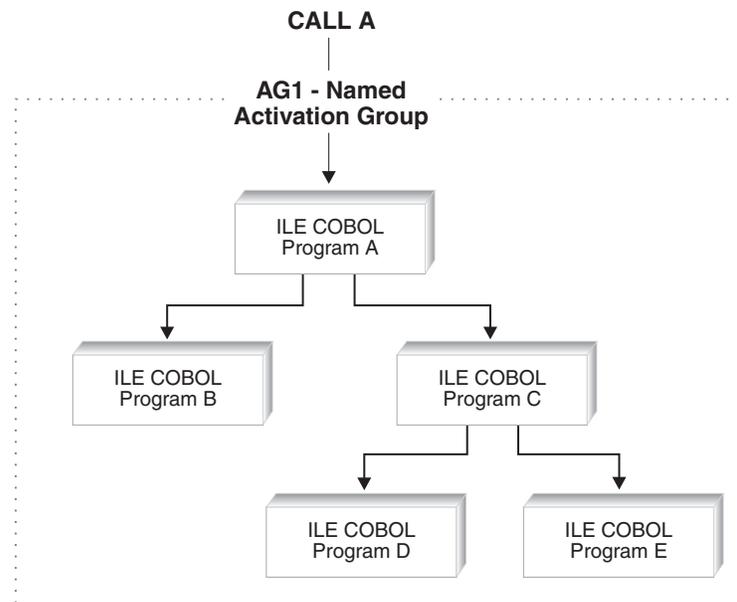


Figure 54. Example of EXIT PROGRAM, STOP RUN, and GOBACK behavior in a Single Named Activation Group

Statement	Program A	Program B	Program C	Program D	Program E
EXIT PROGRAM	1	4	4	2	2
STOP RUN	3	3	3	3	3
GOBACK	3	4	4	2	2

- 1 No operation is processed if an EXIT PROGRAM without the CONTINUE phrase is coded because the statement is in a main program. Processing continues with the next statement in the program. An EXIT PROGRAM with the CONTINUE phrase returns control to the caller of Program A, and leaves the activation group active. All files and resources used in the activation group are left in their last used state.
- 2 The activation group remains active and control is returned to Program C. All files and resources used in the activation group are left in their last used state.

- 3** The activation group is ended and control is returned to the caller of the main program. The activation group will close all files scoped to the activation group. Any pending commit operations scoped to the activation group will be implicitly committed. All resources allocated to the activation group will be returned back to the system. As a result of the activation group ending, all programs that were active in the activation group are placed in their initial state.
- 4** The activation group remains active and control is returned to Program A. All files and resources used in the activation group are left in their last used state.

Figure 55 shows two activation groups. Activation group 1 contains programs A and B. Activation group 2 contains programs C, D, and E. A calls B and C. C calls D and E.

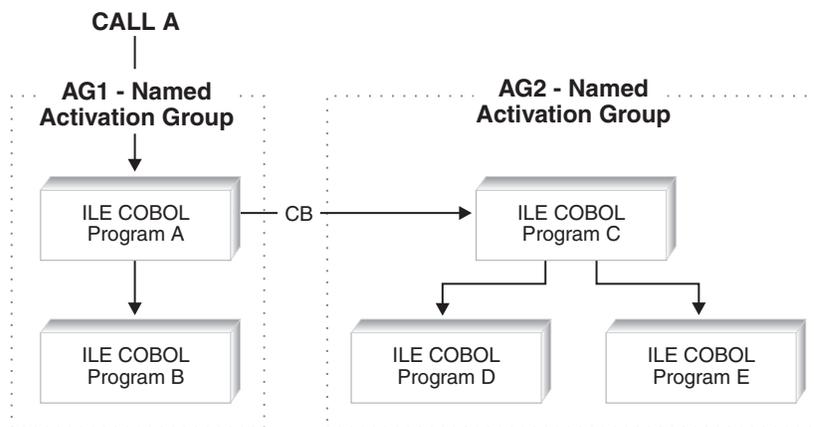


Figure 55. Example of EXIT PROGRAM, STOP RUN, and GOBACK behavior in Two Named Activation Groups

Statement	Program A	Program B	Program C	Program D	Program E
EXIT PROGRAM	1	5	1	2	2
STOP RUN	3	3	4	4	4
GOBACK	3	5	4	2	2

- 1** If an EXIT PROGRAM statement without the CONTINUE phrase was used, no operation is processed because the statement is in a main program. Processing continues with the next statement in the program. If an EXIT PROGRAM statement with the CONTINUE phrase was used, the activation group remains active and control is returned to the calling program or command. All files and resources used in the activation group are left in their last used state.
- 2** The activation group remains active and control is returned to Program C. All files and resources used in the activation group are left in their last used state.
- 3** The activation group is ended and control is returned to the caller of the main program. The activation group will close all files scoped to the activation group. Any pending commit operations scoped to the activation group will be implicitly committed. All resources allocated to the activation

group will be returned back to the system. As a result of the activation group ending, all programs that were active in the activation group are placed in their initial state.

- 4** The activation group is ended and control is returned to Program A. The activation group will close all files scoped to the activation group. Any pending commit operations scoped to the activation group will be implicitly committed. All resources allocated to the activation group will be returned back to the system. As a result of the activation group ending, all programs that were active in the activation group are placed in their initial state.
- 5** The activation group remains active and control is returned to Program A. All files and resources used in the activation group are left in their last used state.

Figure 56 shows two named activation groups and a *NEW activation group. Activation group 1 contains programs A and D. Activation group 2 contains programs C and E. *NEW activation group contains program B. A calls B and C. C calls D and E.

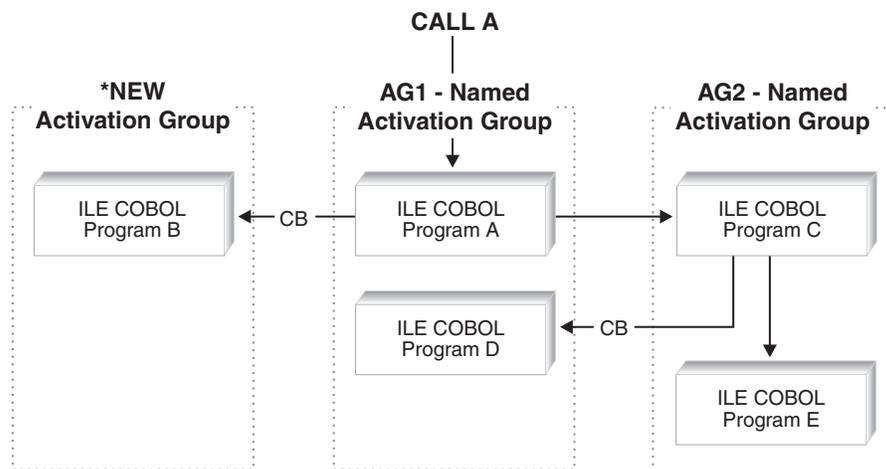


Figure 56. Example of EXIT PROGRAM, STOP RUN, and GOBACK behavior in multiple *NEW and Named Activation Groups

Statement	Program A	Program B	Program C	Program D	Program E
EXIT PROGRAM	1	5	1	2	2
STOP RUN	3	4	4	2	4
GOBACK	3	4	4	2	2

- 1** If an EXIT PROGRAM statement without the CONTINUE phrase was used, no operation is processed because the statement is in a main program. Processing continues with the next statement in the program. If an EXIT PROGRAM statement with the CONTINUE phrase was used, the activation group remains active and control is returned to the calling program or command. All files and resources used in the activation group are left in their last used state.

- 2** The activation group remains active and control is returned to Program C. All files and resources used in the activation group are left in their last used state.
- 3** The activation group is ended and control is returned to the caller of the main program. The activation group will close all files scoped to the activation group. Any pending commit operations scoped to the activation group will be implicitly committed. All resources allocated to the activation group will be returned back to the system. As a result of the activation group ending, all programs that were active in the activation group are placed in their initial state.
- 4** The activation group is ended and control is returned to Program A. The activation group will close all files scoped to the activation group. Any pending commit operations scoped to the activation group will be implicitly committed. All resources allocated to the activation group will be returned back to the system. As a result of the activation group ending, all programs that were active in the activation group are placed in their initial state.
- 5** If an EXIT PROGRAM statement without the CONTINUE phrase was used, no operation is processed because the statement is in a main program. Processing continues with the next statement in the program.
If an EXIT PROGRAM statement with the CONTINUE phrase was used, control is returned to the calling program or command. In a *NEW activation group, when a main program returns control to the caller, the activation group is ended. The activation group will close all files scoped to the activation group. Any pending commit operation scoped to the activation group will be implicitly committed.
All resources allocated to the activation group will be returned back to the system. As a result of the activation group ending, all programs that were active in the activation group are placed in their initial state.

Figure 57 on page 231 shows interaction between a named activation group, the default activation group and a *NEW activation group. Activation group 1 contains programs A and D. *DFTACTGP activation group contains OPM COBOL/400 programs C and E. *NEW activation group contains program B. A calls B and C. C calls D and E.

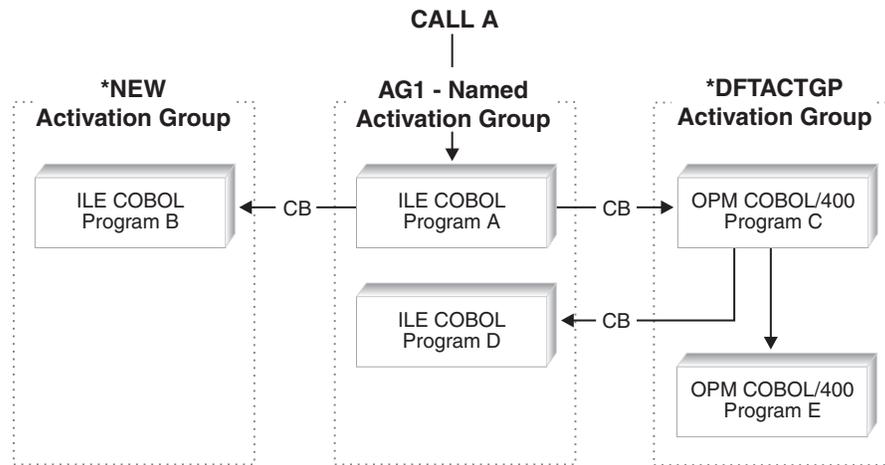


Figure 57. Example of EXIT PROGRAM, STOP RUN, and GOBACK behavior in *NEW, Named, and *DFTACTGP Activation Groups

Statement	Program A	Program B	Program C	Program D	Program E
EXIT PROGRAM	1	6	7	2	2
STOP RUN	3	4	5	2	5
GOBACK	3	4	5	2	2

- 1** If an EXIT PROGRAM statement without the CONTINUE phrase was used, no operation is processed because the statement is in a main program. Processing continues with the next statement in the program. If an EXIT PROGRAM statement with the CONTINUE phrase was used, the activation group remains active and control is returned to the calling program or command. All files and resources used in the activation group are left in their last used state.
- 2** The activation group remains active and control is returned to Program C. All files and resources used in the activation group are left in their last used state.
- 3** The activation group is ended and control is returned to the caller of the main program. The activation group will close all files scoped to the activation group. Any pending commit operations scoped to the activation group will be implicitly committed. All resources allocated to the activation group will be returned back to the system. As a result of the activation group ending, all programs that were active in the activation group are placed in their initial state.
- 4** The activation group is ended and control is returned to Program A. The activation group will close all files scoped to the activation group. Any pending commit operations scoped to the activation group will be implicitly committed. All resources allocated to the activation group will be returned back to the system. As a result of the activation group ending, all programs that were active in the activation group are placed in their initial state.
- 5** The activation group remains active and control is returned to Program A. All files that were opened by Program C or Program E are closed. Any

pending commit operations for files opened by Program C or Program E will be implicitly committed. Storage is freed for Program C and Program E.

6 If an EXIT PROGRAM statement without the CONTINUE phrase was used, no operation is processed because the statement is in a main program. Processing continues with the next statement in the program.

If an EXIT PROGRAM statement with the CONTINUE phrase was used, control is returned to the calling program or command.

In a *NEW activation group, when a main program returns control to the caller, the activation group is ended. The activation group will close all files scoped to the activation group. Any pending commit operation scoped to the activation group will be implicitly committed.

All resources allocated to the activation group will be returned back to the system. As a result of the activation group ending, all programs that were active in the activation group are placed in their initial state.

7 No operation is processed because the statement is in a main program. Processing continues with the next statement in the program.

Passing Return Code Information (RETURN-CODE Special Register)

You can use the RETURN-CODE special register to pass and receive return codes between ILE COBOL programs. You can set the RETURN-CODE special register before returning from a called ILE COBOL program.

When used in nested programs, the RETURN-CODE special register is implicitly defined as GLOBAL in the outermost ILE COBOL program. Any changes made to the RETURN-CODE special register is global to all ILE COBOL programs within the module object.

When an ILE COBOL program returns to its caller, the contents of its RETURN-CODE special register are transferred into the RETURN-CODE special register of the calling program.

When control is returned from a main ILE COBOL program to the operating system, the RETURN-CODE special register contents are returned as a user return code.

Passing and Sharing Data Between Programs

There are many ways to pass or share data between ILE COBOL programs:

- Data can be declared as GLOBAL so that it can be used by nested programs.
- Data can be returned to a calling program using the RETURNING phrase of the CALL statement.
- Data can be passed to a called program BY REFERENCE, BY VALUE, or BY CONTENT when the CALL statement is run.
- Data that is declared as EXTERNAL can be shared by separately compiled programs. EXTERNAL data can also be shared between nested ILE COBOL programs within a module object.
- Files that are declared as EXTERNAL can be shared by separately compiled programs. EXTERNAL files can also be shared between nested ILE COBOL programs within a module object.

- Pointers can be used when you want to pass and receive addresses of dynamically-located data items.
- Data can be passed using Data Areas.

Comparing Local and Global Data

The concept of local and global data applies only to nested programs.

Local data is accessible only from within the program in which the local data is declared. Local data is not visible or accessible to any program outside of the one where it is declared; this includes both contained and containing programs.

All data is considered to be local data unless it is explicitly declared as being global data.

Global data is accessible from within the program in which the global data is declared or from within any other nested programs which are directly or indirectly contained in the program that declared the global data.

Data-names, file-names and record-names can be declared as global.

To declare a data-name as global, specify the GLOBAL clause either in the data description entry by which the data-name is declared or in another entry to which that data description entry is subordinate.

To declare a file-name as global, specify the GLOBAL clause in the file description entry for that file-name.

To declare a record-name as global, specify the GLOBAL clause in the record description entry by which the record-name is declared or, in the case of record description entries in the File Section, specify the GLOBAL clause in the file description entry for the file-name associated with the record description entry.

For a detailed description of the GLOBAL clause, refer to the *IBM Rational Development Studio for i: ILE COBOL Reference*.

Passing Data Using CALL...BY REFERENCE, BY VALUE, or BY CONTENT

BY REFERENCE means that any changes made by the subprogram to the variables it received are visible by the calling program.

BY CONTENT means that the calling program is passing only the **contents** of the *literal* or *identifier*. With a CALL...BY CONTENT, the called program cannot change the value of the *literal* or *identifier* in the calling program, even if it modifies the parameters it received.

BY VALUE means that the calling program is passing the value of the *literal*, or *identifier*, not a reference to the sending item. The called program can change the parameter in the called program. However, because the subprogram has access only to a temporary copy of the sending item, those changes don't affect the argument in the calling program.

Whether you pass data items BY REFERENCE, BY VALUE, or BY CONTENT depends on what you want your program to do with the data:

- If you want the definition of the argument of the CALL statement in the calling program and the definition of the parameter in the called program to share the same memory, specify:
CALL...BY REFERENCE identifier
Any changes made by the subprogram to the parameter affect the argument in the calling program.
- If you want to pass the address of a record area to a called program, specify:
CALL...BY REFERENCE ADDRESS OF record-name
The subprogram receives the ADDRESS OF special register for the record-name you specify.
You must define the record name as a level-01 or level-77 item in the Linkage Section of the called and calling programs. A separate ADDRESS OF special register is provided for each record in the Linkage Section.
- If you want to pass the address of any data item in the DATA DIVISION to a called program, specify:
CALL...BY CONTENT ADDRESS OF data-item-name
- If you do not want the definition of the argument of the CALL statement in the calling program and the definition of the parameter in the called subprogram to share the same memory, specify:
CALL...BY CONTENT identifier
- If you want to pass data to ILE programs that require BY VALUE parameters use:
CALL...BY VALUE item
- If you want to pass a numeric integer of various lengths specify:
CALL...BY VALUE integer-1 SIZE integer-2
The numeric integer is passed as a binary value of length integer-2. The SIZE phrase is optional. If not specified, integer-1 is passed as a 4 byte binary number.
- If you want to call an ILE C, C++ or RPG function with a function return value, use:
CALL...RETURNING identifier
- If you want to pass a literal value to a called program, specify:
CALL...BY CONTENT literal

The called program cannot change the value of the literal.
- If you want to pass the length of a data item, specify:
CALL...BY CONTENT LENGTH OF identifier
The calling program passes the length of *identifier* from its LENGTH OF special register.
- If you want to pass both a data item and its length to a subprogram, specify a combination of BY REFERENCE and BY CONTENT. For example:
CALL 'ERRPROC' USING BY REFERENCE A
BY CONTENT LENGTH OF A.
- If you do not want the called program to receive a corresponding argument or if you want the called program to use the default value for the argument, specify the OMITTED phrase in place of each data item to be omitted on the CALL...BY REFERENCE or CALL...BY CONTENT statement. For example:
CALL...BY REFERENCE OMITTED
CALL...BY CONTENT OMITTED OMITTED

In the called program, you can use the CEETSTA API to determine if a specified parameter is OMITTED or not.

- If you want to pass data items with **operational descriptors**, specify the LINKAGE TYPE IS PRC...USING ALL DESCRIBED clause in the SPECIAL-NAMES paragraph. Then use the CALL...BY REFERENCE, CALL...BY CONTENT or CALL...BY VALUE statement to pass the data. Operational descriptors provide descriptive information to the called ILE procedure in cases where the called ILE procedure cannot precisely anticipate the form of the data items being passed. You use operational descriptors when they are expected by a called ILE procedure written in a different ILE language and when they are expected by an ILE bindable API. Refer to the *ILE Concepts* book for more information about operational descriptors. For example:

```
SPECIAL-NAMES. LINKAGE TYPE PRC FOR 'ERRPROC'
                USING ALL DESCRIBED.
```

```
⋮
CALL 'ERRPROC' USING BY REFERENCE identifier.
```

or

```
SPECIAL-NAMES. LINKAGE TYPE PRC FOR 'ERRPROC'
                USING ALL DESCRIBED.
```

```
⋮
CALL 'ERRPROC' USING BY CONTENT identifier.
```

Data items in a calling program can be described in the Linkage Section of all the programs it calls directly or indirectly. In this case, storage for these items is allocated in the outermost calling program.

Describing Arguments in the Calling Program

The data that is passed from a calling program is called an **argument**. In the calling program, the arguments are described in the Data Division in the same manner as other data items in the Data Division. Unless they are in the Linkage Section, storage is allocated for these items in the calling program. If you reference data in a file, the file must be open when the data is referenced. Code the USING clause of the CALL statement to pass the arguments.

Describing Parameters in the Called Program

The data that is received in a called program is called a **parameter**. In the called program, parameters are described in the Linkage Section. Code the USING clause after the PROCEDURE-DIVISION header to receive the parameters.

Writing the Linkage Section in the Called Program: You must know what is being passed from the calling program and set up the Linkage Section in the called program to accept it. To the called program, it doesn't matter which clause of the CALL statement you use to pass the data (BY REFERENCE, BY VALUE or BY CONTENT). In all cases, the called program must describe the data it is receiving. It does this in the Linkage Section.

The number of *data-names* in the *identifier* list of a called program should not be greater than the number of *data-names* in the *identifier* list of the calling program. There is a one-to-one positional correspondence; that is, the first *identifier* of the calling program is passed to the first *identifier* of the called program, and so forth. The ILE COBOL compiler does not enforce consistency in terms of number of arguments and number of parameters nor does it enforce consistency in terms of type and size between an argument and its corresponding parameter.

Any inconsistencies in terms of number of arguments and number of parameters may result in runtime exceptions. For a dynamic program call, when the number

of arguments is greater than the number of parameters, a runtime exception is generated in the calling program when the CALL statement is attempted. This exception can be captured if the ON EXCEPTION phrase is specified on the CALL statement.

When the number of arguments is less than the number of parameters, a runtime exception is not generated in the calling program when the CALL statement is performed. Instead, a pointer exception is generated in the called program when it tries to access an unsupplied parameter.

If an argument was passed BY VALUE, the PROCEDURE DIVISION header of the subprogram must indicate that:

```
PROCEDURE DIVISION USING BY VALUE DATA-ITEM.
```

If an argument was passed BY REFERENCE or BY CONTENT, the PROCEDURE DIVISION header does not need to indicate how the argument was passed. The header can either be:

```
PROCEDURE DIVISION USING DATA-ITEM
```

or:

```
PROCEDURE DIVISION USING BY REFERENCE DATA-ITEM
```

Grouping Data to be Passed

Consider grouping all the data items you want to pass between programs and putting them under one level-01 item. If you do this, you can pass a single level-01 record between programs. For an example of this method, see Figure 58 on page 237.

To make the possibility of mismatched records even smaller, put the level-01 record in a copy member, and copy it in both programs. (That is, copy it in the Working-Storage Section of the calling program and in the Linkage Section of the called program.)

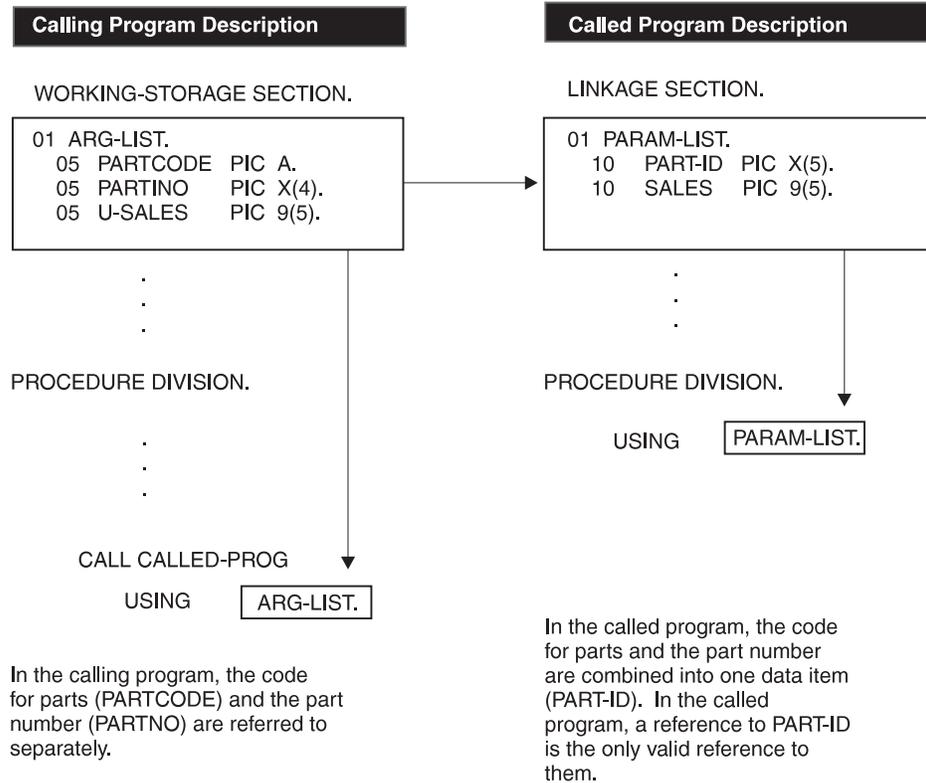


Figure 58. Common Data Items in Subprogram Linkage

Sharing EXTERNAL Data

Separately compiled ILE COBOL programs (including programs within a sequence of ILE COBOL source programs) can share data items by using the EXTERNAL clause. This EXTERNAL data is handled as weak exports. Refer to *ILE Concepts* for further information about strong and weak exports.

You specify the EXTERNAL clause on the 01-level data description in the Working-Storage Section of the ILE COBOL program, and the following rules apply:

1. Items subordinate to an EXTERNAL group item are themselves EXTERNAL.
2. The name used for the data item cannot be used on another EXTERNAL item within the same program.
3. The VALUE clause cannot be specified for any group item, or subordinate item, that is EXTERNAL.
4. EXTERNAL data cannot be initialized and its initial value at runtime is undefined. If your application requires that EXTERNAL data items be initialized, it is recommended that they are explicitly initialized in the main program.

Any ILE COBOL program within a run unit, having the same data description for the item as the program containing the item, can access and process the data item. For example, if program A had the following data description:

```
01 EXT-ITEM1          PIC 99 EXTERNAL.
```

Program B could access the data item by having the identical data description in its Working-Storage Section.

The size must be the same for the same named EXTERNAL data item in all module objects declaring it. If different sized EXTERNAL data items with the same name are declared in multiple ILE COBOL programs in a compilation unit, the longest size is used to represent the data item.

Also, when different sized EXTERNAL data items of the same name are represented in multiple program objects or service programs that are activated in the same activation group, and the later activated program object or service program has a larger size for the same named EXTERNAL data item, then the activation of the later activated program object or service program will fail.

The type consistency across data items of the same name that are declared in multiple ILE COBOL programs is not enforced by the ILE COBOL compiler. You are responsible for ensuring that the usage of these data items is consistent.

Remember, any program that has access to an EXTERNAL data item can change its value. Do not use this clause for data items you need to protect.

Sharing EXTERNAL Files

Using the EXTERNAL clause for files allows separately compiled programs within the run unit to have access to common files. These EXTERNAL files are handled as weak exports. Refer to *ILE Concepts* for further information about strong and weak exports.

When an EXTERNAL file is defined in multiple ILE COBOL programs, once it is opened by one of these ILE COBOL programs, it is accessible to all of the programs. Similarly, if one of the programs closes the EXTERNAL file, it is no longer accessible by any of the programs.

For multiple ILE COBOL programs in multiple module objects, a runtime consistency check is made the first time the ILE COBOL program declaring a given EXTERNAL file is called to see if the definition in that module object is consistent with the definitions in already called ILE COBOL programs in other module objects. If any inconsistency is found, then a runtime exception message is issued.

The example in Figure 59 on page 239 shows some of the advantages of using EXTERNAL files:

- The main program can reference the record area of the file, even though the main program does not contain any input-output statements.
- Each subprogram can control a single input-output function, such as OPEN or READ.
- Each program has access to the file.

The following table gives the program (or subprogram) name for the example in Figure 59 on page 239 and describes its function.

Table 16. Program Names for Input-Output Using EXTERNAL Files Example

Name	Function
EF1MAIN	This is the main program. It calls all the subprograms and then verifies the contents of a record area.

Table 16. Program Names for Input-Output Using EXTERNAL Files Example (continued)

Name	Function
EF1OPENO	This program opens the external file for output and checks the File Status Code.
EF1WRITE	This program writes a record to the external file and checks the File Status Code.
EF1OPENI	This program opens the external file for input and checks the File Status Code.
EF1READ	This program reads record from the external file and checks the File Status Code.
EF1CLOSE	This program closes the external file and checks the File Status Code.

The sample program also uses the EXTERNAL clause for a data item in the Working-Storage Section. This item is used for checking File Status Codes.

```

5722WDS V5R4M0 060210 LN  IBM ILE COBOL                CBLGUIDE/EXTLFL                ISERIES1  06/02/15 13:11:39                Page      2
                               S o u r c e
STMT PL SEQNBR -A 1 B.+....2....+....3....+....4....+....5....+....6....+....7..IDENTFCN  S COPYNAME  CHG DATE
 1 000100 IDENTIFICATION DIVISION.
 2 000200 PROGRAM-ID. EF1MAIN.
   000300*
   000400* This is the main program that controls
   000500* the external file processing.
   000600*
   000700
 3 000800 ENVIRONMENT DIVISION.
 4 000900 INPUT-OUTPUT SECTION.
 5 001000 FILE-CONTROL.
 6 001100     SELECT EF1
 7 001200     ASSIGN TO DISK-EFILE1
 8 001300     FILE STATUS IS EFS1
 9 001400     ORGANIZATION IS SEQUENTIAL.
   001500
10 001600 DATA DIVISION.
11 001700 FILE SECTION.
12 001800 FD  EF1 IS EXTERNAL
   001900     RECORD CONTAINS 80 CHARACTERS.
13 002000 01 EF-RECORD-1.
14 002100 05 EF-ITEM-1    PIC X(80).
   002200
15 002300 WORKING-STORAGE SECTION.
16 002400 01 EFS1          PIC 99 EXTERNAL.
   002500
17 002600 PROCEDURE DIVISION.
   002700 EF1MAIN-PROGRAM SECTION.
   002800 MAINLINE.
18 002900     CALL "EF1OPENO"
19 003000     CALL "EF1WRITE"
20 003100     CALL "EF1CLOSE"
21 003200     CALL "EF1OPENI"
22 003300     CALL "EF1READ"
23 003400     IF EF-RECORD-1 = "First Record" THEN
24 003500         DISPLAY "First record correct"
   003600     ELSE
25 003700         DISPLAY "First record incorrect"
26 003800         DISPLAY "Expected: First Record"
27 003900         DISPLAY "Found:      " EF-RECORD-1
   004000     END-IF
28 004100     CALL "EF1CLOSE"
29 004200     GOBACK.
30 004300 END PROGRAM EF1MAIN.
   004400
   004600
                               * * * * * E N D   O F   S O U R C E * * * * *

```

Figure 59. Input-Output Using EXTERNAL Files (Part 1 of 6)

```

          S o u r c e
STMT PL SEQNBR -A 1 B.+....2....+....3....+....4....+....5....+....6....+....7..IDENTFCN S COPYNAME  CHG DATE
 1 004700 IDENTIFICATION DIVISION.
 2 004800 PROGRAM-ID. EF1OPENO.
   004900*
   005000* This program opens the external file for output.
   005100*
   005200
 3 005300 ENVIRONMENT DIVISION.
 4 005400 INPUT-OUTPUT SECTION.
 5 005500 FILE-CONTROL.
 6 005600     SELECT EF1
 7 005700     ASSIGN TO DISK-EFILE1
 8 005800     FILE STATUS IS EFS1
 9 005900     ORGANIZATION IS SEQUENTIAL.
   006000
10 006100 DATA DIVISION.
11 006200 FILE SECTION.
12 006300 FD EF1 IS EXTERNAL
   006400     RECORD CONTAINS 80 CHARACTERS.
13 006500 01 EF-RECORD-1.
14 006600 05 EF-ITEM-1 PIC X(80).
   006700
15 006800 WORKING-STORAGE SECTION.
16 006900 01 EFS1 PIC 99 EXTERNAL.
   007000
17 007100 PROCEDURE DIVISION.
   007200 EF1OPENO-PROGRAM SECTION.
   007300 MAINLINE.
18 007400     OPEN OUTPUT EF1
19 007500     IF EFS1 NOT = 0 THEN
20 007600         DISPLAY "File Status " EFS1 " on OPEN OUTPUT"
21 007700     STOP RUN
   007800     END-IF
22 007900     GOBACK.
23 008000 END PROGRAM EF1OPENO.
   008100
   008300
          * * * * * E N D   O F   S O U R C E   * * * * *

```

Figure 59. Input-Output Using EXTERNAL Files (Part 2 of 6)

```

          S o u r c e
STMT PL SEQNBR -A 1 B.+....2....+....3....+....4....+....5....+....6....+....7..IDENTFCN S COPYNAME  CHG DATE
 1 008400 IDENTIFICATION DIVISION.
 2 008500 PROGRAM-ID. EF1WRITE.
   008600*
   008700* This program writes a record to the external file.
   008800*
   008900
 3 009000 ENVIRONMENT DIVISION.
 4 009100 INPUT-OUTPUT SECTION.
 5 009200 FILE-CONTROL.
 6 009300     SELECT EF1
 7 009400     ASSIGN TO DISK-EFILE1
 8 009500     FILE STATUS IS EFS1
 9 009600     ORGANIZATION IS SEQUENTIAL.
   009700
10 009800 DATA DIVISION.
11 009900 FILE SECTION.
12 010000 FD EF1 IS EXTERNAL
   010100     RECORD CONTAINS 80 CHARACTERS.
13 010200 01 EF-RECORD-1.
14 010300 05 EF-ITEM-1 PIC X(80).
   010400
15 010500 WORKING-STORAGE SECTION.
16 010600 01 EFS1 PIC 99 EXTERNAL.
   010700
17 010800 PROCEDURE DIVISION.
   010900 EF1WRITE-PROGRAM SECTION.
   011000 MAINLINE.
18 011100     MOVE "First record" TO EF-RECORD-1
19 011200     WRITE EF-RECORD-1
20 011300     IF EFS1 NOT = 0 THEN
21 011400         DISPLAY "File Status " EFS1 " on WRITE"
22 011500         STOP RUN
   011600     END-IF
23 011700     GOBACK.
24 011800 END PROGRAM EF1WRITE.
   011900
   012100
          * * * * * E N D   O F   S O U R C E   * * * * *

```

Figure 59. Input-Output Using EXTERNAL Files (Part 3 of 6)

```

          S o u r c e
STMT PL SEQNBR -A 1 B.+....2....+....3....+....4....+....5....+....6....+....7..IDENTFCN S COPYNAME  CHG DATE
 1      012200 IDENTIFICATION DIVISION.
 2      012300 PROGRAM-ID. EF1OPENI.
        012400*
        012500* This program opens the external file for input.
        012600*
        012700
 3      012800 ENVIRONMENT DIVISION.
 4      012900 INPUT-OUTPUT SECTION.
 5      013000 FILE-CONTROL.
 6      013100     SELECT EF1
 7      013200     ASSIGN TO DISK-EFILE1
 8      013300     FILE STATUS IS EFS1
 9      013400     ORGANIZATION IS SEQUENTIAL.
        013500
10      013600 DATA DIVISION.
11      013700 FILE SECTION.
12      013800 FD EF1 IS EXTERNAL
        013900     RECORD CONTAINS 80 CHARACTERS.
13      014000 01 EF-RECORD-1.
14      014100 05 EF-ITEM-1 PIC X(80).
        014200
15      014300 WORKING-STORAGE SECTION.
16      014400 01 EFS1 PIC 99 EXTERNAL.
        014500
17      014600 PROCEDURE DIVISION.
        014700 EF1OPENI-PROGRAM SECTION.
        014800 MAINLINE.
18      014900     OPEN INPUT EF1
19      015000     IF EFS1 NOT = 0 THEN
20      015100         DISPLAY "File Status " EFS1 " on OPEN INPUT"
21      015200     STOP RUN
        015300     END-IF
22      015400     GOBACK.
23      015500 END PROGRAM EF1OPENI.
        015600
        015800
          * * * * * E N D   O F   S O U R C E   * * * * *

```

Figure 59. Input-Output Using EXTERNAL Files (Part 4 of 6)

```

          S o u r c e
STMT PL SEQNBR -A 1 B.+....2....+....3....+....4....+....5....+....6....+....7..IDENTFCN S COPYNAME  CHG DATE
 1 015900 IDENTIFICATION DIVISION.
 2 016000 PROGRAM-ID. EF1READ.
   016100*
   016200* This program reads a record from the external file.
   016300*
   016400
 3 016500 ENVIRONMENT DIVISION.
 4 016600 INPUT-OUTPUT SECTION.
 5 016700 FILE-CONTROL.
 6 016800     SELECT EF1
 7 016900     ASSIGN TO DISK-EFILE1
 8 017000     FILE STATUS IS EFS1
 9 017100     ORGANIZATION IS SEQUENTIAL.
   017200
10 017300 DATA DIVISION.
11 017400 FILE SECTION.
12 017500 FD EF1 IS EXTERNAL
   017600     RECORD CONTAINS 80 CHARACTERS.
13 017700 01 EF-RECORD-1.
14 017800 05 EF-ITEM-1 PIC X(80).
   017900
15 018000 WORKING-STORAGE SECTION.
16 018100 01 EFS1 PIC 99 EXTERNAL.
   018200
17 018300 PROCEDURE DIVISION.
   018400 EF1READ-PROGRAM SECTION.
   018500 MAINLINE.
18 018600     READ EF1
19 018700     IF EFS1 NOT = 0 THEN
20 018800         DISPLAY "File Status " EFS1 " on READ"
21 018900     STOP RUN
   019000     END-IF
22 019100     GOBACK.
23 019200 END PROGRAM EF1READ.
   019300
   019500
          * * * * * E N D   O F   S O U R C E   * * * * *

```

Figure 59. Input-Output Using EXTERNAL Files (Part 5 of 6)

```

5722WDS V5R4M0 060210 LN  IBM ILE COBOL          CBLGUIDE/EXTLFL          ISERIES1  06/02/15 13:11:39          Page    17
                               S o u r c e
STMT PL SEQNBR -A 1 B.+....2....+....3....+....4....+....5....+....6....+....7..IDENTFCN  S COPYNAME  CHG DATE
 1  019600 IDENTIFICATION DIVISION.
 2  019700 PROGRAM-ID. EF1CLOSE.
    019800*
    019900* This program reads a record from the external file.
    020000*
    020100
 3  020200 ENVIRONMENT DIVISION.
 4  020300 INPUT-OUTPUT SECTION.
 5  020400 FILE-CONTROL.
 6  020500     SELECT EF1
 7  020600     ASSIGN TO DISK-EFILE1
 8  020700     FILE STATUS IS EFS1
 9  020800     ORGANIZATION IS SEQUENTIAL.
    020900
10  021000 DATA DIVISION.
11  021100 FILE SECTION.
12  021200 FD  EF1 IS EXTERNAL
    021300     RECORD CONTAINS 80 CHARACTERS.
13  021400 01 EF-RECORD-1.
14  021500 05 EF-ITEM-1    PIC X(80).
    021600
15  021700 WORKING-STORAGE SECTION.
16  021800 01 EFS1          PIC 99 EXTERNAL.
    021900
17  022000 PROCEDURE DIVISION.
    022100 EF1CLOSE-PROGRAM SECTION.
    022200 MAINLINE.
18  022300     CLOSE EF1
19  022400     IF EFS1 NOT = 0 THEN
20  022500         DISPLAY "File Status " EFS1 " on CLOSE"
21  022600     STOP RUN
    022700     END-IF
22  022800     GOBACK.
23  022900 END PROGRAM EF1CLOSE.
    023000
    023100
                               * * * * *  E N D   O F   S O U R C E  * * * * *

```

Figure 59. Input-Output Using EXTERNAL Files (Part 6 of 6)

Passing Data Using Pointers

You can use a pointer within an ILE COBOL program when you want to pass and receive addresses of a dynamically-located data item.

For a full description of how pointers are used in an ILE COBOL program, refer to Chapter 14, "Using Pointers in an ILE COBOL Program," on page 335.

Passing Data Using Data Areas

A data area is an IBM i object used to communicate data such as variable values between programs within a job and between jobs. A data area can be created and declared to a program before it is used in that program or job. For information on how to create and declare a data area, see the *CL Programming* manual.

Using Local Data Area

The local data area can be used to pass any desired information between programs in a job. This information may be free-form data, such as informal messages, or may consist of a fully structured or formatted set of fields.

Internal and external floating-point data items can be passed using the local data area. Internal floating-point numbers written to the local data area using a DISPLAY statement are converted to external floating-point numbers.

The system automatically creates a local data area for each job. The local data area is defined outside the ILE COBOL program as an area of 1024 bytes.

When a job is submitted, the submitting job's local data area is copied into the submitted job's local data area. If there is no submitting job, the local data area is initialized to blanks.

An ILE COBOL program can access the local data area for its job with the ACCEPT and DISPLAY statements, using a mnemonic name associated with the environment-name LOCAL-DATA.

There is only one local data area associated with each job. Even if several work stations are acquired by a single job, only one local data area exists for that job. There is *not* a local data area for each workstation.

Using Data Areas You Create

You can pass data between programs using data areas that you create. This information may be free-form data, such as informal messages, or may consist of a fully structured or formatted set of fields. You specify the library and the name of the data area when you create it.

Using the Data Area formats (as opposed to the Local Data Area formats) of the ACCEPT and DISPLAY statements, you can access these data areas. The FOR phrase allows you to specify the name of the data area. Optionally, you can specify an IN LIBRARY phrase to indicate the IBM i library where the data area exists. If the IN LIBRARY phrase is not specified, the library defaults to *LIBL.

When you use the DISPLAY statement to write data to a data area you have created, it is locked by the system with a LEAR (Lock Exclusive Allow Read) lock before any data is written to the data area. If any other lock exists on the data area, the LEAR lock is not applied, and the data area is not written. By specifying the WITH LOCK phrase, you can keep the data area locked after the Display operation has completed.

When you use the ACCEPT statement to retrieve data from a data area you have created, the system applies an LSRD (Lock Shared for Read) lock to prevent the data area from being changed while it is read. After the read is complete, the LSRD lock is removed, and a LEAR lock is placed on the data area if a WITH LOCK phrase was specified.

For both the ACCEPT and DISPLAY statements, if a WITH LOCK phrase was not specified, any LEAR lock held prior to the statement will be removed.

In ILE COBOL only one LEAR lock will be placed on a data area while the COBOL Run unit (activation group) is active. If any data areas remain locked when an activation group ends, the locks are removed.

An ON EXCEPTION condition can exist for several reasons:

- Data area specified in the FOR phrase:
 - Cannot be found
 - You do not have authority to the data area
 - The data area was locked in a previous activation group or in another job
- AT position:
 - Was less than 1 or greater than the length of the data area.

Internal and external floating-point data items can be passed using a data area. Internal floating-point numbers written to the data area using a DISPLAY statement are converted to external floating-point numbers.

ILE COBOL supports decimal (*DEC), character (*CHAR), logical (*LGL), and DDM (*DDM) data areas. Regardless of the type of data area, information is moved to and from a data area left-justified. When referencing a decimal data area, or a logical data area, the AT position, if specified, must be 1.

Data is moved in packed format to and from a decimal data area. A decimal data area is created with a specified number of total digits and decimal digits. This same number of digits must be declared in an ILE COBOL program accessing the decimal data area. For example:

- CL command to create the data area:
CRTDTAARA DTAARA(QGPL/DECDATA) TYPE(*DEC) LEN(5 2)
- Partial ILE COBOL program to access data area:
WORKING-STORAGE SECTION.
01 data-value.
05 returned-packed1 pic s9(3)v9(2) packed-decimal.
PROCEDURE DIVISION.
move 345.67 to returned-packed1.
DISPLAY data-value UPON data-area
FOR "DECDATA" LIBRARY "QGPL".
ACCEPT data-value FROM data-area
FOR "DECDATA" LIBRARY "QGPL".

Using Program Initialization Parameters (PIP) Data Area

The PIP data area is used by a prestart job. Generally, a prestart job is a job from a remote system under ICF that you start and keep ready to run until you call it.

If you use a prestart job, you do not have to wait for a program that you call to go through job initiation processing. Job initiation is performed before a program can actually start. Because job initiation has already taken place, a prestart job allows your program to start more quickly after the program start request is received.

An ILE COBOL program can access the PIP data area for its job with the ACCEPT statement, using a mnemonic name associated with the function-name PIP-DATA.

The PIP data area is a 2 000-byte alphanumeric item and contains parameters received from a calling program. It provides the program initialization parameters that, in non-prestart jobs, is provided through Standard COBOL parameters.

You use a Format 5 ACCEPT statement to access the PIP data area, similar to the way in which you use a Format 4 ACCEPT statement to read from the local data area. Note that you cannot update the PIP data area using ILE COBOL. See the *IBM Rational Development Studio for i: ILE COBOL Reference* for detailed syntax information.

For more information regarding prestart jobs and the PIP data area, refer to the *CL Programming* manual.

Effect of EXIT PROGRAM, STOP RUN, GOBACK, and CANCEL on Internal Files

The following statements affect the state of a file differently:

- An EXIT PROGRAM statement does not change the status of any of the files in a run unit unless:
 - The ILE COBOL program issuing the EXIT PROGRAM has the INITIAL attribute. If it has the INITIAL attribute, then all internal files defined in that program are closed.
 - An EXIT PROGRAM statement with the AND CONTINUE RUN UNIT phrase is issued in the main program of a *NEW activation group. In this case, control returns from the main program to the caller, which, in turn, causes the *NEW activation group to end, closing all of the files scoped to the activation group.
- A STOP RUN statement returns control to the caller of the program at the nearest control boundary. If this is a hard control boundary, the activation group (run unit) will end, and all files scoped to the activation group will be closed.
- A GOBACK statement issued from a main program (which is always at a hard control boundary) behaves the same as the STOP RUN statement. A GOBACK statement issued from a subprogram behaves the same as the EXIT PROGRAM statement. It does not change the status of any of the files in a run unit unless the ILE COBOL program issuing the GOBACK has the INITIAL attribute. If it has the INITIAL attribute, then all internal files defined in that program are closed.
- A CANCEL statement resets the storage that contains information about the internal file. If the program has internal files that are open when the CANCEL statement is processed, those internal files are closed when that program is canceled. The program can no longer use the files unless it reopens them. If the canceled program is called again, the program considers the file closed. If the program opens the file, a new linkage to the file is established.

Canceling an ILE COBOL Program

A subprogram, unless it has the INITIAL attribute, is left in its last-used state when it ends with EXIT PROGRAM or GOBACK. A subprogram that uses the EXIT PROGRAM statement with the AND CONTINUE RUN UNIT phrase is also left in its last-used state. The next time it is called in the run unit, its internal values will be as they were left, except for PERFORM statements, which are reset.

To reset the internal values of a subprogram to their initial state before it is called again, you must cancel the subprogram. Canceling the subprogram ensures that the next time the subprogram is called, it will be entered in its initial state.

Canceling from Another ILE COBOL Program

In ILE COBOL, you use the CANCEL statement to cancel a subprogram. The subprogram must be in the same activation group as the program that is canceling it in order for the CANCEL statement to work.

After a CANCEL statement for a called subprogram has been executed, that subprogram no longer has a logical connection to the program. The contents of data items in EXTERNAL data records and EXTERNAL files described by the subprogram are not changed when a subprogram is canceled. If a CALL statement is executed later in the run unit naming the same subprogram, that subprogram will be entered in its initial state.

Called subprograms may contain CANCEL statements; however, a called subprogram must not contain a CANCEL statement that directly or indirectly cancels its calling program or any other program higher than itself in the calling

hierarchy. If a called subprogram attempts to cancel its calling program, the CANCEL statement in the subprogram is ignored.

A program named in a CANCEL statement must not refer to any program that has been called and has not yet returned control to the calling program. A program can cancel any program that has been called and already returned from, provided that they are in the same activation group. For example:

A calls B and B calls C	When A receives control, it can cancel C.
A calls B and A calls C	When C receives control, it can cancel B.

Note: When canceling a program object that contains multiple ILE COBOL programs, only the ILE COBOL program associated with the PEP of the program object is canceled.

Refer to the *IBM Rational Development Studio for i: ILE COBOL Reference* for a full description of the CANCEL statement.

Canceling from Another Language

You can cancel an outermost ILE COBOL program from ILE RPG, ILE C, and ILE CL, by calling its cancel procedure using a static procedure call. The name of the cancel procedure is formed by taking the name of the outermost ILE COBOL program and adding the suffix `_reset`.

You cannot cancel an ILE COBOL program from an OPM COBOL/400 program or an OPM RPG/400 program.

Do not use the Reclaim Resources (RCLSRC) CL command to cancel an ILE
COBOL program. If RCLRSC is issued within an ILE activation group, it will cause
an exception. For more information on the RCLRSC command, refer to the *CL and*
APIs section of the *Programming* category in the **i5/OS Information Center** at this
Web site -<http://www.ibm.com/systems/i/infocenter/>.

Chapter 10. COBOL and the eBusiness World

This chapter describes how you can use ILE COBOL as part of an eBusiness solution. It includes:

- “COBOL and XML”
- “COBOL and MQSeries”
- “COBOL and Java Programs” on page 250

COBOL and XML

The Extensible Markup Language (XML) is a subset of SGML that is developed by the World Wide Web Consortium (W3C). Its goal is to enable generic SGML to be served, received, and processed on the Web in the way that is now possible with HTML. XML has been designed for ease of implementation and for interoperability with both SGML and HTML. You can use XML as both a datastore and I/O mechanism.

A well-formedness XML parser has been added into ILE COBOL, for more information see Chapter 11, “Processing XML Input,” on page 277.

For more information about XML, see <http://www.w3.org/XML>

IBM has developed two tools which can be used to integrate XML and COBOL programs. IBM's XML for C++ parser (XML4C) is a validating XML parser written in a portable subset of C++. It consists of three shared libraries (2 code and 1 data) which provide classes for parsing, generating, manipulating, and validating XML documents.

#

In order to use the parser with procedural languages such as C, COBOL and RPG, you will also need XML Interface for RPG. This wrapper interface allows ILE C, RPG and COBOL programs on the i5/OS to interface with the XML parser.

Both these products are constantly evolving. They are available through alphaWorks, which gives early adopters direct access to IBM's emerging “alpha-code” technologies. You can download alpha code and participate in online discussions with IBM's researchers and developers. For the latest information about these alpha technologies, including hardware and software requirements, see <http://www.alphaWorks.ibm.com/>

COBOL and MQSeries

IBM MQSeries messaging products enable application integration by helping business applications to exchange information across different platforms by sending and receiving data as messages. They take care of network interfaces, assure 'once only' delivery of messages, deal with communications protocols, dynamically distribute workload across available resources, handle recovery after system problems, and help make programs portable. MQSeries is available on over 35 platforms.

With MQSeries for iSeries V5.2, your ILE COBOL program can communicate with other programs on the same platform or a different platform using the same

messaging product. For more information on MQSeries for iSeries V5.2, including hardware and software requirements, please see the IBM Systems Software Information Center.

For several examples of i5/OS COBOL applications using the MQSeries API, check
the MQSeries documentation. In the Information Center, click **Programming >**
WebSphere MQ.

COBOL and Java Programs

The Java Native Interface (JNI) allows Java code inside a Java Virtual Machine (JVM) to interoperate with applications and libraries that are written in other programming languages, such as COBOL, RPG, C, C++, and Assembler. This chapter describes how to make COBOL and Java programs work together, using the JNI.

You can also use the following components of the IBM Toolbox for Java to combine COBOL and Java programs:

- # • The ProgramCall class in the IBM Toolbox for Java uses the IBM i Host System
Program Call driver to call IBM i program objects.
- # • Program Call Markup Language (PCML) is a tag language based on the
Extensible Markup Language (XML). You can generate tags that fully describe
the input and output parameters for ILE COBOL programs that are called by
your Java application by specifying the PGMINFO and INFOSTMF parameters
on the COBOL command. The IBM Toolbox for Java includes an application
programming interface (API) that interprets the PCML, calls the program, and
simplifies the retrieval of data returned from the System i machine.

For more information about these approaches, refer to the *Java* section of the
Programming category in the **i5/OS Information Center** at this Web site -
<http://www.ibm.com/systems/i/infocenter/>.

System Requirements

In order to to integrate COBOL and Java programs, consider the following requirements:

- IBM Qshell Interpreter is a no-charge option of IBM i (5722-SS1, option 30).
- IBM Developer Kit for Java (5722-JV1) is a WebSphere Development Studio for i5/OS component and can be specified at installation time.
- IBM's Java 2 Software Development Kit (J2SDK), Standard Edition v1.2.2 is shipped with WebSphere Development Studio for i5/OS. If you choose to use a different Java Development Kit, then it must be version 1.2 or higher to ensure the functionality of all pieces of Java code.
- IBM Toolbox for Java (5722-JC1) is required to use IBM Toolbox for Java classes, including PCML. It is provided with WebSphere Development Studio for i5/OS component and can be specified at installation time.

COBOL and PCML

A COBOL program can be called from a Java application using a Program Call
Markup Language (PCML) source file that describes the input and output
parameters for the COBOL program. The Java application can use PCML by
constructing a ProgramCallDocument object with a reference to the PCML source
file. See **Programming->Java->IBM Toolbox for Java->Program Call Markup**
Language in the i5/OS Information Center at <http://www.ibm.com/systems/i/>

infocenter/ for more information on how to use PCML with Java. PCML handles
the data type conversions between the COBOL format and the Java format.

The ILE COBOL compiler will generate PCML for your COBOL program when
you specify the PGMINFO command parameter or the PGMINFO PROCESS
option. The PCML can be generated in a stream file or it can be made part of your
COBOL module. If the PCML is part of your COBOL module, it can be retrieved
from a program or service program containing your module using the QBNRPII
API.

To have the PCML generated into a stream file, specify the PGMINFO(*PCML)
command parameter along with the INFOSTMF compiler parameter to specify the
name of an Integrated File System output file to receive the generated file.

To have the PCML generated directly into the COBOL module, specify the
PGMINFO(*PCML *MODULE) command parameter, or specify the
PGMINFO(PCML MODULE) PROCESS option.

You can have the PCML generated both into a stream file and into the COBOL
module by specifying PGMINFO(*PCML *ALL) command parameter along with
the INFOSTMF parameter, or by specifying the PGMINFO(*PCML) and
INFOSTMF command parameters, and specifying the PGMINFO(PCML MODULE)
PROCESS option.

If you specify PROCESS option PGMINFO(NOPGMINFO), no PCML will be
generated even if you specified the PGMINFO(*PCML) command parameter.

17 shows the support in PCML for the COBOL datatypes:

Table 17. COBOL Datatypes and Corresponding PCML Support

COBOL Data Type	COBOL Format	Supported in PCML	PCML Data Type	Length	Precision	Count
Character	X(n)	Yes	character	n		
	A(n)	Yes	character	n		
	X(n) OCCURS DEPENDING ON M	Yes	structure			m
	A(n) OCCURS DEPENDING ON m	Yes	structure			m

Table 17. COBOL Datatypes and Corresponding PCML Support (continued)

Numeric	9(n) DISPLAY	Yes	zoned decimal	n	0	
	S9(n-p)V9(p) DISPLAY	Yes	zoned decimal	n	p	
	9(n-p)V9(p) PACKED-DECIMAL see note 3	Yes	packed decimal	n	p	
	S9(n-p)V9(p) PACKED-DECIMAL see note 3	Yes	packed decimal	n	p	
	9(4) BINARY see notes 1, 2	Yes	integer	2	15	
	9(4) COMP-5	Yes	integer	2	16	
	S9(4) BINARY S9(4) COMP-5 see notes 1, 2	Yes	integer	2	15	
	9(9) BINARY see notes 1, 2	Yes	integer	4	31	
	9(9) COMP-5	Yes	integer	4	32	
	S9(9) BINARY S9(9) COMP-5 see notes 1, 2	Yes	integer	4	31	
	S9(18) BINARY S9(18) COMP-5 see notes 1, 2	Yes	integer	8	63	
	9(18) BINARY see notes 1, 2	Yes	integer	8	63	
		9(18) COMP-5	not supported	integer	8	64
	9(n)V9(p) BINARY S9(n)V9(p) BINARY 9(n)V9(p) COMP-5 S9(n)V9(p) COMP-5	not supported				
	USAGE COMP-1	Yes	float	4		
	USAGE COMP-2	Yes	float	8		
UCS2	N(n) see note 4	Yes	UCS-2/graphics	n		
	N(n) OCCURS DEPENDING ON m see note 4	Yes	structure			m
Graphic	G(n)	Yes	UCS-2/graphics	n		
	G(n) OCCURS DEPENDING ON M	Yes	structure			m
Index	USAGE INDEX	Yes	integer	4	31	
Boolean	1	not supported				
Date	FORMAT DATE	not supported				
Time	FORMAT TIME	not supported				
Timestamp	FORMAT TIMESTAMP	not supported				
Pointer	USAGE POINTER	not supported				

Table 17. COBOL Datatypes and Corresponding PCML Support (continued)

Procedure Pointer	PROCEDURE POINTER	not supported				
-------------------	-------------------	---------------	--	--	--	--

Notes:

1. To reduce truncation for BINARY data items, specify NOSTDTRUNC on the PROCESS statement. NOSTDTRUNC should always be specified when passing BINARY data items.
2. BINARY, COMP-4, COMPUTATIONAL-4 map to an integer in PCML.
3. PACKED-DECIMAL, COMP-3, COMPUTATIONAL-3, COMP, and COMPUTATIONAL are equivalent and map to the same PCML (unless COMPASBIN PROCESS option is specified, see "PROCESS Statement Options" on page 59 for more information).
4. PIC N is a national (UCS2) item if USAGE NATIONAL is specified or if USAGE is not specified and the NATIONAL compiler option is specified, otherwise USAGE DISPLAY-1 (DBCS) is implied.

PCML is generated based on the contents of the Procedure Division USING and GIVING/RETURNING phrases and the contents of the LINKAGE section in your COBOL program. PCML will be generated for all parameters specified in the PROCEDURE DIVISION header USING phrase. PCML will be generated for a parameter specified in the GIVING/RETURNING phrase for this header. An error will be issued if the GIVING/RETURNING item is not a 4 byte signed binary integer. Items specified in the USING phrase that are defined as "inputoutput" in the generated PCML can be used to return information to the calling program. Items defined with the TYPE clause will receive a PCML error. For the calling program (eg JAVA program) to see the contents of the RETURN-CODE special register, the RETURN-CODE special register must be specified on the USING phrase of the PROCEDURE DIVISION header. The object data item for an OCCURS DEPENDING ON (ODO) must be defined in the linkage section and be specified as a parameter in the PROCEDURE DIVISION header USING phrase for PCML to be correctly generated for the ODO subject data item.

PCML will not be generated for renamed/redefined items.

When you use CRTCBMOD, and create a service program, you specify the service program in your Java code using the setPath(String) method of the ProgramCallDocument class. For example:

```
AS400 as400;
ProgramCallDocument pcd;
String path = "/QSYS.LIB/MYLIB.LIB/MYSRVPGM.SRVPGM";
as400 = new AS400 ();
pcd = new ProgramCallDocument (as400, "myModule");
pcd.setPath ("MYFUNCTION", path);
pcd.setValue ("MYFUNCTION.PARM1", "abc");
rc = pcd.callProgram("MYFUNCTION");
```

If you use CRTCBMOD and create a program, not a service program, you will need to remove the entrypoint attribute from the PCML, since this attribute is needed only when calling service programs.

Example:

The following is an example COBOL source program and corresponding PCML generated for this program:

```

5722WDS V5R4M0 060210 LN  IBM ILE COBOL          TESTLIB/MYPCML          ISERIES1 06/02/15 12:09:25          Page 2
                               S o u r c e
STMT PL SEQNBR -A 1 B.+. . . . . 2 . . . . . 3 . . . . . 4 . . . . . 5 . . . . . 6 . . . . . 7 . IDENTFCN S COPYNAME  CHG DATE
 1      000100 IDENTIFICATION DIVISION.
 2      000200 PROGRAM-ID.          MYPGM4.
      000300
 3      000400 DATA DIVISION.
 4      000500 WORKING-STORAGE SECTION.
 5      000600 01 RETN-VAL PIC S9(8) USAGE COMP-4.
      000700
 6      000800 LINKAGE SECTION.
 7      000900 01 PARM-LIST.
 8      001000 05 EMPL OCCURS 5 TIMES.
 9      001100 10 NAMES             PIC A(20).
10     001200 10 ADDRESSES          PIC X(60).
11     001300 10 PHN-NUM            PIC 9(11) DISPLAY.
12     001400 05 NUM-1A             PIC S9(5)V9(3) PACKED-DECIMAL.
13     001500 05 NUM-2A             PIC 9(5)V9(3) COMP.
14     001600 05 TAB-NUM-3A         PIC S9(5)V9(3) COMP OCCURS 10 TIMES.
15     001700 05 NUM-4A             PIC 9(5)V9(3) COMP-3.
16     001800 05 NUM-5A             PIC S9(5)V9(3) COMP-3.
17     001900 05 NUM-6A             PIC 9(4) BINARY.
18     002000 05 NUM-7A             COMP-1.
19     002100 05 NUM-8A             COMP-2.
20     002200 05 INTLNAME           PIC N(10) NATIONAL.
      002300
      002400*****
      002500* Test PCML for arrays of basic supported types.
      002600*****
21     002700 PROCEDURE DIVISION USING BY REFERENCE PARM-LIST
      002800 GIVING RETN-VAL.
      002900 MAIN-LINE.
22     003000 MOVE 1 TO RETN-VAL.
23     003100 DISPLAY "THIS PGM TO BE CALLED BY A JAVA PGM".
24     003200 STOP RUN.
          ***** END OF SOURCE *****

```

Figure 60. PCML Source Program

The following is an example of PCML that is generated when the program is compiled with options PGMINFO(*PCML) and INFOSTMF('/dirname/mypgm4.pcm') specified on the CRTBNDCBL command:

```

<pcm version="4.0">
  <!-- COBOL program: MYPCML -->
  <!-- created: 02/03/21 12:09:25 -->
  <!-- source: TESTLIB/QCBLLESRC(MYPCML) -->
  <programname="MYPCML" path="/QSYS.LIB/TESTLIB.LIB/MYPCML.PGM" returnvalue="integer">
    <struct name="PARM-LIST" usage="inputoutput">
      <struct name="EMPL" usage="inherit" count="5">
        <data name="NAMES" type="char" length="20" usage="inherit">
          <data name="ADDRESSES" type="char" length="60" usage="inherit">
            <data name="PHN-NUM" type="zoned" length="11" precision="0" usage="inherit">
          </struct>
          <data name="NUM-1A" type="packed" length="8" precision="3" usage="inherit">
          <data name="NUM-2A" type="packed" length="8" precision="3" usage="inherit">
          <data name="TAB-NUM-3A" type="packed" length="8" precision="3" count="10"
usage="inherit">
          <data name="NUM-4A" type="packed" length="8" precision="3" usage="inherit">
          <data name="NUM-5A" type="packed" length="8" precision="3" usage="inherit">
          <data name="NUM-6A" type="int" length="2" precision="16" usage="inherit">
          <data name="NUM-7A" type="float" length="4" usage="inherit">
          <data name="NUM-8A" type="float" length="8" usage="inherit">
          <data name="INTLNAME" type="char" length="10" chartype="twobyte" ccid="13488" usage="inherit">
        </struct>
        <data name="RETN-VAL" type="int" length="4" precision="32" passby="value"
usage="output">
      </program></pcm>

```

COBOL and JNI

Calling a COBOL Program from a Java Program

To call a COBOL program from a Java program, perform the following steps:

- “Code the COBOL Program”
- “Create the COBOL Module” on page 259
- “Create a Service Program” on page 260
- “Code the Java Program” on page 261
- “Compile the Java Program” on page 262.

Code the COBOL Program: This section describes how to code a COBOL program that is called by a Java program. The guidelines are illustrated in two sample COBOL programs. A later section shows two Java programs that interact with these COBOL programs.

If your COBOL program will be called by a Java program:

1. Use the PROCESS statement NOMONOPRC (for case-sensitive names) and the option THREAD(SERIALIZE). When the COBOL program is invoked from a Java program, it will run in a Java thread. Specify the NOSTDTRUNC Process option to preserve the content of binary data items.
2. Identify the COBOL program with a name that consists of:
 - The prefix Java_
 - A mangled fully-qualified class name
 - An underscore (_) separator
 - A mangled method name
 - For an overloaded native method, two underscores (__) followed by the mangled argument signature.
3. Copy the predefined interface function table into the program. For a listing of the predefined interface function table, see “Member JNI” on page 270.
4. To pass a variable from a COBOL program to a Java program, specify the BY VALUE phrase on the CALL. Receive the following arguments, in the following order:
 - a. The JNI interface pointer
 - b. A reference to the Java class (for a static native method) or to the object (for a nonstatic native method)
 - c. Any additional arguments that are required. These arguments correspond to regular Java method arguments.

Note that COBOL and Java data types are not completely equivalent. See “COBOL and Java Data Types” on page 268.

PROCESS NOMONOPRC NOSTDTRUNC OPTIONS THREAD (SERIALIZE). **1**

*** COBOL native program called from Java
*** static method

IDENTIFICATION DIVISION.
PROGRAM-ID. "Java_Hello_displayHello". **2**
Author.
INSTALLATION. IBM Toronto Lab.
DATE-WRITTEN.
DATE-COMPILED.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. IBM-ISERIES
OBJECT-COMPUTER. IBM-ISERIES

INPUT-OUTPUT SECTION.
FILE-CONTROL.

DATA DIVISION.
FILE SECTION.
WORKING-STORAGE SECTION.

01 IS-COPY PIC 1.
01 NAME-PTR USAGE POINTER.
01 NAME-LENGTH PIC 9(4) BINARY.
01 I PIC 9(4) BINARY.

01 NAME-X.
05 CHAR-X OCCURS 20 TIMES PIC X.

LINKAGE SECTION.

*** JNI interface function table

COPY JNI. **3**

01 NAME.
05 CHAR OCCURS 20 TIMES PIC N USAGE NATIONAL.

01 ENV-PTR USAGE POINTER.
01 CLASS-REF PIC S9(9) BINARY.
01 TITLE-CODE PIC S9(9) BINARY.
01 NAME-REF PIC S9(9) BINARY.

01 INTERFACE-PTR USAGE POINTER.

Figure 61. COBOL Program HELLO (Part 1 of 2)

```

PROCEDURE DIVISION USING BY VALUE ENV-PTR
CLASS-REF
TITLE-CODE
NAME-REF.

MAIN-LINE SECTION.
MAIN-PROGRAM-LOGIC.

    SET ADDRESS OF INTERFACE-PTR TO ENV-PTR.
    SET ADDRESS OF JNI-NATIVE-INTERFACE TO INTERFACE-PTR.

*** Callback JNI interface function GET-STRING-LENGTH to
*** retrieve the name length

    CALL GET-STRING-LENGTH USING BY VALUE ENV-PTR
                                NAME-REF
                                RETURNING INTO NAME-LENGTH.

*** Callback JNI interface function GET-STRING-CHARS to
*** retrieve the name characters

    CALL GET-STRING-CHARS USING BY VALUE ENV-PTR
                                NAME-REF
                                IS-COPY
                                RETURNING INTO NAME-PTR.

    SET ADDRESS OF NAME TO NAME-PTR.
    INITIALIZE NAME-X.

    PERFORM VARYING I FROM 1 BY 1 UNTIL (I > NAME-LENGTH)
        MOVE CHAR(I) TO CHAR-X(I)
    END-PERFORM.

    EVALUATE TITLE-CODE
        WHEN 1          DISPLAY "Hello, Mr. ", NAME-X
        WHEN 2          DISPLAY "Hello, Ms. ", NAME-X
        WHEN OTHER      DISPLAY "Hello, ", NAME-X
    END-EVALUATE.

    GOBACK.

```

Figure 61. COBOL Program HELLO (Part 2 of 2)

PROCESS NOMONOPRC NOSTDTRUNC OPTIONS THREAD(SERIALIZE).

1

*** COBOL native program called from Java
*** instance method

IDENTIFICATION DIVISION.
PROGRAM-ID. "Java_Bye_displayBye".
Author.
INSTALLATION. IBM Toronto Lab.
DATE-WRITTEN.
DATE-COMPILED.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. IBM-ISERIES
OBJECT-COMPUTER. IBM-ISERIES

INPUT-OUTPUT SECTION.
FILE-CONTROL.

DATA DIVISION.
FILE SECTION.

WORKING-STORAGE SECTION.

01 IS-COPY PIC 1.
01 NAME-PTR USAGE POINTER.
01 NAME-LENGTH PIC 9(4) BINARY.
01 I PIC 9(4) BINARY.

01 NAME-X.
05 CHAR-X OCCURS 20 TIMES PIC X.

LINKAGE SECTION.

*** JNI interface function table

COPY JNI.

3

01 NAME.
05 CHAR OCCURS 20 TIMES PIC N USAGE NATIONAL.

01 ENV-PTR USAGE POINTER.
01 OBJECT-REF PIC S9(9) BINARY.
01 TITLE-CODE PIC S9(9) BINARY.
01 NAME-REF PIC S9(9) BINARY.

01 INTERFACE-PTR USAGE POINTER.

Figure 62. COBOL Program BYE (Part 1 of 2)

```

PROCEDURE DIVISION USING BY VALUE ENV-PTR 4
                                OBJECT-REF
                                TITLE-CODE
                                NAME-REF.

MAIN-LINE SECTION.
MAIN-PROGRAM-LOGIC.

                                SET ADDRESS OF INTERFACE-PTR TO ENV-PTR.
                                SET ADDRESS OF JNI-NATIVE-INTERFACE TO INTERFACE-PTR.

*** Callback JNI interface function GET-STRING-LENGTH to
*** retrieve the name length

                                CALL GET-STRING-LENGTH USING BY VALUE ENV-PTR 4
                                                                NAME-REF
                                                                RETURNING INTO NAME-LENGTH.

*** Callback JNI interface function GET-STRING-CHARS to
*** retrieve the name characters

                                CALL GET-STRING-CHARS USING BY VALUE ENV-PTR 4
                                                                NAME-REF
                                                                IS-COPY
                                                                RETURNING INTO NAME-PTR.

                                SET ADDRESS OF NAME TO NAME-PTR.
                                INITIALIZE NAME-X.

                                PERFORM VARYING I FROM 1 BY 1 UNTIL (I > NAME-LENGTH)
                                    MOVE CHAR(I) TO CHAR-X(I)
                                END-PERFORM.

                                EVALUATE TITLE-CODE
                                    WHEN 1          DISPLAY "Bye, Mr. ", NAME-X
                                    WHEN 2          DISPLAY "Bye, Ms. ", NAME-X
                                    WHEN OTHER      DISPLAY "Bye, ", NAME-X
                                END-EVALUATE.

                                GOBACK.

```

Figure 62. COBOL Program BYE (Part 2 of 2)

Create the COBOL Module: To create a COBOL module, use the CRTCLMOD command, as shown in the examples on the two following screens.

```

Create COBOL Module (CRTCBMOD)

Type choices, press Enter.

Module . . . . . > BYE           Name, *PGMID
Library . . . . . > *CURLIB      Name, *CURLIB
Source file . . . . . > QCBLLSRC  Name
Library . . . . . > *LIBL        Name, *LIBL, *CURLIB
Source member . . . . . > BYE     Name, *MODULE
Source stream file . . . . .
Output . . . . . *PRINT          *PRINT, *NONE
Generation severity level . . . 30 0-30
Text 'description' . . . . . *SRCMBRTXT

Additional Parameters

Replace module . . . . . > *YES    *YES, *NO

F3=Exit  F4=Prompt  F5=Refresh  F10=Additional parameters  F12=Cancel
F13=How to use this display  F24=More keys

Bottom

```

```

Create COBOL Module (CRTCBMOD)

Type choices, press Enter.

Module . . . . . > HELLO         Name, *PGMID
Library . . . . . > *CURLIB      Name, *CURLIB
Source file . . . . . > QCBLLSRC  Name
Library . . . . . > *LIBL        Name, *LIBL, *CURLIB
Source member . . . . . > HELLO   Name, *MODULE
Source stream file . . . . .
Output . . . . . *PRINT          *PRINT, *NONE
Generation severity level . . . 30 0-30
Text 'description' . . . . . *SRCMBRTXT

Additional Parameters

Replace module . . . . . > *YES    *YES, *NO

F3=Exit  F4=Prompt  F5=Refresh  F10=Additional parameters  F12=Cancel
F13=How to use this display  F24=More keys

Bottom

```

Create a Service Program: Bind the module or modules into a service program, using the CRTSRVPGM command as shown below. Specify the EXPORT option.

```

                                Create Service Program (CRTSRVPGM)

Type choices, press Enter.

Service program . . . . . SRVPGM      > HELLOBYE
Library . . . . .           > *CURLIB
Module . . . . .           MODULE     > HELLO
Library . . . . .           > *CURLIB
                                + for more values > BYE
                                > *CURLIB

Export . . . . .           EXPORT     > *ALL
Export source file . . . . . SRCFILE  QSRVSRC
Library . . . . .           *LIBL
Export source member . . . . . SRCMBR *SRVPGM
Text 'description' . . . . . TEXT     *BLANK

                                                                More...
F3=Exit  F4=Prompt  F5=Refresh  F12=Cancel  F13=How to use this display
F24=More keys

```

Code the Java Program: This section describes how to code a Java program that calls a COBOL program. The guidelines are illustrated in two sample Java programs, which call the COBOL programs that were shown in a previous section.

Java source files are stored in the Integrated File System (IFS). You can use the stream file editor, EDTF, to edit these files.

If your Java program will call a COBOL program:

1. Make a static initializing call to the system method `System.loadLibrary` to load the COBOL service program that you created in the previous step. (In this example, the service program is named HELLOBYE.)
2. Declare the COBOL method with the keyword `native`. For the body of the native method, specify only a semicolon. This indicates that the implementation is omitted.

You can specify the short name (the name without the argument signature). The JVM will look for a method with this name in the native library; if that fails, the JVM will look for the long name. If you want to overload another native method, use the long name. If a native method has the same name as a Java method, you do not need to specify the long name because the Java method will not exist in the native library.

```

class Hello {
    static {
        System.loadLibrary("HELLOBYE");      1
    }

    static native void displayHello(int parm1, String parm2);      2

    public static void main(String[ ] args) {
        int titleCode;
        String name;

        switch (args.length) {
            case 1:
                titleCode = Integer.parseInt(args[0]);
                name = "Someone";
                break;

            case 2:
                titleCode = Integer.parseInt(args[0]);
                name = args[1];
                break;

            default:
                titleCode = 0;
                name = "Someone";
                break;
        }
        displayHello(titleCode, name);
        Bye bye = new Bye( );
        bye.displayBye(titleCode, name);
    }
}

```

Figure 63. Java Program Hello.java

```

class Bye {
    static {
        System.loadLibrary("HELLOBYE");      1
    }

    static native void displayBye(int parm1, String parm2);      2
}

```

Figure 64. Java Program Bye.java

Compile the Java Program: To compile the Java source programs, you can enter the Qshell interpreter (QSH) and issue the following commands:

```
javac Hello.java
```

```
javac Bye.java
```

Invoke the Java program: To invoke the Java source programs, you can enter the Qshell interpreter (QSH) and issue the following commands:

```

>java Hello
Hello, Someone
Bye, Someone
>java Hello 1
Hello, Mr. Someone
Bye, Mr. Someone
>java Hello 2 USA
Hello, Ms. USA
Bye, Ms. USA

```



```

FILE-CONTROL.

DATA DIVISION.
FILE SECTION.

WORKING-STORAGE SECTION.

*** JDK 1.2 VM initialization arguments

01 VM-INIT-ARGS.
   05 VERSION                PIC S9(9) BINARY VALUE 65538.
   05 NUMBER-OF-OPTIONS     PIC S9(9) BINARY.
   05 OPTIONS-PTR           USAGE POINTER.
   05 FILLER                 PIC X(1).

01 VM-OPTIONS.
   05 OPTIONS-STRING-PTR    USAGE POINTER.
   05 EXTRA-INFO-PTR       USAGE POINTER.

***

01 JVM-PTR                   USAGE POINTER.
01 ENV-PTR                   USAGE POINTER.

01 RC1                       PIC S9(9) BINARY VALUE 1.
01 RC2                       PIC S9(9) BINARY VALUE 1.
01 RC3                       PIC S9(9) BINARY VALUE 1.

01 CLASS-NAME                PIC X(30).
01 CLASS-NAME-PTR           USAGE POINTER.

01 METHOD-NAME                PIC X(30).
01 METHOD-NAME-PTR          USAGE POINTER.

01 SIGNATURE-NAME           PIC X(30).
01 SIGNATURE-NAME-PTR      USAGE POINTER.

*** CLASSPATH Parameters
01 CLASSPATH                 PIC X(500).

*** Object Reference Variables
01 MY-CLASS-REF              PIC S9(9) BINARY.
01 STRING-CLASS-REF         PIC S9(9) BINARY.
01 METHOD-ID                 PIC S9(9) BINARY.
01 INIT-METHOD-ID         PIC S9(9) BINARY.
01 STATIC-METHOD-ID       PIC S9(9) BINARY.
01 OBJECT-REF               PIC S9(9) BINARY.
01 ARG-REF                  PIC S9(9) BINARY.
01 STRING-REF               PIC S9(9) BINARY.

*** Parameter Array for calling METHODDA
01 PARM-ARRAY.
   05 PARM-ARRAY-ELEMENT OCCURS 10 TIMES.
       10 PARM-ARRAY-ELEMENT-VALUE PIC S9(9) BINARY.
       10 FILLER                    PIC X(4).

01 PARM-ARRAY-PTR           USAGE POINTER.

LINKAGE SECTION.

*** JNI interface function table

COPY JNI.  2

```

```
01 INTERFACE-PTR          USAGE POINTER.
01 JVM                    PIC S9(9) BINARY.
```

PROCEDURE DIVISION.

MAIN-LINE SECTION.
MAIN-PROGRAM-LOGIC.

```
3a      STRING FUNCTION UTF8STRING("-Djava.class.path=/home/myclass")
          DELIMITED BY SIZE
          X"00" DELIMITED BY SIZE
          INTO CLASSPATH
```

```
SET OPTIONS-STRING-PTR TO ADDRESS OF CLASSPATH.
MOVE 1 TO NUMBER-OF-OPTIONS.
SET OPTIONS-PTR TO ADDRESS OF VM-OPTIONS.
```

*** Load and initializes the Java VM

```
3b      CALL PROCEDURE "JNI_CreateJavaVM"
          USING JVM-PTR ENV-PTR VM-INIT-ARGS
          RETURNING INTO RC2.
```

```
DISPLAY RC2.
```

```
SET ADDRESS OF INTERFACE-PTR TO ENV-PTR.
SET ADDRESS OF JNI-NATIVE-INTERFACE TO INTERFACE-PTR.
```

*** Callback JNI interface function FIND-CLASS "HelloWorld"

```
STRING FUNCTION UTF8STRING("HelloWorld") DELIMITED BY SIZE
          X"00" DELIMITED BY SIZE
          INTO CLASS-NAME.
```

```
SET CLASS-NAME-PTR TO ADDRESS OF CLASS-NAME.
```

```
CALL FIND-CLASS USING BY VALUE ENV-PTR
          CLASS-NAME-PTR
          RETURNING INTO MY-CLASS-REF.
```

```
DISPLAY MY-CLASS-REF.
```

*** Callback JNI interface function FIND-CLASS "java/lang/String"

```
STRING FUNCTION UTF8STRING("java/lang/String")
          DELIMITED BY SIZE
          X"00" DELIMITED BY SIZE
          INTO CLASS-NAME.
```

```
SET CLASS-NAME-PTR TO ADDRESS OF CLASS-NAME.
```

```
CALL FIND-CLASS USING BY VALUE ENV-PTR
          CLASS-NAME-PTR
          RETURNING INTO STRING-CLASS-REF.
```

```
DISPLAY STRING-CLASS-REF.
```

*** Callback JNI interface function GET-METHOD-ID "<init>"
*** to retrieve constructor method ID

```
STRING FUNCTION UTF8STRING("<init>") DELIMITED BY SIZE
          X"00" DELIMITED BY SIZE
          INTO METHOD-NAME.
```

```
STRING FUNCTION UTF8STRING("()V") DELIMITED BY SIZE
```

```

        X"00" DELIMITED BY SIZE
    INTO SIGNATURE-NAME.

    SET METHOD-NAME-PTR TO ADDRESS OF METHOD-NAME.
    SET SIGNATURE-NAME-PTR TO ADDRESS OF SIGNATURE-NAME.

    CALL GET-METHOD-ID USING BY VALUE ENV-PTR
                                MY-CLASS-REF
                                METHOD-NAME-PTR
                                SIGNATURE-NAME-PTR
    RETURNING INTO INIT-METHOD-ID.

    DISPLAY INIT-METHOD-ID.

*** Callback JNI interface function NEW-OBJECT "HelloWorld"

    CALL NEW-OBJECT USING BY VALUE ENV-PTR
                                MY-CLASS-REF
                                INIT-METHOD-ID
    RETURNING INTO OBJECT-REF.

    DISPLAY OBJECT-REF.

*** Callback JNI interface function GET-STATIC-METHOD-ID "main"

    STRING FUNCTION UTF8STRING("main") DELIMITED BY SIZE
        X"00" DELIMITED BY SIZE
    INTO METHOD-NAME.

    STRING FUNCTION UTF8STRING("[Ljava/lang/String;V")
        DELIMITED BY SIZE
        X"00" DELIMITED BY SIZE
    INTO SIGNATURE-NAME.

    SET METHOD-NAME-PTR TO ADDRESS OF METHOD-NAME.
    SET SIGNATURE-NAME-PTR TO ADDRESS OF SIGNATURE-NAME.

    CALL GET-STATIC-METHOD-ID USING BY VALUE ENV-PTR
                                MY-CLASS-REF
                                METHOD-NAME-PTR
                                SIGNATURE-NAME-PTR
    RETURNING INTO STATIC-METHOD-ID.

    DISPLAY STATIC-METHOD-ID.

*** Callback JNI interface function NEW-OBJECT-ARRAY

    CALL NEW-OBJECT-ARRAY USING BY VALUE ENV-PTR
                                0
                                STRING-CLASS-REF
                                0
    RETURNING INTO ARG-REF.

    DISPLAY ARG-REF.

*** Callback JNI interface function CALL-STATIC-VOID-METHODA

    SET PARM-ARRAY-PTR TO ADDRESS OF PARM-ARRAY.

    INITIALIZE PARM-ARRAY.

    MOVE ARG-REF TO PARM-ARRAY-ELEMENT-VALUE(1).

    CALL CALL-STATIC-VOID-METHODA USING BY VALUE ENV-PTR
                                MY-CLASS-REF
                                STATIC-METHOD-ID
                                PARM-ARRAY-PTR.

```

```

*** Callback JNI interface function GET-METHOD-ID "display"

    STRING FUNCTION UTF8STRING("display") DELIMITED BY SIZE
        X"00" DELIMITED BY SIZE
        INTO METHOD-NAME.

    STRING FUNCTION UTF8STRING("([II)V") DELIMITED BY SIZE
        X"00" DELIMITED BY SIZE
        INTO SIGNATURE-NAME.

    SET METHOD-NAME-PTR TO ADDRESS OF METHOD-NAME.
    SET SIGNATURE-NAME-PTR TO ADDRESS OF SIGNATURE-NAME.

    CALL GET-METHOD-ID USING BY VALUE ENV-PTR
        MY-CLASS-REF
        METHOD-NAME-PTR
        SIGNATURE-NAME-PTR
        RETURNING INTO METHOD-ID.

    DISPLAY METHOD-ID.

*** Callback JNI interface function NEW-INT-ARRAY

    CALL NEW-INT-ARRAY USING BY VALUE ENV-PTR
        10
        RETURNING INTO ARG-REF.

    DISPLAY ARG-REF.

*** Callback JNI interface function CALL-VOID-METHODA

    SET PARM-ARRAY-PTR TO ADDRESS OF PARM-ARRAY.

    INITIALIZE PARM-ARRAY.

    MOVE ARG-REF TO PARM-ARRAY-ELEMENT-VALUE(1).
    MOVE 2      TO PARM-ARRAY-ELEMENT-VALUE(2).

    CALL CALL-VOID-METHODA USING BY VALUE ENV-PTR
        OBJECT-REF
        METHOD-ID
        PARM-ARRAY-PTR.

    GOBACK.

```

Create the COBOL Program: To create a COBOL module, use the CRTBNDCBL command, as shown below.

```

                                Create Bound COBOL Program (CRTBNDCBL)

Type choices, press Enter.

Program . . . . . > HELLOWORLD      Name, *PGMID
Library . . . . .          *CURLIB    Name, *CURLIB
Source file . . . . . > QCBLLSRC     Name
Library . . . . .          *CURLIB    Name, *LIBL, *CURLIB
Source member . . . . . > HELLOWORLD Name, *PGM
Source stream file . . . . .
Output . . . . .          *PRINT      *PRINT, *NONE
Generation severity level . . . 30     0-30
Text 'description' . . . . . *SRCMBRTX

                                Additional Parameters

Replace program . . . . . > *YES      *YES, *NO

                                Bottom
F3=Exit  F4=Prompt  F5=Refresh  F10=Additional parameters  F12=Cancel
F13=How to use this display  F24=More keys

```

Code the Java Program:

```

class HelloWorld {

    public static void main(String[] args) {

        System.out.println("Hello World");

    }

    void display(int[] args, int i) {

        System.out.println("Length of integer array is " + args.length);
        System.out.println("Value of integer variable is " + i);
        System.out.println("Bye World");

    }

}

```

Figure 65. Java Program HelloWorld.java

Compile the Java Program: To compile the Java source program, you can enter the Qshell interpreter (QSH) and issue the following command:

```
javac HelloWorld.java
```

COBOL and Java Data Types

The following table shows the COBOL data type that corresponds to each Java primitive type.

Table 18. Comparison of COBOL and Java Data Types

Java Primitive Type	Description	Java Data Range	COBOL Data Type	COBOL Data Range
boolean	unsigned 8 bits	0 (false) or 1 (true)	PIC 9(4) BINARY	0 to 255
byte	signed 8 bits	-128 to 127	PIC X	-128 to 127

Table 18. Comparison of COBOL and Java Data Types (continued)

Java Primitive Type	Description	Java Data Range	COBOL Data Type	COBOL Data Range
char	unsigned 16 bits	0 ('\u0000') to 65535 ('\uffff')	PIC N USAGE NATIONAL	0 ('\u0000') to 65535 ('\uffff')
short	signed 16 bits	-32768 to 32767	PIC S9(4) BINARY ¹	-32768 to 32767
int	signed 32 bits	-2147483648 to 2147483647	PIC S9(9) BINARY ¹	-2147483648 to 2147483647
long	signed 64 bits	-9223372036854775808 to 9223372036854775807	PIC S9(18) BINARY ¹	-9223372036854775808 to 9223372036854775807
float	32 bits	1.40239846e-45f to 3.40282347e+38f	USAGE COMP-1	0.14012985e-44 to 0.34028235e39
double	64 bits	4.94065645841246544e-324 to 1.79769313486231570e+308	USAGE COMP-2	.11125369292536009e-307 to .17976931348623155e+309
void	n/a	n/a	n/a	n/a

Notes:

- To preserve truncation for short, int, and long primitive types, you must specify NOSTDTRUNC on the PROCESS statement.

The COBOL and Java data ranges are similar.

- For boolean, byte, char, short, and int, the COBOL range is identical to the Java range or larger.
- For float and double, the COBOL data range depends on the machine implementation.
- Void has no COBOL equivalent.

A Java reference type consists of a class, an interface and an array. A reference type is passed as a Java int type argument.

```

01 JBOOLEAN          TYPEDEF PIC 9(4) BINARY.
01 JBYTE             TYPEDEF PIC X.
01 JCHAR             TYPEDEF PIC N USAGE NATIONAL.
01 JSHORT            TYPEDEF PIC S9(4) BINARY.      (and NOSTDTRUNC on PROCESS statement)
01 JINT              TYPEDEF PIC S9(9) BINARY.      (and NOSTDTRUNC on PROCESS statement)
01 JLONG             TYPEDEF PIC S9(18) BINARY.     (and NOSTDTRUNC on PROCESS statement)
01 JFLOAT            TYPEDEF USAGE COMP-1.
01 JDOUBLE           TYPEDEF USAGE COMP-2.

```

Figure 66. Defining Java Data Types

JNI Copy Members for COBOL

These layouts are the COBOL implementation of the JNI interface function table. They can be found in library QSYSINC. For more information about the parameters associated with each JNI function, refer to *Java Native Interface Specification Release 1.1 (Revised May, 1997)*.

- “Member JNI” on page 270
- “Member JDK11INIT” on page 275.

Member JNI

```
*** COBOL copybook for JNI native interface
*** based on Java Native Interface Specification Release 1.1
*** (Revised May, 1997)

01 JNI-NATIVE-INTERFACE.

    05 FILLER                                USAGE PROCEDURE-POINTER.
    05 FILLER                                USAGE PROCEDURE-POINTER.
    05 FILLER                                USAGE PROCEDURE-POINTER.
    05 FILLER                                USAGE PROCEDURE-POINTER.
    05 GET-VERSION                           USAGE PROCEDURE-POINTER.

    05 DEFINE-CLASS                          USAGE PROCEDURE-POINTER.
    05 FIND-CLASS                            USAGE PROCEDURE-POINTER.
    05 FILLER                                USAGE PROCEDURE-POINTER.
    05 FILLER                                USAGE PROCEDURE-POINTER.
    05 FILLER                                USAGE PROCEDURE-POINTER.
    05 GET-SUPERCLASS                         USAGE PROCEDURE-POINTER.
    05 IS-ASSIGNABLE-FROM                    USAGE PROCEDURE-POINTER.
    05 FILLER                                USAGE PROCEDURE-POINTER.

    05 THROW                                  USAGE PROCEDURE-POINTER.
    05 THROW-NEW                             USAGE PROCEDURE-POINTER.
    05 EXCEPTION-OCCURRED                    USAGE PROCEDURE-POINTER.
    05 EXCEPTION-DESCRIBE                    USAGE PROCEDURE-POINTER.
    05 EXCEPTION-CLEAR                       USAGE PROCEDURE-POINTER.
    05 FATAL-ERROR                           USAGE PROCEDURE-POINTER.
    05 FILLER                                USAGE PROCEDURE-POINTER.
    05 FILLER                                USAGE PROCEDURE-POINTER.

    05 NEW-GLOBAL-REF                         USAGE PROCEDURE-POINTER.
    05 DELETE-GLOBAL-REF                     USAGE PROCEDURE-POINTER.
    05 DELETE-LOCAL-REF                       USAGE PROCEDURE-POINTER.
    05 IS-SAME-OBJECT                         USAGE PROCEDURE-POINTER.
    05 FILLER                                USAGE PROCEDURE-POINTER.
    05 FILLER                                USAGE PROCEDURE-POINTER.

    05 ALLOC-OBJECT                           USAGE PROCEDURE-POINTER.
    05 NEW-OBJECT                             USAGE PROCEDURE-POINTER.
    05 NEW-OBJECTV                             USAGE PROCEDURE-POINTER.
    05 NEW-OBJECTA                             USAGE PROCEDURE-POINTER.

    05 GET-OBJECT-CLASS                       USAGE PROCEDURE-POINTER.
    05 IS-INSTANCE-OF                         USAGE PROCEDURE-POINTER.

    05 GET-METHOD-ID                         USAGE PROCEDURE-POINTER.

    05 CALL-OBJECT-METHOD                   USAGE PROCEDURE-POINTER.
    05 CALL-OBJECT-METHODV                   USAGE PROCEDURE-POINTER.
    05 CALL-OBJECT-METHODA                   USAGE PROCEDURE-POINTER.
```

Figure 67. Member JNI (Part 1 of 5)

05 CALL-BOOLEAN-METHOD	USAGE PROCEDURE-POINTER.
05 CALL-BOOLEAN-METHODV	USAGE PROCEDURE-POINTER.
05 CALL-BOOLEAN-METHODA	USAGE PROCEDURE-POINTER.
05 CALL-BYTE-METHOD	USAGE PROCEDURE-POINTER.
05 CALL-BYTE-METHODV	USAGE PROCEDURE-POINTER.
05 CALL-BYTE-METHODA	USAGE PROCEDURE-POINTER.
05 CALL-CHAR-METHOD	USAGE PROCEDURE-POINTER.
05 CALL-CHAR-METHODV	USAGE PROCEDURE-POINTER.
05 CALL-CHAR-METHODA	USAGE PROCEDURE-POINTER.
05 CALL-SHORT-METHOD	USAGE PROCEDURE-POINTER.
05 CALL-SHORT-METHODV	USAGE PROCEDURE-POINTER.
05 CALL-SHORT-METHODA	USAGE PROCEDURE-POINTER.
05 CALL-INT-METHOD	USAGE PROCEDURE-POINTER.
05 CALL-INT-METHODV	USAGE PROCEDURE-POINTER.
05 CALL-INT-METHODA	USAGE PROCEDURE-POINTER.
05 CALL-LONG-METHOD	USAGE PROCEDURE-POINTER.
05 CALL-LONG-METHODV	USAGE PROCEDURE-POINTER.
05 CALL-LONG-METHODA	USAGE PROCEDURE-POINTER.
05 CALL-FLOAT-METHOD	USAGE PROCEDURE-POINTER.
05 CALL-FLOAT-METHODV	USAGE PROCEDURE-POINTER.
05 CALL-FLOAT-METHODA	USAGE PROCEDURE-POINTER.
05 CALL-DOUBLE-METHOD	USAGE PROCEDURE-POINTER.
05 CALL-DOUBLE-METHODV	USAGE PROCEDURE-POINTER.
05 CALL-DOUBLE-METHODA	USAGE PROCEDURE-POINTER.
05 CALL-VOID-METHOD	USAGE PROCEDURE-POINTER.
05 CALL-VOID-METHODV	USAGE PROCEDURE-POINTER.
05 CALL-VOID-METHODA	USAGE PROCEDURE-POINTER.
05 CALL-NONVIRTUAL-OBJECT-METHOD	USAGE PROCEDURE-POINTER.
05 CALL-NONVIRTUAL-OBJECT-METHODV	USAGE PROCEDURE-POINTER.
05 CALL-NONVIRTUAL-OBJECT-METHODA	USAGE PROCEDURE-POINTER.
05 CALL-NONVIRTUAL-BOOLEAN-METHOD	USAGE PROCEDURE-POINTER.
*** Note that the naming of the following 2 procedures deviates	
*** slightly from the others due to the 30 character field	
*** name limitation.	
05 CALL-NONVIRTUAL-BOOLEAN-MTHDV	USAGE PROCEDURE-POINTER.
05 CALL-NONVIRTUAL-BOOLEAN-MTHDA	USAGE PROCEDURE-POINTER.
05 CALL-NONVIRTUAL-BYTE-METHOD	USAGE PROCEDURE-POINTER.
05 CALL-NONVIRTUAL-BYTE-METHODV	USAGE PROCEDURE-POINTER.
05 CALL-NONVIRTUAL-BYTE-METHODA	USAGE PROCEDURE-POINTER.
05 CALL-NONVIRTUAL-CHAR-METHOD	USAGE PROCEDURE-POINTER.
05 CALL-NONVIRTUAL-CHAR-METHODV	USAGE PROCEDURE-POINTER.
05 CALL-NONVIRTUAL-CHAR-METHODA	USAGE PROCEDURE-POINTER.
05 CALL-NONVIRTUAL-SHORT-METHOD	USAGE PROCEDURE-POINTER.
05 CALL-NONVIRTUAL-SHORT-METHODV	USAGE PROCEDURE-POINTER.
05 CALL-NONVIRTUAL-SHORT-METHODA	USAGE PROCEDURE-POINTER.
05 CALL-NONVIRTUAL-INT-METHOD	USAGE PROCEDURE-POINTER.
05 CALL-NONVIRTUAL-INT-METHODV	USAGE PROCEDURE-POINTER.

Figure 67. Member JNI (Part 2 of 5)

05	CALL-NONVIRTUAL-INT-METHODA	USAGE	PROCEDURE-POINTER.
05	CALL-NONVIRTUAL-LONG-METHOD	USAGE	PROCEDURE-POINTER.
05	CALL-NONVIRTUAL-LONG-METHODV	USAGE	PROCEDURE-POINTER.
05	CALL-NONVIRTUAL-LONG-METHODA	USAGE	PROCEDURE-POINTER.
05	CALL-NONVIRTUAL-FLOAT-METHOD	USAGE	PROCEDURE-POINTER.
05	CALL-NONVIRTUAL-FLOAT-METHODV	USAGE	PROCEDURE-POINTER.
05	CALL-NONVIRTUAL-FLOAT-METHODA	USAGE	PROCEDURE-POINTER.
05	CALL-NONVIRTUAL-DOUBLE-METHOD	USAGE	PROCEDURE-POINTER.
05	CALL-NONVIRTUAL-BOOLEAN-MTHDA	USAGE	PROCEDURE-POINTER.
05	CALL-NONVIRTUAL-BYTE-METHOD	USAGE	PROCEDURE-POINTER.
05	CALL-NONVIRTUAL-BYTE-METHODV	USAGE	PROCEDURE-POINTER.
05	CALL-NONVIRTUAL-BYTE-METHODA	USAGE	PROCEDURE-POINTER.
05	CALL-NONVIRTUAL-CHAR-METHOD	USAGE	PROCEDURE-POINTER.
05	CALL-NONVIRTUAL-CHAR-METHODV	USAGE	PROCEDURE-POINTER.
05	CALL-NONVIRTUAL-CHAR-METHODA	USAGE	PROCEDURE-POINTER.
05	CALL-NONVIRTUAL-SHORT-METHOD	USAGE	PROCEDURE-POINTER.
05	CALL-NONVIRTUAL-SHORT-METHODV	USAGE	PROCEDURE-POINTER.
05	CALL-NONVIRTUAL-SHORT-METHODA	USAGE	PROCEDURE-POINTER.
05	CALL-NONVIRTUAL-INT-METHOD	USAGE	PROCEDURE-POINTER.
05	CALL-NONVIRTUAL-INT-METHODV	USAGE	PROCEDURE-POINTER.
05	CALL-NONVIRTUAL-INT-METHODA	USAGE	PROCEDURE-POINTER.
05	CALL-NONVIRTUAL-LONG-METHOD	USAGE	PROCEDURE-POINTER.
05	CALL-NONVIRTUAL-LONG-METHODV	USAGE	PROCEDURE-POINTER.
05	CALL-NONVIRTUAL-LONG-METHODA	USAGE	PROCEDURE-POINTER.
05	CALL-NONVIRTUAL-FLOAT-METHOD	USAGE	PROCEDURE-POINTER.
05	CALL-NONVIRTUAL-FLOAT-METHODV	USAGE	PROCEDURE-POINTER.
05	CALL-NONVIRTUAL-FLOAT-METHODA	USAGE	PROCEDURE-POINTER.
05	CALL-NONVIRTUAL-DOUBLE-METHOD	USAGE	PROCEDURE-POINTER.
05	SET-CHAR-FIELD	USAGE	PROCEDURE-POINTER.
05	SET-SHORT-FIELD	USAGE	PROCEDURE-POINTER.
05	SET-INT-FIELD	USAGE	PROCEDURE-POINTER.
05	SET-LONG-FIELD	USAGE	PROCEDURE-POINTER.
05	SET-FLOAT-FIELD	USAGE	PROCEDURE-POINTER.
05	SET-DOUBLE-FIELD	USAGE	PROCEDURE-POINTER.
05	GET-STATIC-METHOD-ID	USAGE	PROCEDURE-POINTER.
05	CALL-STATIC-OBJECT-METHOD	USAGE	PROCEDURE-POINTER.
05	CALL-STATIC-OBJECT-METHODV	USAGE	PROCEDURE-POINTER.
05	CALL-STATIC-OBJECT-METHODA	USAGE	PROCEDURE-POINTER.
05	CALL-STATIC-BOOLEAN-METHOD	USAGE	PROCEDURE-POINTER.
05	CALL-STATIC-BOOLEAN-METHODV	USAGE	PROCEDURE-POINTER.
05	CALL-STATIC-BOOLEAN-METHODA	USAGE	PROCEDURE-POINTER.
05	CALL-STATIC-BYTE-METHOD	USAGE	PROCEDURE-POINTER.
05	CALL-STATIC-BYTE-METHODV	USAGE	PROCEDURE-POINTER.
05	CALL-STATIC-BYTE-METHODA	USAGE	PROCEDURE-POINTER.
05	CALL-STATIC-CHAR-METHOD	USAGE	PROCEDURE-POINTER.
05	CALL-STATIC-CHAR-METHODV	USAGE	PROCEDURE-POINTER.
05	CALL-STATIC-CHAR-METHODA	USAGE	PROCEDURE-POINTER.
05	CALL-STATIC-SHORT-METHOD	USAGE	PROCEDURE-POINTER.

Figure 67. Member JNI (Part 3 of 5)

05 CALL-STATIC-SHORT-METHODV	USAGE PROCEDURE-POINTER.
05 CALL-STATIC-SHORT-METHODA	USAGE PROCEDURE-POINTER.
05 CALL-STATIC-INT-METHOD	USAGE PROCEDURE-POINTER.
05 CALL-STATIC-INT-METHODV	USAGE PROCEDURE-POINTER.
05 CALL-STATIC-INT-METHODA	USAGE PROCEDURE-POINTER.
05 CALL-STATIC-LONG-METHOD	USAGE PROCEDURE-POINTER.
05 CALL-STATIC-LONG-METHODV	USAGE PROCEDURE-POINTER.
05 CALL-STATIC-LONG-METHODA	USAGE PROCEDURE-POINTER.
05 CALL-STATIC-FLOAT-METHOD	USAGE PROCEDURE-POINTER.
05 CALL-STATIC-FLOAT-METHODV	USAGE PROCEDURE-POINTER.
05 CALL-STATIC-FLOAT-METHODA	USAGE PROCEDURE-POINTER.
05 CALL-STATIC-DOUBLE-METHOD	USAGE PROCEDURE-POINTER.
05 CALL-STATIC-DOUBLE-METHODV	USAGE PROCEDURE-POINTER.
05 CALL-STATIC-DOUBLE-METHODA	USAGE PROCEDURE-POINTER.
05 CALL-STATIC-VOID-METHOD	USAGE PROCEDURE-POINTER.
05 CALL-STATIC-VOID-METHODV	USAGE PROCEDURE-POINTER.
05 CALL-STATIC-VOID-METHODA	USAGE PROCEDURE-POINTER.
05 GET-STATIC-FILED-ID	USAGE PROCEDURE-POINTER.
05 GET-STATIC-OBJECT-FIELD	USAGE PROCEDURE-POINTER.
05 GET-STATIC-OBJECT-BOOLEAN-FIELD	USAGE PROCEDURE-POINTER.
05 GET-STATIC-OBJECT-BYTE-FIELD	USAGE PROCEDURE-POINTER.
05 GET-STATIC-OBJECT-CHAR-FIELD	USAGE PROCEDURE-POINTER.
05 GET-STATIC-OBJECT-SHORT-FIELD	USAGE PROCEDURE-POINTER.
05 GET-STATIC-OBJECT-INT-FIELD	USAGE PROCEDURE-POINTER.
05 GET-STATIC-OBJECT-LONG-FIELD	USAGE PROCEDURE-POINTER.
05 GET-STATIC-OBJECT-FLOAT-FIELD	USAGE PROCEDURE-POINTER.
05 GET-STATIC-OBJECT-DOUBLE-FIELD	USAGE PROCEDURE-POINTER.
05 SET-STATIC-OBJECT-FIELD	USAGE PROCEDURE-POINTER.
05 SET-STATIC-OBJECT-BOOLEAN-FIELD	USAGE PROCEDURE-POINTER.
05 SET-STATIC-OBJECT-BYTE-FIELD	USAGE PROCEDURE-POINTER.
05 SET-STATIC-OBJECT-CHAR-FIELD	USAGE PROCEDURE-POINTER.
05 SET-STATIC-OBJECT-SHORT-FIELD	USAGE PROCEDURE-POINTER.
05 SET-STATIC-OBJECT-INT-FIELD	USAGE PROCEDURE-POINTER.
05 SET-STATIC-OBJECT-LONG-FIELD	USAGE PROCEDURE-POINTER.
05 SET-STATIC-OBJECT-FLOAT-FIELD	USAGE PROCEDURE-POINTER.
05 SET-STATIC-OBJECT-DOUBLE-FIELD	USAGE PROCEDURE-POINTER.
05 NEW-STRING	USAGE PROCEDURE-POINTER.
05 GET-STRING-LENGTH	USAGE PROCEDURE-POINTER.
05 GET-STRING-CHARS	USAGE PROCEDURE-POINTER.
05 RELEASE-STRING-CHARS	USAGE PROCEDURE-POINTER.
05 NEW-STRING-UTF	USAGE PROCEDURE-POINTER.
05 GET-STRING-UTF-LENGTH	USAGE PROCEDURE-POINTER.
05 GET-STRING-UTF-CHARS	USAGE PROCEDURE-POINTER.
05 RELEASE-STRING-UTF-CHARS	USAGE PROCEDURE-POINTER.
05 GET-ARRAY-LENGTH	USAGE PROCEDURE-POINTER.

Figure 67. Member JNI (Part 4 of 5)

05 NEW-OBJECT-ARRAY	USAGE PROCEDURE-POINTER.
05 GET-OBJECT-ARRAY-ELEMENT	USAGE PROCEDURE-POINTER.
05 SET-OBJECT-ARRAY-ELEMENT	USAGE PROCEDURE-POINTER.
05 NEW-BOOLEAN-ARRAY	USAGE PROCEDURE-POINTER.
05 NEW-BYTE-ARRAY	USAGE PROCEDURE-POINTER.
05 NEW-CHAR-ARRAY	USAGE PROCEDURE-POINTER.
05 NEW-SHORT-ARRAY	USAGE PROCEDURE-POINTER.
05 NEW-INT-ARRAY	USAGE PROCEDURE-POINTER.
05 NEW-LONG-ARRAY	USAGE PROCEDURE-POINTER.
05 NEW-FLOAT-ARRAY	USAGE PROCEDURE-POINTER.
05 NEW-DOUBLE-ARRAY	USAGE PROCEDURE-POINTER.
05 GET-BOOLEAN-ARRAY-ELEMENTS	USAGE PROCEDURE-POINTER.
05 GET-BYTE-ARRAY-ELEMENTS	USAGE PROCEDURE-POINTER.
05 GET-CHAR-ARRAY-ELEMENTS	USAGE PROCEDURE-POINTER.
05 GET-SHORT-ARRAY-ELEMENTS	USAGE PROCEDURE-POINTER.
05 GET-INT-ARRAY-ELEMENTS	USAGE PROCEDURE-POINTER.
05 GET-LONG-ARRAY-ELEMENTS	USAGE PROCEDURE-POINTER.
05 GET-FLOAT-ARRAY-ELEMENTS	USAGE PROCEDURE-POINTER.
05 GET-DOUBLE-ARRAY-ELEMENTS	USAGE PROCEDURE-POINTER.
05 RELEASE-BOOLEAN-ARRAY-ELEMENTS	USAGE PROCEDURE-POINTER.
05 RELEASE-BYTE-ARRAY-ELEMENTS	USAGE PROCEDURE-POINTER.
05 RELEASE-CHAR-ARRAY-ELEMENTS	USAGE PROCEDURE-POINTER.
05 RELEASE-SHORT-ARRAY-ELEMENTS	USAGE PROCEDURE-POINTER.
05 RELEASE-INT-ARRAY-ELEMENTS	USAGE PROCEDURE-POINTER.
05 RELEASE-LONG-ARRAY-ELEMENTS	USAGE PROCEDURE-POINTER.
05 RELEASE-FLOAT-ARRAY-ELEMENTS	USAGE PROCEDURE-POINTER.
05 RELEASE-DOUBLE-ARRAY-ELEMENTS	USAGE PROCEDURE-POINTER.
05 GET-BOOLEAN-ARRAY-REGION	USAGE PROCEDURE-POINTER.
05 GET-BYTE-ARRAY-REGION	USAGE PROCEDURE-POINTER.
05 GET-CHAR-ARRAY-REGION	USAGE PROCEDURE-POINTER.
05 GET-SHORT-ARRAY-REGION	USAGE PROCEDURE-POINTER.
05 GET-INT-ARRAY-REGION	USAGE PROCEDURE-POINTER.
05 GET-LONG-ARRAY-REGION	USAGE PROCEDURE-POINTER.
05 GET-FLOAT-ARRAY-REGION	USAGE PROCEDURE-POINTER.
05 GET-DOUBLE-ARRAY-REGION	USAGE PROCEDURE-POINTER.
05 SET-BOOLEAN-ARRAY-REGION	USAGE PROCEDURE-POINTER.
05 SET-BYTE-ARRAY-REGION	USAGE PROCEDURE-POINTER.
05 SET-CHAR-ARRAY-REGION	USAGE PROCEDURE-POINTER.
05 SET-SHORT-ARRAY-REGION	USAGE PROCEDURE-POINTER.
05 SET-INT-ARRAY-REGION	USAGE PROCEDURE-POINTER.
05 SET-LONG-ARRAY-REGION	USAGE PROCEDURE-POINTER.
05 SET-FLOAT-ARRAY-REGION	USAGE PROCEDURE-POINTER.
05 SET-DOUBLE-ARRAY-REGION	USAGE PROCEDURE-POINTER.
05 REGISTER-NATIVES	USAGE PROCEDURE-POINTER.
05 UNREGISTER-NATIVES	USAGE PROCEDURE-POINTER.
05 MONITOR-ENTER	USAGE PROCEDURE-POINTER.
05 MONITOR-EXIT	USAGE PROCEDURE-POINTER.
05 GET-JAVA-VM	USAGE PROCEDURE-POINTER.

Figure 67. Member JNI (Part 5 of 5)

Member JDK11INIT

```
*** COBOL copybook for JDK 1.1 VM initialization arguments
*** based on Java Native Interface Specification Release 1.1
*** (Revised May, 1997)

01 VM-INIT-ARGS.
   05 VERSION                PIC S9(9) BINARY VALUE 65537.
   05 FILLER                  PIC S9(9) BINARY.
   05 FILLER                  PIC S9(9) BINARY.
   05 FILLER                  PIC S9(9) BINARY.
   05 PROPERTIES              USAGE PROCEDURE-POINTER.
   05 CHECK-SOURCE           PIC S9(9) BINARY.
   05 NATIVE-STACK-SIZE     PIC S9(9) BINARY.
   05 JAVA-STACK-SIZE      PIC S9(9) BINARY.
   05 MIN-HEAP-SIZE        PIC S9(9) BINARY.
   05 MAX-HEAP-SIZE        PIC S9(9) BINARY.
   05 VERIFY-MODE          PIC S9(9) BINARY.
   05 FILLER                PIC S9(9) BINARY.
   05 FILLER                PIC S9(9) BINARY.
   05 CLASSPATH             USAGE POINTER.
   05 MESSAGE-HOOK          USAGE PROCEDURE-POINTER.
   05 EXIT-HOOK             USAGE PROCEDURE-POINTER.
   05 ABORT-HOOK           USAGE PROCEDURE-POINTER.
   05 ENABLE-CLASSIC-GC    PIC S9(9) BINARY.
   05 ENABLE-VERBOSE-GC   PIC S9(9) BINARY.
   05 DISABLE-ASYNC-GC    PIC S9(9) BINARY.
   05 FILLER                PIC S9(9) BINARY.
   05 FILLER                PIC S9(9) BINARY.
   05 FILLER                PIC S9(9) BINARY.
```

Figure 68. Member JDK11INIT

Chapter 11. Processing XML Input

You can process XML documents from your ILE COBOL program by using the XML PARSE statement. The XML PARSE statement is the COBOL language interface to the high-speed XML parser, which is part of the COBOL run time. Processing an XML document involves control being passed to and received from the XML parser. You start this exchange of control with the XML PARSE statement, which specifies a processing procedure that receives control from the XML parser to handle the parser events. You use special registers in your processing procedure to exchange information with the parser.

Use these COBOL facilities to process XML documents:

- XML PARSE statement to begin the XML parse and to identify the document and your processing procedure
- Processing procedure to control the parse: receive and process the XML events and associated document fragments and optionally handle exceptions
- Special registers to receive and pass information:
 - XML-CODE to determine the status of XML parsing
 - XML-EVENT to receive the name of each XML event
 - XML-TEXT to receive XML document fragments from an alphanumeric document
 - XML-NTEXT to receive XML document fragments from a national document

RELATED CONCEPTS

“XML parser in COBOL”

RELATED TASKS

“Accessing XML documents” on page 279

“Parsing XML documents” on page 279

“Processing XML events” on page 280

“Handling errors in XML documents” on page 294

“Understanding XML document encoding” on page 293

RELATED REFERENCE

Appendix F, “XML reference material,” on page 635

XML specification (www.w3c.org/XML/)

XML parser in COBOL

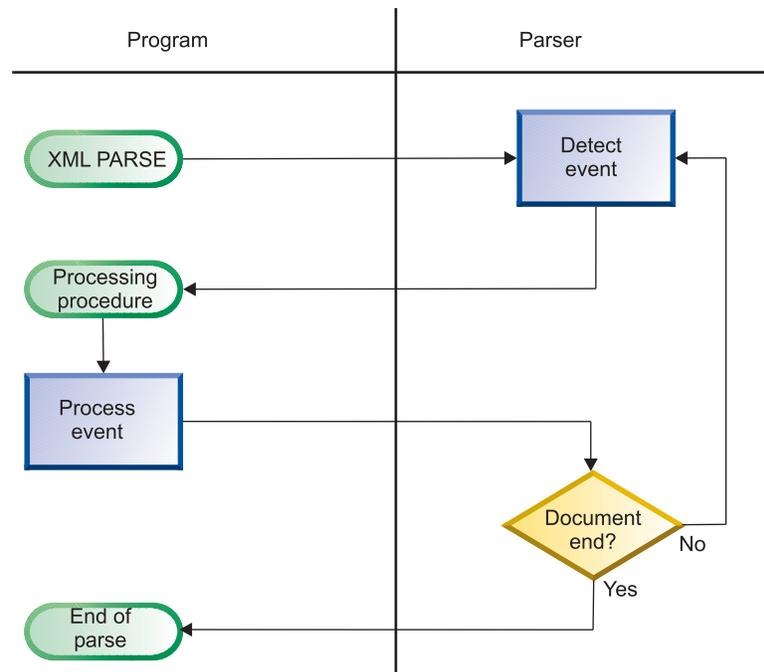
ILE COBOL provides an event-based interface that enables you to parse XML documents and transform them to COBOL data structures. The XML parser finds fragments (associated with XML events) within the document, and your processing procedure acts on these fragments. You code your procedure to handle each XML event. Throughout this operation, control passes back and forth between the parser and your procedure.

You start this exchange with the parser by using the XML PARSE statement, in which you designate your processing procedure. Execution of this XML PARSE statement begins the parse and establishes your processing procedure with the parser. Each execution of your procedure causes the XML parser to continue analyzing the XML document and report the next event, passing back to your procedure the fragment

that it finds, such as the start of a new element. You can also specify on the XML PARSE statement two imperative statements to which you want control to be passed at the end of the parse: one when a normal end occurs and one when an exception condition exists.

This figure gives a high-level overview of the basic exchange of control between the parser and your program:

XML parsing flow overview



Normally, parsing continues until the entire XML document has been parsed.

When the XML parser parses XML documents, it checks them for most aspects of well formedness as defined in the XML specification. A document is well formed if it adheres to the XML syntax and follows some additional rules such as proper use of end tags and uniqueness of attribute names.

RELATED TASKS

- “Accessing XML documents” on page 279
- “Parsing XML documents” on page 279
- “Writing procedures to process XML” on page 285
- “Handling errors in XML documents” on page 294
- “Understanding XML document encoding” on page 293

RELATED REFERENCE

- XML specification* (www.w3c.org/XML/)
- “XML conformance” on page 643

Accessing XML documents

Before you can parse an XML document with an XML PARSE statement, you must make the document available to your program. The most likely method of acquiring the document is by retrieval from an MQSeries message, a CICS transient queue or communication area.

If the XML document that you want to parse is held in a file, use ordinary COBOL facilities to place the document into a data item in your program:

- A FILE-CONTROL entry to define the file to your program
- The OPEN statement to open the file
- The READ statement to read all the records from the file into an alphanumeric or national data item that is defined in the WORKING-STORAGE SECTION or LOCAL-STORAGE SECTION of your program
- Optionally the STRING statement to string all of the separate records together into one continuous stream, to remove extraneous blanks, and to handle variable-length records

Alternatively, you can copy the XML document to an IFS stream file, and use format 2 of the XML PARSE statement to access and parse the document.

```
# RELATED TASKS
# CICS for i5/OS Application Programming Guide. See the i5/OS Information
# Center http://www.ibm.com/systems/infocenter/ .
```

Parsing XML documents

To parse XML documents, when the XML document is in a data item, use the XML PARSE statement, as in the following example:

```
# XML PARSE XMLDOCUMENT
# PROCESSING PROCEDURE XMLEVENT-HANDLER
# ON EXCEPTION
# DISPLAY 'XML document error ' XML-CODE
# STOP RUN
# NOT ON EXCEPTION
# DISPLAY 'XML document was successfully parsed.'
# END-XML
```

```
# In the XML PARSE statement you first identify the data item (XMLDOCUMENT in the
# example) that contains the XML document character stream. In the DATA DIVISION,
# you can declare the identifier as an alphanumeric data item or as a national data
# item. If it is alphanumeric, its contents must be encoded with one of the supported
# single-byte EBCDIC or ASCII character sets. If it is a national data item, its
# contents must be encoded with Unicode UCS-2 CCSID specified in the National
# CCSID compiler option or in the NTLCCSID PROCESS option. Alphanumeric XML
# documents that do not contain an encoding declaration are parsed with the CCSID
# of the COBOL source member, or if COBOL source is in an IFS stream file, the
# CCSID of the stream file is used.
```

Next you specify the name of the procedure (XMLEVENT-HANDLER in the example) that is to handle the XML events from the document.

In addition, you can specify either or both of the following imperative statements to receive control at the end of the parse:

- ON EXCEPTION, to receive control when an unhandled exception occurs

- NOT ON EXCEPTION, to receive control otherwise

You can end the XML PARSE statement with END-XML. Use this scope terminator to nest your XML PARSE statement in a conditional statement or in another XML PARSE statement.

The exchange of control between the XML parser and your processing procedure continues until one of the following occurs:

- The entire XML document has been parsed, indicated by the END-OF-DOCUMENT event.
- The parser detects an error in the document and signals an EXCEPTION event. Your processing procedure does not reset the special register XML-CODE to zero before returning to the parser.
- You terminate the parsing process deliberately by setting the special register XML-CODE to -1 before returning to the parser.

RELATED TASKS

“Understanding XML document encoding” on page 293

RELATED REFERENCES

XML PARSE statement (*ILE COBOL Language Reference*)

Control flow (*ILE COBOL Language Reference*)

Processing XML events

Use the XML-EVENT special register to determine the event that the parser passes to your processing procedure. XML-EVENT contains an event name such as 'START-OF-ELEMENT'. The parser passes the content for the event in special register XML-TEXT or XML-NTEXT, depending on the type of the XML identifier in your XML PARSE statement.

The events are shown in basically the order that they would occur for this sample XML document. The text shown under “Sample XML text” comes from this sample; exact text is shown between these delimiters: <<>>:

```
<?xml version="1.0" encoding="ibm-1140" standalone="yes" ?>
<!--This document is just an example-->
<sandwich>
  <bread type="baker&apos;s best" />
  <?spread please use real mayonnaise ?>
  <meat>Ham &amp; turkey</meat>
  <filling>Cheese, lettuce, tomato, etc.</filling>
  <![CDATA[We should add a <relish> element in future!]]>
</sandwich>junk
```

START-OF-DOCUMENT

Description

Occurs once, at the beginning of parsing the document. XML text is the entire document, including any line-control characters, such as LF (Line Feed) or NL (New Line).

Sample XML text

The text for this sample is 336 characters in length.

VERSION-INFORMATION

Description

Occurs within the optional XML declaration for the version

information. XML text contains the version value. An *XML declaration* is XML text that specifies the version of XML being used and the encoding of the document.

Sample XML text

```
<<1.0>>
```

ENCODING-DECLARATION

Description

Occurs within the XML declaration for the optional encoding declaration. XML text contains the encoding value.

Sample XML text

```
<<ibm-1140>>
```

STANDALONE-DECLARATION

Description

Occurs within the XML declaration for the optional standalone declaration. XML text contains the standalone value.

Sample XML text

```
<<yes>>
```

DOCUMENT-TYPE-DECLARATION

Description

Occurs when the parser finds a document type declaration (DTD). Document type declarations begin with the character sequence '`<!DOCTYPE`' and end with a '`>`' character, with some fairly complicated grammar rules describing the content in between. (See the *XML specification* for details.) For this event, XML text contains the entire declaration, including the opening and closing character sequences. This is the only event where XML text includes the delimiters.

Sample XML text

The sample does not have a document type declaration.

COMMENT

Description

Occurs for any comments in the XML document. XML text contains the data between the opening and closing comment delimiters, '`<!--`' and '`-->`', respectively.

Sample XML text

```
<<This document is just an example>>
```

START-OF-ELEMENT

Description

Occurs once for each element start tag or empty element tag. XML text is set to the element name.

Sample XML text

In the order that they occur as START-OF-ELEMENT events:

1. <<sandwich>>
2. <<bread>>
3. <<meat>>
4. <<filling>>

ATTRIBUTE-NAME

Description

Occurs for each attribute in an element start tag or empty element tag, after recognizing a valid name. XML text contains the attribute name.

Sample XML text

<<type>>

ATTRIBUTE-CHARACTERS

Description

Occurs for each fragment of an attribute value. XML text contains the fragment. An attribute value normally consists of a single string only, even if it is split across lines. The attribute value might consist of multiple events, however.

Sample XML text

In the order that they occur as ATTRIBUTE-CHARACTERS events:

1. <<baker>>
2. <<s best>>

Notice that the value of the 'type' attribute in the sample consists of three fragments: the string 'baker', the single character '"', and the string 's best'. The single character '"' fragment is passed separately as an ATTRIBUTE-CHARACTER event.

ATTRIBUTE-CHARACTER

Description

Occurs in attribute values for the predefined entity references '&', ''', '>', '<', and '"'. See the *XML specification* for details of predefined entities.

Sample XML text

<<'>>

ATTRIBUTE-NATIONAL-CHARACTER

Description

Occurs in attribute values for numeric character references (Unicode code points or "scalar values") of the form '&#dd.;' or '&#hh.;', where 'd' and 'h' represent decimal and hexadecimal digits, respectively.

Sample XML text

The sample does not contain a numeric character reference.

PROCESSING-INSTRUCTION-TARGET

Description

Occurs when the parser recognizes the name following the processing instruction (PI) opening character sequence, '<?'. PIs allow XML documents to contain special instructions for applications.

Sample XML text

```
<<spread>>
```

PROCESSING-INSTRUCTION-DATA

Description

Occurs for the data following the PI target, up to but not including the PI closing character sequence, '?>'. XML text contains the PI data, which includes trailing, but not leading white space characters.

Sample XML text

```
<<please use real mayonnaise >>
```

CONTENT-CHARACTERS

Description

This event represents the principal part of an XML document: the character data between element start and end tags. XML text contains this data, which usually consists of a single string only, even if it is split across lines. If the content of an element includes any references or other elements, the complete content might consist of several events. The parser also uses the CONTENT-CHARACTERS event to pass the text of CDATA sections to your program.

Sample XML text

In the order that they occur as CONTENT-CHARACTERS events:

1. <<Ham >>
2. << turkey>>
3. <<Cheese, lettuce, tomato, etc.>>
4. <<We should add a <relish> element in future!>>

Notice that the content of the 'meat' element in the sample consists of the string 'Ham ', the character '&' and the string ' turkey'. The single character '&' fragment is passed separately as a CONTENT-CHARACTER event. Also notice the trailing and leading spaces, respectively, in these two string fragments.

CONTENT-CHARACTER

Description

Occurs in element content for the predefined entity references '&'; '''; '>'; '<'; and '"'. See the *XML specification* for details of predefined entities.

Sample XML text

<&>

CONTENT-NATIONAL-CHARACTER

Description

Occurs in element content for numeric character references (Unicode code points or “scalar values”) of the form '&#dd..;' or '&#hh..;', where 'd' and 'h' represent decimal and hexadecimal digits, respectively.

Sample XML text

The sample does not contain a numeric character reference.

END-OF-ELEMENT

Description

Occurs once for each element end tag or empty element tag when the parser recognizes the closing angle bracket of the tag. XML text contains the element name.

Sample XML text

In the order that they occur as END-OF-ELEMENT events:

1. <<bread>>
2. <<meat>>
3. <<filling>>
4. <<sandwich>>

START-OF-CDATA-SECTION

Description

Occurs at the start of a CDATA section. CDATA sections begin with the string '<![CDATA[' and end with the string ']]>'. Such sections are used to “escape” blocks of text containing characters that would otherwise be recognized as XML markup. XML text always contains the opening character sequence '<![CDATA['. The parser passes the content of a CDATA section between these delimiters as a single CONTENT-CHARACTERS event.

Sample XML text

<<<![CDATA[>>

END-OF-CDATA-SECTION

Description

Occurs when the parser recognizes the end of a CDATA section.

Sample XML text

<<]]>>>

UNKNOWN-REFERENCE-IN-ATTRIBUTE

Description

Occurs within attribute values for entity references other than the five predefined entity references, as shown for ATTRIBUTE-CHARACTER above.

Sample XML text

The sample does not have any unknown entity references.

UNKNOWN-REFERENCE-IN-CONTENT**Description**

Occurs within element content for entity references other than the predefined entity references, as shown for CONTENT-CHARACTER above.

Sample XML text

The sample does not have any unknown entity references.

END-OF-DOCUMENT**Description**

Occurs when document parsing has completed

Sample XML text

XML text is empty for the END-OF-DOCUMENT event.

EXCEPTION**Description**

Occurs when an error in processing the XML document is detected. For encoding conflict exceptions, which are signaled before parsing begins, XML-TEXT is either zero-length or contains just the encoding declaration value from the document.

Sample XML text

The part of the document that was parsed up to and including the point where the exception (the superfluous 'junk' after the <sandwich> element end tag) was detected.

RELATED REFERENCE

XML-EVENT (*ILE COBOL Language Reference*)

4.6 Predefined entities (*XML specification* at www.w3.org/TR/REC-xml#sec-predefined-ent)

2.8 Prolog and document type declaration (*XML specification* at www.w3.org/TR/REC-xml#sec-prolog-dtd)

Writing procedures to process XML

In your processing procedure, code the statements to handle XML events.

For each event that the parser encounters, it passes information to your processing procedure in several special registers, as shown in the following table. Use these registers to populate your data structures and to control your processing.

Table 19. Special registers used by the XML parser

Special register	Contents	Implicit definition and usage
XML-EVENT ¹	The name of the XML event	PICTURE X(30) USAGE DISPLAY VALUE SPACE
XML-CODE	An exception code or zero for each XML event	PICTURE S9(9) USAGE BINARY VALUE ZERO
XML-TEXT ¹	Text (corresponding to the event that the parser encountered) from the XML document if you specify an alphanumeric data item for the XML PARSE identifier	Variable-length alphanumeric data item; 16,000,000 byte size limit
XML-NTEXT ¹	Text (corresponding to the event that the parser encountered) from the XML document if you specify a national data item for the XML PARSE identifier	Variable-length national data item; 16,000,000 byte size limit
1. You cannot use this special register as a receiving data item.		

When used in nested programs, these special registers are implicitly defined as GLOBAL in the outermost program.

Understanding the contents of XML-CODE

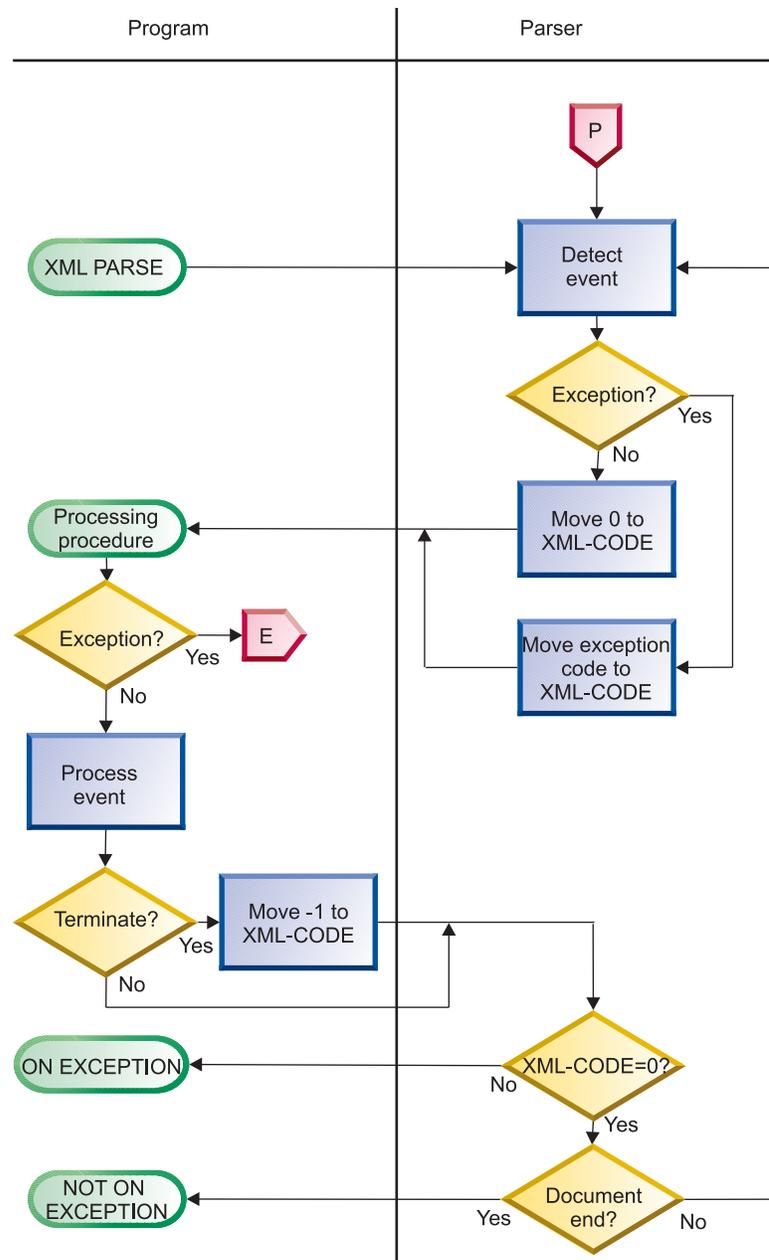
When the parser returns control to your XML PARSE statement, XML-CODE contains the most recent value set by the parser or your processing procedure.

For all events except the EXCEPTION event, the value of the XML-CODE special register is zero. If you set the XML-CODE special register to -1 before you return control to the XML parser for an event other than EXCEPTION, processing stops with a user-initiated exception indicated by the returned XML-CODE value of -1. The result of changing the XML-CODE to any other nonzero value before returning from any event is undefined.

For the EXCEPTION event, special register XML-CODE contains the exception code.

The following figure shows the flow of control between the parser and your processing procedure and how XML-CODE is used to pass information between the two. The off-page connectors, such as , connect the multiple charts in this chapter. In particular,  in the following figure connects to the chart Control flow for XML exceptions, and  connects from XML CCSID exception flow control.

Control flow between XML parser and program, showing XML-CODE usage



Using XML-TEXT and XML-NTEXT

The special registers XML-TEXT and XML-NTEXT are mutually exclusive. The type of XML identifier that you specify determines which special register is set, except for the ATTRIBUTE-NATIONAL-CHARACTER and CONTENT-NATIONAL-CHARACTER events. For these events, XML-NTEXT is set regardless of the data item that you specify for the XML PARSE identifier.

When the parser sets XML-TEXT, XML-NTEXT is undefined (length of 0). When the parser sets XML-NTEXT, XML-TEXT is undefined (length of 0).

To determine how many national characters XML-NTEXT contains, use the LENGTH function. LENGTH OF XML-NTEXT contains the number of bytes, rather than characters, used by XML-NTEXT.

Transforming XML text to COBOL data items

Because XML data is neither fixed length nor fixed format, you need to use special techniques when you move the XML data to COBOL data items.

For alphanumeric items, decide whether the XML data should go at the left (default) end of your COBOL item, or at the right end. If it should go at the right end, specify the JUSTIFIED RIGHT clause in the declaration of the COBOL item.

Give special consideration to numeric XML values, particularly “decorated” monetary values such as '\$1,234.00' and '\$1234'. These mean the same thing in XML but have completely different declarations as COBOL sending fields. Use this technique:

- For simplicity and vastly increased flexibility, use the following for alphanumeric XML data:
 - Function NUMVAL to extract and decode simple numeric values from XML data representing plain numbers
 - Function NUMVAL-C to extract and decode numeric values from XML data representing monetary quantities

Note, however, that use of these functions is at the expense of performance.

Restriction on your processing procedure

Your processing procedure must not directly execute an XML PARSE statement. However, if your processing procedure passes control to an outermost program using a CALL statement, the target method or program can execute the same or a different XML PARSE statement. You can also execute the same XML statement or different XML statements simultaneously from a program that is executing on multiple threads.

Ending your processing procedure

The compiler inserts a return mechanism after the last statement in your processing procedure. You can code a STOP RUN statement in your processing procedure to terminate the run unit. However, the GOBACK or EXIT PROGRAM statements do not return control to the parser. Using either statement in your processing procedure results in a severe error.

“Examples: parsing XML”

RELATED REFERENCES

“XML exceptions that allow continuation” on page 635

“XML exceptions that do not allow continuation” on page 639

XML-CODE (*ILE COBOL Language Reference*)

XML-EVENT (*ILE COBOL Language Reference*)

XML-NTEXT (*ILE COBOL Language Reference*)

XML-TEXT (*ILE COBOL Language Reference*)

Examples: parsing XML

This example shows the basic organization of an XML PARSE statement and a processing procedure, where the XML document is in a COBOL data item. The XML document is given in the source so that you can follow the flow of the parse. The output of the program is given below. Compare the document to the output of the program to follow the interaction of the parser and the processing procedure and to match events to document fragments.

Example: parsing XML from a data item

```

Process APOST
Identification division.
  Program-id. xmlsaml1.

```

```

Data division.
  Working-storage section.

```

```

*****
* XML document, encoded as initial values of data items. *
*****

```

```

1 xml-document.
2 pic x(39) value '<?xml version="1.0" encoding="ibm-37"'.
2 pic x(19) value ' standalone="yes"?>'.
2 pic x(39) value '<!--This document is just an example-->'.
2 pic x(10) value '<sandwich>'.
2 pic x(35) value ' <bread type="baker&apos;s best"/>'.
2 pic x(41) value ' <?spread please use real mayonnaise ?>'.
2 pic x(31) value ' <meat>Ham &amp; turkey</meat>'.
2 pic x(40) value ' <filling>Cheese, lettuce, tomato, etc.'.
2 pic x(10) value '</filling>'.
2 pic x(35) value ' <![CDATA[We should add a <relish>'.
2 pic x(22) value ' element in future!]]>'.
2 pic x(31) value ' <listprice>$4.99 </listprice>'.
2 pic x(27) value ' <discount>0.10</discount>'.
2 pic x(11) value '</sandwich>'.
1 xml-document-length computational pic 999.

```

```

*****
* Sample data definitions for processing numeric XML content. *
*****

```

```

1 current-element pic x(30).
1 list-price computational pic 9v99 value 0.
1 discount computational pic 9v99 value 0.
1 display-price pic $$9.99.

```

```

Procedure division.
  mainline section.

```

```

XML PARSE xml-document PROCESSING PROCEDURE xml-handler
  ON EXCEPTION
    display 'XML document error ' XML-CODE
  NOT ON EXCEPTION
    display 'XML document successfully parsed'
END-XML

```

```

*****
* Process the transformed content and calculate promo price. *
*****

```

```

display ' '
display '-----+++++***** Using information from XML '
  '*****+++++-----'
display ' '
move list-price to display-price
display ' Sandwich list price: ' display-price
compute display-price = list-price * (1 - discount)
display ' Promotional price: ' display-price
display ' Get one today!'

goback.

```

```

xml-handler section.
  evaluate XML-EVENT
* ==> Order XML events most frequent first
  when 'START-OF-ELEMENT'
    display 'Start element tag: <' XML-TEXT '>'
    move XML-TEXT to current-element
  when 'CONTENT-CHARACTERS'
    display 'Content characters: <' XML-TEXT '>'

```

```

* ==> Transform XML content to operational COBOL data item...
      evaluate current-element
        when 'listprice'
* ==> Using function NUMVAL-C...
          compute list-price = function numval-c(XML-TEXT)
          when 'discount'
            compute discount = function numval-c(XML-TEXT)
          end-evaluate
        when 'END-OF-ELEMENT'
          display 'End element tag: <' XML-TEXT '>'
          move spaces to current-element
        when 'START-OF-DOCUMENT'
          compute xml-document-length = function length(XML-TEXT)
          display 'Start of document: length=' xml-document-length
            ' characters.'
        when 'END-OF-DOCUMENT'
          display 'End of document.'
        when 'VERSION-INFORMATION'
          display 'Version: <' XML-TEXT '>'
        when 'ENCODING-DECLARATION'
          display 'Encoding: <' XML-TEXT '>'
        when 'STANDALONE-DECLARATION'
          display 'Standalone: <' XML-TEXT '>'
        when 'ATTRIBUTE-NAME'
          display 'Attribute name: <' XML-TEXT '>'
        when 'ATTRIBUTE-CHARACTERS'
          display 'Attribute value characters: <' XML-TEXT '>'
        when 'ATTRIBUTE-CHARACTER'
          display 'Attribute value character: <' XML-TEXT '>'
        when 'START-OF-CDATA-SECTION'
          display 'Start of CData: <' XML-TEXT '>'
        when 'END-OF-CDATA-SECTION'
          display 'End of CData: <' XML-TEXT '>'
        when 'CONTENT-CHARACTER'
          display 'Content character: <' XML-TEXT '>'
        when 'PROCESSING-INSTRUCTION-TARGET'
          display 'PI target: <' XML-TEXT '>'
        when 'PROCESSING-INSTRUCTION-DATA'
          display 'PI data: <' XML-TEXT '>'
        when 'COMMENT'
          display 'Comment: <' XML-TEXT '>'
        when 'EXCEPTION'
          compute xml-document-length = function length (XML-TEXT)
          display 'Exception ' XML-CODE ' at offset '
            xml-document-length '.'
        when other
          display 'Unexpected XML event: ' XML-EVENT '.'
      end-evaluate
    .
End program xmlsaml1.

```

Output from parse example: From the following output you can see which events of the parse came from which fragments of the document:

```

Start of document: length=390 characters.
Version: <1.0>
Encoding: <ibm-37>
Standalone: <yes>
Comment: <This document is just an example>
Start element tag: <sandwich>
Content characters: < >
Start element tag: <bread>
Attribute name: <type>
Attribute value characters: <baker>
Attribute value character: <'>
Attribute value characters: <s best>
End element tag: <bread>

```

```

Content characters: < >
PI target: <spread>
PI data: <please use real mayonnaise >
Content characters: < >
Start element tag: <meat>
Content characters: <Ham >
Content character: <&>
Content characters: < turkey>
End element tag: <meat>
Content characters: < >
Start element tag: <filling>
Content characters: <Cheese, lettuce, tomato, etc.>
End element tag: <filling>
Content characters: < >
Start of CData: <<![CDATA[>
Content characters: <We should add a <relish> element in future!>
End of CData: <]]>>
Content characters: < >
Start element tag: <listprice>
Content characters: <$4.99 >
End element tag: <listprice>
Content characters: < >
Start element tag: <discount>
Content characters: <0.10>
End element tag: <discount>
End element tag: <sandwich>
End of document.
XML document successfully parsed

```

```

-----+***** Using information from XML *****-----

```

```

    Sandwich list price: $4.99
    Promotional price:   $4.49
    Get one today!

```

Example: parsing XML from an IFS file

This example shows an XML PARSE statement that parses an XML document located in an IFS file. The output from the program is the same as in the previous example. The IFS file must have a valid CCSID. The end of each line in the IFS file must have only a CR (carriage return) and not a LF (line feed).

```

Process APOST
Identification division.
  Program-id. xmlsaml2.

```

```

Data division.
  Working-storage section.

```

```

*****
* XML document, encoded as initial values of data items. *
*****
  1 xml-id pic x(27) value '/home/user1/xmlsaml2.xml'.
  1 xml-document-length computational pic 999.

```

```

*****
* Sample data definitions for processing numeric XML content. *
*****
  1 current-element pic x(30).
  1 list-price computational pic 9v99 value 0.
  1 discount computational pic 9v99 value 0.
  1 display-price pic $$9.99.

```

```

Procedure division.
  mainline section.

```

```

XML PARSE FILE-STREAM xml-id PROCESSING PROCEDURE xml-handler

```

```

ON EXCEPTION
  display 'XML document error ' XML-CODE
NOT ON EXCEPTION
  display 'XML document successfully parsed'
END-XML

*****
*   Process the transformed content and calculate promo price. *
*****
display ' '
display '-----+***** Using information from XML '
      '*****+-----'
display ' '
move list-price to display-price
display ' Sandwich list price: ' display-price
compute display-price = list-price * (1 - discount)
display ' Promotional price: ' display-price
display ' Get one today!'

goback.

xml-handler section.
evaluate XML-EVENT
* ==> Order XML events most frequent first
  when 'START-OF-ELEMENT'
    display 'Start element tag: <' XML-TEXT '>'
    move XML-TEXT to current-element
  when 'CONTENT-CHARACTERS'
    display 'Content characters: <' XML-TEXT '>'
* ==> Transform XML content to operational COBOL data item...
  evaluate current-element
    when 'listprice'
* ==> Using function NUMVAL-C...
    compute list-price = function numval-c(XML-TEXT)
    when 'discount'
    compute discount = function numval-c(XML-TEXT)
  end-evaluate
when 'END-OF-ELEMENT'
  display 'End element tag: <' XML-TEXT '>'
  move spaces to current-element
when 'START-OF-DOCUMENT'
  compute xml-document-length = function length(XML-TEXT)
  display 'Start of document: length=' xml-document-length
    ' characters.'
when 'END-OF-DOCUMENT'
  display 'End of document.'
when 'VERSION-INFORMATION'
  display 'Version: <' XML-TEXT '>'
when 'ENCODING-DECLARATION'
  display 'Encoding: <' XML-TEXT '>'
when 'STANDALONE-DECLARATION'
  display 'Standalone: <' XML-TEXT '>'
when 'ATTRIBUTE-NAME'
  display 'Attribute name: <' XML-TEXT '>'
when 'ATTRIBUTE-CHARACTERS'
  display 'Attribute value characters: <' XML-TEXT '>'
when 'ATTRIBUTE-CHARACTER'
  display 'Attribute value character: <' XML-TEXT '>'
when 'START-OF-CDATA-SECTION'
  display 'Start of CData: <' XML-TEXT '>'
when 'END-OF-CDATA-SECTION'
  display 'End of CData: <' XML-TEXT '>'
when 'CONTENT-CHARACTER'
  display 'Content character: <' XML-TEXT '>'
when 'PROCESSING-INSTRUCTION-TARGET'
  display 'PI target: <' XML-TEXT '>'
when 'PROCESSING-INSTRUCTION-DATA'

```

```

        display 'PI data: <' XML-TEXT '>'
    when 'COMMENT'
        display 'Comment: <' XML-TEXT '>'
    when 'EXCEPTION'
        compute xml-document-length = function length (XML-TEXT)
        display 'Exception ' XML-CODE ' at offset '
            xml-document-length '.'
    when other
        display 'Unexpected XML event: ' XML-EVENT '.'
end-evaluate
.
End program xmlsaml2.

```

Here is the IFS file for this example, from /home/user1/xmlsaml2doc.xml.

```

<?xml version="1.0" encoding="ibm-37"
  standalone="yes"?>
<!--This document is just an example-->
<sandwich>
  <bread type="baker&apos;s best"/>
  <?spread please use real mayonnaise ?>
  <meat>Ham &amp; turkey</meat>
  <filling>Cheese, lettuce, tomato, etc.
</filling>
  <![CDATA[We should add a <relish> element in future!]]>
  <listprice>$4.99 </listprice>
  <discount>0.10</discount>
</sandwich>

```

Understanding XML document encoding

The XML PARSE statement supports only XML documents that contain one of the following types of data items:

- National data items that are encoded using Unicode UCS-2
- Alphanumeric data items that are encoded using one of the supported single-byte EBCDIC or ASCII character sets

If your XML document includes an encoding declaration, ensure that it is consistent with the encoding information provided by your XML PARSE statement and with the basic encoding of the document. The parser determines the encoding of a document by using up to three sources of information in the following order:

1. The initial characters of the document
2. The encoding information provided by your XML PARSE statement
3. If step 2 succeeds, an encoding declaration in the document

Thus if the XML document begins with an XML declaration that includes an encoding declaration specifying one of the supported code pages, the parser honors the encoding declaration if it does not conflict with either the basic document encoding or the encoding information from the XML PARSE statement.

If the XML document does not have an XML declaration, or if the XML declaration omits the encoding declaration, the parser uses the encoding information from your XML PARSE statement to process the document, as long as it does not conflict with the basic document encoding.

The parser signals an XML exception event if it finds a conflict among these sources.

Specifying the code page

You can specify the encoding information for the document in the XML declaration, with which most XML documents begin. This is an example of an XML declaration that includes an encoding declaration:

```
<?xml version="1.0" encoding="ibm-1140" ?>
```

Specify the encoding declaration in either of the following ways:

- Specify the CCSID number (with or without any number of leading zeros), prefixed by any of the following (in any mixture of uppercase or lowercase):
 - IBM-
 - IBM_
 - CCSID-
 - CCSID_
- Use one of the following supported aliases (in any mixture of uppercase or lowercase):

Table 20. Aliases for XML encoding declarations

Code page	Supported aliases
037	EBCDIC-CP-US, EBCDIC-CP-CA, EBCDIC-CP-WT, EBCDIC-CP-NL
500	EBCDIC-CP-BE, EBCDIC-CP-CH
813	iso-8859-7, iso_8859-7
819	iso-8859-1, iso_8859-1
920	iso-8859-9, iso_8859-9

Parsing documents in other code pages

```
# You can parse XML documents that are encoded in code pages other than the
# explicitly supported single-byte code pages by converting them to Unicode UCS-2
# in a national data item, using the MOVE statement when the document is in a
# COBOL data item. If the XML document is in an IFS file, use the copy object (CPY)
# command to copy and convert the document to the UCS-2 CCSID specified in the
# National CCSID compiler option or in the NTLCCSID PROCESS option. You can
# then convert the individual pieces of document text passed to your processing
# procedure in special register XML-NTEXT back to the original code page as necessary,
# using the MOVE statement.
```

RELATED REFERENCES

Coded character sets for XML documents (*ILE COBOL Language Reference*)

Handling errors in XML documents

Use these facilities to handle errors in your XML document:

- Your processing procedure
- The ON EXCEPTION phrase of your XML PARSE statement
- Special register XML-CODE

When the XML parser detects an error in an XML document, it generates an XML exception event. The parser returns this exception event by passing control to your processing procedure along with the following information:

- The special register XML-EVENT contains 'EXCEPTION'.

- The special register XML-CODE contains the numeric exception code.
- XML-TEXT contains the document text up to and including the point where the exception was detected.

If the value of the error code is within one of the following ranges:

- 1-99
- 100,001-165,535
- 200,001-265,535

you might be able to handle the exception in your processing procedure and continue the parse. If the error code has any other nonzero value, the parse cannot continue. The exceptions for encoding conflicts (50-99 and 300-499) are signaled before the parse of the document is started. For these exceptions, XML-TEXT is either zero length or contains just the encoding declaration value from the document.

Exceptions in the range 1-49 are fatal errors according to the XML specification, therefore the parser does not continue normal parsing even if you handle the exception. However the parser does continue scanning for further errors until it reaches the end of the document or encounters an error that does not allow continuation. For these exceptions, the parser does not signal any further normal events, except for the END-OF-DOCUMENT event.

Use the following figure to understand the flow of control between the parser and your processing procedure. It illustrates how you can handle certain exceptions and how you use XML-CODE to identify the exception. The off-page connectors, such as , connect the multiple charts in this chapter. In particular,  in the following figure connects to the chart XML CCSID exception flow control. Within this figure  /  serves both as an off-page and an on-page connector.

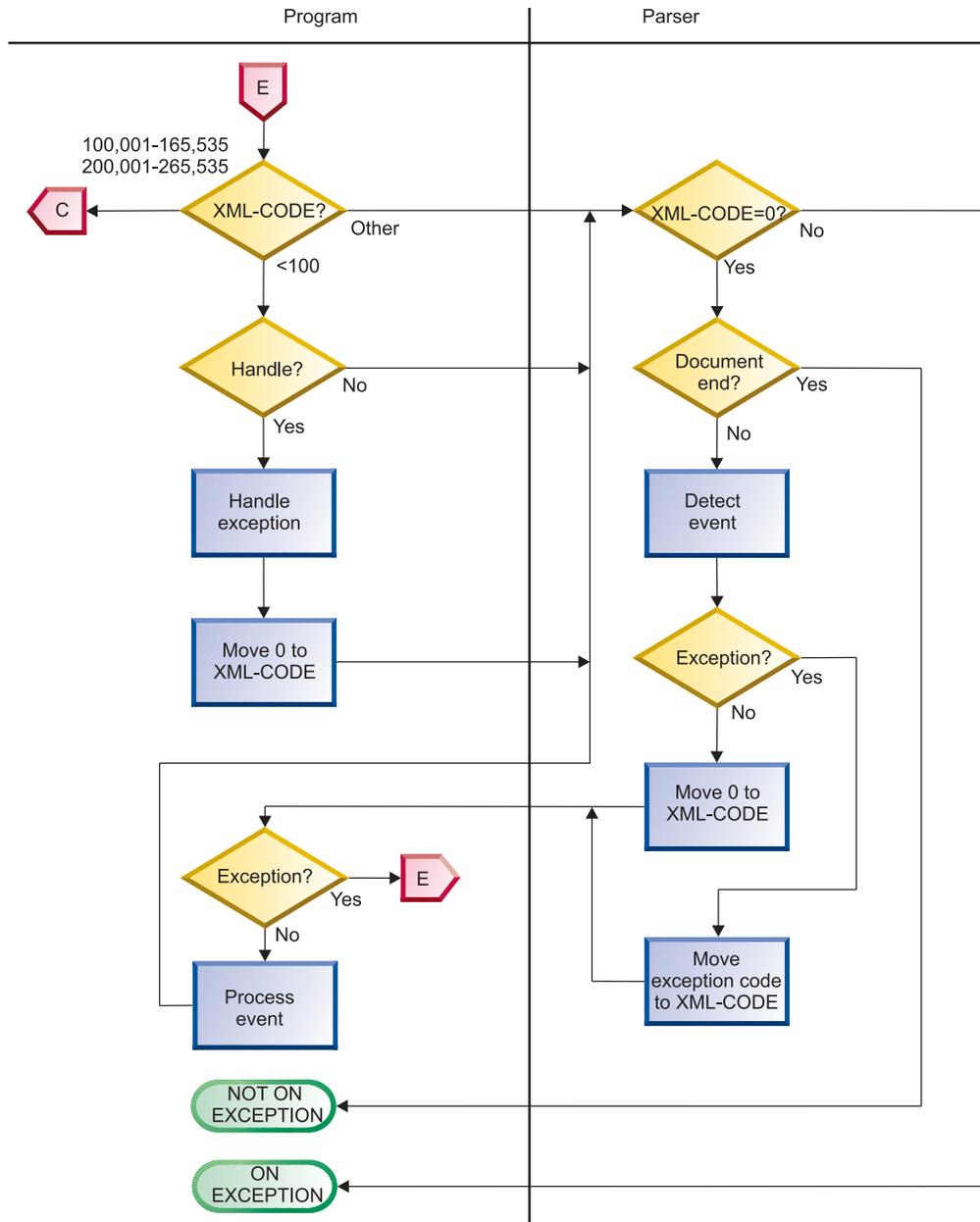


Figure 69. Control flow for XML exceptions

Unhandled exceptions

If you do not want to handle the exception, return control to the parser without changing the value of XML-CODE. The parser then transfers control to the statement that you specify on the ON EXCEPTION phrase. If you do not code an ON EXCEPTION phrase, control is transferred to the end of the XML PARSE statement.

Handling exceptions

To handle the exception event in your processing procedure, do these steps:

1. Use the contents of XML-CODE to guide your actions.
2. Set XML-CODE to zero to indicate that you have handled the exception.
3. Return control to the parser. The exception condition then no longer exists.

If no unhandled exceptions occur before the end of parsing, control is passed to the statement that you specify on the NOT ON EXCEPTION phrase (normal end of parse). If you do not code a NOT ON EXCEPTION phrase, control is passed to the end of the XML PARSE statement. The special register XML-CODE contains zero.

You can handle exceptions in this way only if the exception code passed in XML-CODE is within one of the following ranges:

- 1-99
- 100,001-165,535
- 200,001-265,535

Otherwise, the parser signals no further events, and passes control to the statement that you specify on your ON EXCEPTION phrase. In this case, XML-CODE contains the original exception number, even if you set XML-CODE to zero in your processing procedure before returning control to the parser.

If you return control to the parser with XML-CODE set to a nonzero value different from the original exception code, the results are undefined.

Terminating the parse

You can terminate parsing deliberately by setting XML-CODE to -1 in your processing procedure before returning to the parser from any normal XML event (that is, not an EXCEPTION event). You can use this technique when you have seen enough of the document for your purposes or have detected some irregularity in the document that precludes further meaningful processing.

In this case, the parser does not signal any further events, although an exception condition exists. Therefore control returns to your ON EXCEPTION phrase, if you have specified it. There you can test if XML-CODE is -1, which indicates that you terminated the parse deliberately. If you do not specify an ON EXCEPTION phrase, control returns to the end of the XML PARSE statement.

You can also terminate parsing after any exception XML event by returning to the parser without changing XML-CODE. The result is similar to the result of deliberate termination, except that the parser returns to the XML PARSE statement with XML-CODE containing the exception number.

CCSID conflict exception

A special case applies to exception events where the exception code in XML-CODE is in the range 100,001 through 165,535 or 200,001 through 265,535. These ranges of exception codes indicate that the CCSID of the document (determined by examining the beginning of the document, including any encoding declaration) conflicts with the CCSID for the XML PARSE statement.

In this case you can determine the CCSID of the document by subtracting 100,000 or 200,000 from the value of XML-CODE (depending on whether it is an EBCDIC CCSID or ASCII CCSID, respectively). For instance, if XML-CODE contains 101,140, the CCSID of the document is 1140.

#

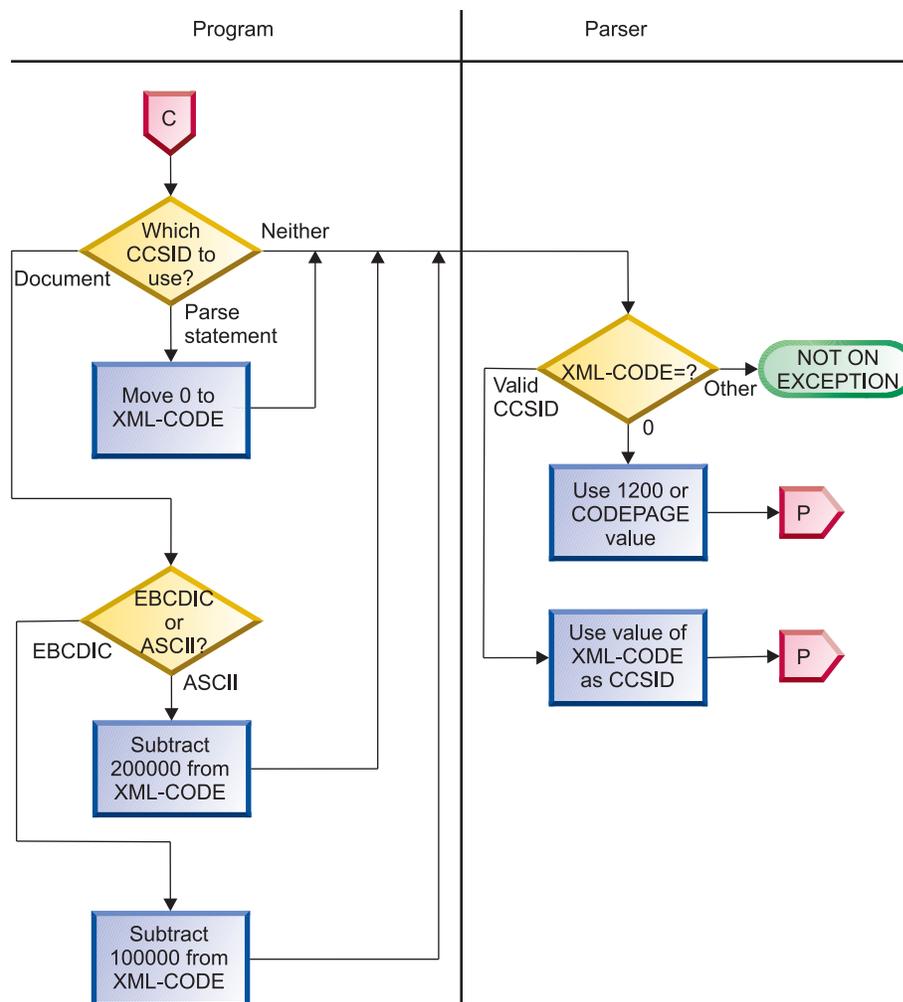
The CCSID for your XML PARSE statement depends on the type of the XML PARSE identifier. If the identifier is a national data item, the CCSID is specified in the National CCSID compiler option or in the NTLCCSID PROCESS option, indicating Unicode. If the XML PARSE identifier is alphanumeric, the CCSID is that of the COBOL source member.

The parser takes one of three actions after returning from your processing procedure for a CCSID conflict exception event:

- #
- #
- #
- #
- If you set XML-CODE to zero, the parser uses the CCSID for the XML PARSE statement: the CCSID specified in the National CCSID compiler option or in the NTLCCSID PROCESS option for national items; the CCSID of the COBOL source member, otherwise.
- If you set XML-CODE to the CCSID of the document (that is, the original XML-CODE value minus 100,000 or 200,000 appropriately), the parser uses the CCSID of the document. This is the only case where the parser continues when XML-CODE has a nonzero value upon returning from your processing procedure.
- Otherwise, the parser stops processing the document, and returns control to your XML PARSE statement with an exception condition. XML-CODE contains the exception code that was originally passed to the exception event.

The following figure illustrates these actions. The off-page connectors, such as  , connect the multiple charts in this chapter. In particular,  in the following figure connects to Control flow between XML parser and program, showing XML-CODE usage and  connects from Control flow for XML exceptions.

XML CCSID exception flow control



RELATED REFERENCES

"XML exceptions that allow continuation" on page 635

"XML exceptions that do not allow continuation" on page 639

Appendix F, "XML reference material," on page 635

XML-CODE (*ILE COBOL Language Reference*)

Chapter 12. Producing XML output

You can produce XML output from a COBOL program by using the XML GENERATE statement. In the XML GENERATE statement, you can also identify a field to receive a count of the number of characters of XML output generated, and a statement to receive control if an exception occurs.

To produce XML output,

- You can use the XML GENERATE statement to identify the source and target data items, count field, and ON EXCEPTION statement
- You can use special register XML-CODE to determine the status of XML generation
- Alternatively, you can generate the XML document to an IFS stream file, and use format 2 of the XML GENERATE statement to generate the document.

After you transform COBOL data items to XML, you can use the resulting XML output in various ways, such as passing it as a message to WebSphere MQ, or transmitting it for subsequent conversion to a CICS communication area.

RELATED TASKS

“Generating XML output”

“Enhancing XML output” on page 306

“Controlling the encoding of generated XML output” on page 311

“Handling errors in generating XML output” on page 311

Generating XML output

To transform COBOL data to XML, use the XML GENERATE statement as in the following example:

```
XML GENERATE XML-OUTPUT FROM SOURCE-REC
      COUNT IN XML-CHAR-COUNT
ON EXCEPTION
  DISPLAY 'XML generation error ' XML-CODE
  STOP RUN
NOT ON EXCEPTION
  DISPLAY 'XML document was successfully generated.'
END-XML
```

In the XML GENERATE statement, you first identify the data item (XML-OUTPUT in the example) that is to receive the XML output. Define the data item to be large enough to contain the generated XML output, typically five to eight times the size of the COBOL source data depending on the length of its data-name or data-names.

In the DATA DIVISION, you can declare the receiving identifier as alphanumeric (either an alphanumeric group item or an elementary item of category alphanumeric) or as national (an elementary item of category national).

The receiving identifier must be national if the XML output will contain any data from the COBOL source record that has any of the following characteristics:

- Is of class national or class DBCS
- Has a DBCS name (that is, is a data item whose name contains DBCS characters)

Next you identify the source data item that is to be transformed to XML format (SOURCE-REC in the example). The source data item can be an alphanumeric group item or elementary data item of class alphanumeric or national. Do not specify the RENAME clause in the data description of that data item.

If the source data item is an alphanumeric group item, the source data item is processed as a group item, not as an elementary item. Any groups that are subordinate to the source data item are also processed as group items.

Some COBOL data items are not transformed to XML, but are ignored. Subordinate data items of an alphanumeric group item that you transform to XML are ignored if they:

- Specify the REDEFINES clause, or are subordinate to such a redefining item
- Specify the RENAME clause

These items in the source data item are also ignored when you generate XML:

- Elementary FILLER (or unnamed) data items
- Slack bytes inserted for SYNCHRONIZED data items

There must be at least one elementary data item that is not ignored when you generate XML. For the data items that are not ignored, ensure that the identifier that you transform to XML satisfies these conditions when you declare it in the DATA DIVISION:

- Each elementary data item is either an index data item or belongs to one of these classes:
 - Alphanumeric
 - Alphanumeric
 - DBCS
 - Numeric
 - National

That is, no elementary data item is described with the USAGE POINTER, or USAGE PROCEDURE-POINTER phrase.

- Each data-name other than FILLER is unique within the immediately containing group, if any.
- Any DBCS data-names, when converted to Unicode, are legal as names in the XML specification, version 1.0.

An XML declaration is not generated. No white space (for example, new lines or indentation) is inserted to make the generated XML more readable.

Optionally, you can code the COUNT IN phrase to obtain the number of XML character positions that are filled during generation of the XML output. Declare the count field as an integer data item that does not have the symbol P in its PICTURE string. You can use the count field and reference modification to obtain only that portion of the receiving data item that contains the generated XML output. For example, XML-OUTPUT(1:XML-CHAR-COUNT) references the first XML-CHAR-COUNT character positions of XML-OUTPUT.

In addition, you can specify either or both of the following phrases to receive control after generation of the XML document:

- ON EXCEPTION, to receive control if an error occurs during XML generation
- NOT ON EXCEPTION, to receive control if no error occurs

You can end the XML GENERATE statement with the explicit scope terminator END-XML. Code END-XML to nest an XML GENERATE statement that has the ON EXCEPTION or NOT ON EXCEPTION phrase in a conditional statement.

XML generation continues until either the COBOL source record has been transformed to XML or an error occurs. If an error occurs, the results are as follows:

- Special register XML-CODE contains a nonzero exception code.
- Control is passed to the ON EXCEPTION phrase, if specified, otherwise to the end of the XML GENERATE statement.

If no error occurs during XML generation, special register XML-CODE contains zero, and control is passed to the NOT ON EXCEPTION phrase if specified or to the end of the XML GENERATE statement otherwise.

“Example: generating XML”

RELATED TASKS

“Controlling the encoding of generated XML output” on page 311

“Handling errors in generating XML output” on page 311

RELATED REFERENCES

Classes and categories of data (*ILE COBOL Language Reference*)

XML GENERATE statement (*ILE COBOL Language Reference*)

Operation of XML GENERATE (*ILE COBOL Language Reference*)

Example: generating XML

The following example simulates the building of a purchase order in a group data item, and generates an XML version of that purchase order.

Program XGFX uses XML GENERATE to produce XML output in elementary data item xmlP0 from the source record, group data item purchaseOrder. Elementary data items in the source record are converted to character format as necessary, and the characters are inserted in XML elements whose names are derived from the data-names in the source record.

XGFX calls program Pretty, which uses the XML PARSE statement with processing procedure p to format the XML output with new lines and indentation so that the XML content can more easily be verified.

Program XGFX

```
PROCESS NOMONOPRC.
Identification division.
  Program-id. XGFX.
Data division.
  Working-storage section.
    01 numItems pic 99 global.
    01 purchaseOrder global.
      05 orderDate pic x(10).
      05 shipTo.
        10 country pic xx value 'US'.
        10 name pic x(30).
        10 street pic x(30).
        10 city pic x(30).
        10 state pic xx.
        10 zip pic x(10).
      05 billTo.
        10 country pic xx value 'US'.
```

```

    10 name pic x(30).
    10 street pic x(30).
    10 city pic x(30).
    10 state pic xx.
    10 zip pic x(10).
    05 orderComment pic x(80).
    05 items occurs 0 to 20 times depending on numItems.
    10 item.
        15 partNum pic x(6).
        15 productName pic x(50).
        15 quantity pic 99.
        15 USPrice pic 999v99.
        15 shipDate pic x(10).
        15 itemComment pic x(40).
    01 numChars comp pic 9(9).
    01 xmlPO pic x(999).
Procedure division.
m.
    Move 20 to numItems
    Move spaces to purchaseOrder

    Move '1999-10-20' to orderDate

    Move 'US' to country of shipTo
    Move 'Alice Smith' to name of shipTo
    Move '123 Maple Street' to street of shipTo
    Move 'Mill Valley' to city of shipTo
    Move 'CA' to state of shipTo
    Move '90952' to zip of shipTo

    Move 'US' to country of billTo
    Move 'Robert Smith' to name of billTo
    Move '8 Oak Avenue' to street of billTo
    Move 'Old Town' to city of billTo
    Move 'PA' to state of billTo
    Move '95819' to zip of billTo
    Move 'Hurry, my lawn is going wild!' to orderComment

    Move 0 to numItems
    Call 'addFirstItem'
    Call 'addSecondItem'
    Move space to xmlPO
    Xml generate xmlPO from purchaseOrder count in numChars
    Call 'PRETTY' using xmlPO numChars
    Goback
.

```

Identification division.

Program-id. 'addFirstItem'.

Procedure division.

```

    Add 1 to numItems
    Move '872-AA' to partNum(numItems)
    Move 'Lawnmower' to productName(numItems)
    Move 1 to quantity(numItems)
    Move 148.95 to USPrice(numItems)
    Move 'Confirm this is electric' to itemComment(numItems)
    Goback.

```

End program 'addFirstItem'.

Identification division.

Program-id. 'addSecondItem'.

Procedure division.

```

    Add 1 to numItems
    Move '926-AA' to partNum(numItems)
    Move 'Baby Monitor' to productName(numItems)
    Move 1 to quantity(numItems)
    Move 39.98 to USPrice(numItems)

```

```

        Move '1999-05-21' to shipDate(numItems)
        Goback.
    End program 'addSecondItem'.

End program XGFX.

```

Program Pretty

```

Identification division.
  Program-id. Pretty.
Data division.
  Working-storage section.
    01 prettyPrint.
      05 pose pic 999.
      05 posd pic 999.
      05 depth pic 99.
      05 element pic x(30).
      05 indent pic x(20).
      05 buffer pic x(100).
  Linkage section.
    1 doc.
      2 pic x occurs 1 to 16384 times depending on len.
    1 len comp pic 9(9).
  Procedure division using doc len.
    m.
      Move space to prettyPrint
      Move 0 to depth posd
      Move 1 to pose
      Xml parse doc processing procedure p
      Goback.
    p.
      Evaluate xml-event
      When 'START-OF-ELEMENT'
        If element not = space
          If depth > 1
            Display indent(1:2 * depth - 2) buffer(1:pose - 1)
          Else
            Display buffer(1:pose - 1)
          End-if
        End-if
      Move xml-text to element
      Add 1 to depth
      Move 1 to pose
      String '<' xml-text '>' delimited by size into buffer
        with pointer pose
      Move pose to posd
      When 'CONTENT-CHARACTERS'
        String xml-text delimited by size into buffer
          with pointer posd
      When 'CONTENT-CHARACTER'
        String xml-text delimited by size into buffer
          with pointer posd
      When 'END-OF-ELEMENT'
        Move space to element
        String '</' xml-text '>' delimited by size into buffer
          with pointer posd
        If depth > 1
          Display indent(1:2 * depth - 2) buffer(1:posd - 1)
        Else
          Display buffer(1:posd - 1)
        End-if
        Subtract 1 from depth
        Move 1 to posd
      When other
        Continue
      End-evaluate
    .
  End program Pretty.

```

Output from program XGFX

```
<purchaseOrder>
  <orderDate>1999-10-20</orderDate>
  <shipTo>
    <country>US</country>
    <name>Alice Smith</name>
    <street>123 Maple Street</street>
    <city>Mill Valley</city>
    <state>CA</state>
    <zip>90952</zip>
  </shipTo>
  <billTo>
    <country>US</country>
    <name>Robert Smith</name>
    <street>8 Oak Avenue</street>
    <city>Old Town</city>
    <state>PA</state>
    <zip>95819</zip>
  </billTo>
  <orderComment>Hurry, my lawn is going wild!</orderComment>
  <items>
    <item>
      <partNum>872-AA</partNum>
      <productName>Lawnmower</productName>
      <quantity>1</quantity>
      <USPrice>148.95</USPrice>
      <shipDate> </shipDate>
      <itemComment>Confirm this is electric</itemComment>
    </item>
  </items>
  <items>
    <item>
      <partNum>926-AA</partNum>
      <productName>Baby Monitor</productName>
      <quantity>1</quantity>
      <USPrice>39.98</USPrice>
      <shipDate>1999-05-21</shipDate>
      <itemComment> </itemComment>
    </item>
  </items>
</purchaseOrder>
```

RELATED REFERENCES

Operation of XML GENERATE (*ILE COBOL Language Reference*)

Enhancing XML output

It might happen that the information that you want to express in XML format already exists in a group item in the DATA DIVISION, but you are unable to use that item directly to generate an XML document because of one or more factors.

For example:

- In addition to the required data, the item has subordinate data items that contain values that are irrelevant to the XML output document.
- The names of the required data items are unsuitable for external presentation, and are possibly meaningful only to programmers.
- The definition of the data is not of the required data type. Perhaps only the redefinitions (which are ignored by the XML GENERATE statement) have the appropriate format.

- The required data items are nested too deeply within irrelevant subordinate groups. The XML output should be “flattened” rather than hierarchical as it would be by default.
- The required data items are broken up into too many components, and should be output as the content of the containing group.
- The group item contains the required information, but in the wrong order.

There are various ways that you can deal with such situations. One possible technique is to define a new data item that has the appropriate characteristics, and move the required data to the appropriate fields of this new data item. However, this approach is somewhat laborious and requires careful maintenance to keep the original and new data items synchronized.

An alternative approach that has some advantages is to provide a redefinition of the original group data item, and to generate the XML output from that redefinition. To do so, start from the original set of data descriptions, and make these changes:

- Exclude elementary data items from the generated XML either by renaming them to FILLER or by deleting their names.
- Provide more meaningful and appropriate names for the selected elementary items and for the group items that contain them.
- Remove unneeded intermediate group items to flatten the hierarchy.
- Specify different data types to obtain the desired trimming behavior.
- Choose a different order for the output by using a sequence of XML GENERATE statements.

The safest way to accomplish these changes is to use another copy of the original declarations accompanied by one or more REPLACE compiler-directing statements.

“Example: enhancing XML output”

You might also find when you generate an XML document that some of the element names and element values contain hyphens, but you want to convert the hyphens in the element names to underscores without changing the hyphens that are in the element values. The example that is referenced below shows a way to do so.

“Example: converting hyphens in element names to underscores” on page 310

RELATED REFERENCES

Operation of XML GENERATE (*ILE COBOL Language Reference*)

Example: enhancing XML output

Consider the following example data structure. The XML that is generated from the structure suffers from several problems that can be corrected.

```
01 CDR-LIFE-BASE-VALUES-BOX.
   15 CDR-LIFE-BASE-VAL-DATE PIC X(08).
   15 CDR-LIFE-BASE-VALUE-LINE OCCURS 2 TIMES.
      20 CDR-LIFE-BASE-DESC.
         25 CDR-LIFE-BASE-DESC1 PIC X(15).
         25 FILLER PIC X(01).
         25 CDR-LIFE-BASE-LIT PIC X(08).
         25 CDR-LIFE-BASE-DTE PIC X(08).
      20 CDR-LIFE-BASE-PRICE.
         25 CDR-LIFE-BP-SPACE PIC X(02).
```

```

25 CDR-LIFE-BP-DASH PIC X(02).
25 CDR-LIFE-BP-SPACE1 PIC X(02).
20 CDR-LIFE-BASE-PRICE-ED REDEFINES
   CDR-LIFE-BASE-PRICE PIC $$$.$$.
20 CDR-LIFE-BASE-QTY.
25 CDR-LIFE-QTY-SPACE PIC X(08).
25 CDR-LIFE-QTY-DASH PIC X(02).
25 CDR-LIFE-QTY-SPACE1 PIC X(02).
25 FILLER PIC X(02) VALUE "00".
20 CDR-LIFE-BASE-QTY-ED REDEFINES
   CDR-LIFE-BASE-QTY PIC ZZ,ZZZ,ZZZ.ZZZ.
20 CDR-LIFE-BASE-VALUE PIC X(15).
20 CDR-LIFE-BASE-VALUE-ED REDEFINES
   CDR-LIFE-BASE-VALUE
   PIC $(4),$$,$$9.99.
15 CDR-LIFE-BASE-TOT-VALUE-LINE.
20 CDR-LIFE-BASE-TOT-VALUE PIC X(15).

```

When this data structure is populated with some sample values, and XML is generated directly from it and then formatted using program Pretty (shown in "Example: generating XML" on page 303), the result is as follows:

```

<CDR-LIFE-BASE-VALUES-BOX>
  <CDR-LIFE-BASE-VAL-DATE>01/02/03</CDR-LIFE-BASE-VAL-DATE>
  <CDR-LIFE-BASE-VALUE-LINE>
    <CDR-LIFE-BASE-DESC>
      <CDR-LIFE-BASE-DESC1>First</CDR-LIFE-BASE-DESC1>
      <CDR-LIFE-BASE-LIT> </CDR-LIFE-BASE-LIT>
      <CDR-LIFE-BASE-DTE>01/01/01</CDR-LIFE-BASE-DTE>
    </CDR-LIFE-BASE-DESC>
    <CDR-LIFE-BASE-PRICE>
      <CDR-LIFE-BP-SPACE>$2</CDR-LIFE-BP-SPACE>
      <CDR-LIFE-BP-DASH>3.</CDR-LIFE-BP-DASH>
      <CDR-LIFE-BP-SPACE1>00</CDR-LIFE-BP-SPACE1>
    </CDR-LIFE-BASE-PRICE>
    <CDR-LIFE-BASE-QTY>
      <CDR-LIFE-QTY-SPACE> 1</CDR-LIFE-QTY-SPACE>
      <CDR-LIFE-QTY-DASH>23</CDR-LIFE-QTY-DASH>
      <CDR-LIFE-QTY-SPACE1>.0</CDR-LIFE-QTY-SPACE1>
    </CDR-LIFE-BASE-QTY>
    <CDR-LIFE-BASE-VALUE> $765.00</CDR-LIFE-BASE-VALUE>
  </CDR-LIFE-BASE-VALUE-LINE>
  <CDR-LIFE-BASE-VALUE-LINE>
    <CDR-LIFE-BASE-DESC>
      <CDR-LIFE-BASE-DESC1>Second</CDR-LIFE-BASE-DESC1>
      <CDR-LIFE-BASE-LIT> </CDR-LIFE-BASE-LIT>
      <CDR-LIFE-BASE-DTE>02/02/02</CDR-LIFE-BASE-DTE>
    </CDR-LIFE-BASE-DESC>
    <CDR-LIFE-BASE-PRICE>
      <CDR-LIFE-BP-SPACE>$3</CDR-LIFE-BP-SPACE>
      <CDR-LIFE-BP-DASH>4.</CDR-LIFE-BP-DASH>
      <CDR-LIFE-BP-SPACE1>00</CDR-LIFE-BP-SPACE1>
    </CDR-LIFE-BASE-PRICE>
    <CDR-LIFE-BASE-QTY>
      <CDR-LIFE-QTY-SPACE> 2</CDR-LIFE-QTY-SPACE>
      <CDR-LIFE-QTY-DASH>34</CDR-LIFE-QTY-DASH>
      <CDR-LIFE-QTY-SPACE1>.0</CDR-LIFE-QTY-SPACE1>
    </CDR-LIFE-BASE-QTY>
    <CDR-LIFE-BASE-VALUE> $654.00</CDR-LIFE-BASE-VALUE>
  </CDR-LIFE-BASE-VALUE-LINE>
  <CDR-LIFE-BASE-TOT-VALUE-LINE>
    <CDR-LIFE-BASE-TOT-VALUE>Very high!</CDR-LIFE-BASE-TOT-VALUE>
  </CDR-LIFE-BASE-TOT-VALUE-LINE>
</CDR-LIFE-BASE-VALUES-BOX>

```

This generated XML suffers from several problems:

- The element names are long and not very meaningful.
- There is unwanted data, for example, CDR-LIFE-BASE-LIT and CDR-LIFE-BASE-DTE.
- Required data has an unnecessary parent. For example, CDR-LIFE-BASE-DESC1 has parent CDR-LIFE-BASE-DESC.
- Other required fields are split into too many subcomponents. For example, CDR-LIFE-BASE-PRICE has three subcomponents for one amount.

These and other characteristics of the XML output can be remedied by redefining the storage as follows:

```

1 BaseValues redefines CDR-LIFE-BASE-VALUES-BOX.
2 BaseValueDate pic x(8).
2 BaseValueLine occurs 2 times.
3 Description pic x(15).
3 pic x(9).
3 BaseDate pic x(8).
3 BasePrice pic x(6) justified.
3 BaseQuantity pic x(14) justified.
3 BaseValue pic x(15) justified.
2 TotalValue pic x(15).

```

The result of generating and formatting XML from the set of definitions of the data values shown above is more usable:

```

<BaseValues>
  <BaseValueDate>01/02/03</BaseValueDate>
  <BaseValueLine>
    <Description>First</Description>
    <BaseDate>01/01/01</BaseDate>
    <BasePrice>$23.00</BasePrice>
    <BaseQuantity>123.000</BaseQuantity>
    <BaseValue>$765.00</BaseValue>
  </BaseValueLine>
  <BaseValueLine>
    <Description>Second</Description>
    <BaseDate>02/02/02</BaseDate>
    <BasePrice>$34.00</BasePrice>
    <BaseQuantity>234.000</BaseQuantity>
    <BaseValue>$654.00</BaseValue>
  </BaseValueLine>
  <TotalValue>Very high!</TotalValue>
</BaseValues>

```

You can redefine the original data definition directly, as shown above. However, it is generally safer to use the original definition, but to modify it suitably using the text-manipulation capabilities of the compiler. An example is shown in the REPLACE compiler-directing statement below. This REPLACE statement might appear complicated, but it has the advantage of being self-maintaining if the original data definitions are modified.

```

replace ==CDR-LIFE-BASE-VALUES-BOX== by
  ==BaseValues redefines CDR-LIFE-BASE-VALUES-BOX==
  ==CDR-LIFE-BASE-VAL-DATE== by ==BaseValueDate==
  ==CDR-LIFE-BASE-VALUE-LINE== by ==BaseValueLine==
  ==20 CDR-LIFE-BASE-DESC.== by ====
  ==CDR-LIFE-BASE-DESC1== by ==Description==
  ==CDR-LIFE-BASE-LIT== by ====
  ==CDR-LIFE-BASE-DTE== by ==BaseDate==
  ==20 CDR-LIFE-BASE-PRICE.== by ====
  ==25 CDR-LIFE-BP-SPACE PIC X(02).== by ====
  ==25 CDR-LIFE-BP-DASH PIC X(02).== by ====
  ==25 CDR-LIFE-BP-SPACE1 PIC X(02).== by ====
  ==CDR-LIFE-BASE-PRICE-ED== by ==BasePrice==

```

```

==REDEFINES CDR-LIFE-BASE-PRICE PIC $$$.$$.== by
  ==pic x(6) justified.==
==20 CDR-LIFE-BASE-QTY.
   25 CDR-LIFE-QTY-SPACE PIC X(08).
   25 CDR-LIFE-QTY-DASH PIC X(02).
   25 CDR-LIFE-QTY-SPACE1 PIC X(02).
   25 FILLER PIC X(02).== by ====
==CDR-LIFE-BASE-QTY-ED== by ==BaseQuantity==
==REDEFINES CDR-LIFE-BASE-QTY PIC ZZ,ZZZ,ZZZ.ZZZ.== by
  ==pic x(14) justified.==
==CDR-LIFE-BASE-VALUE-ED== by ==BaseValue==
==20 CDR-LIFE-BASE-VALUE PIC X(15).== by ====
==REDEFINES CDR-LIFE-BASE-VALUE PIC $(4),$$,$$9.99.==
  by ==pic x(15) justified.==
==CDR-LIFE-BASE-TOT-VALUE-LINE. 20== by ====
==CDR-LIFE-BASE-TOT-VALUE== by ==TotalValue==.

```

The result of this REPLACE statement followed by a second instance of the original set of definitions is similar to the suggested redefinition of group item BaseValues shown above. This REPLACE statement illustrates a variety of techniques for eliminating unwanted definitions and for modifying the definitions that should be retained. Use whichever technique is appropriate for your situation.

RELATED REFERENCES

Operation of XML GENERATE (*ILE COBOL Language Reference*)
 REPLACE statement (*ILE COBOL Language Reference*)

Example: converting hyphens in element names to underscores

When you generate an XML document from a data structure whose items have data-names that contain hyphens, the generated XML will have element names that contain hyphens. This example shows a way to convert the hyphens in the element names to underscores without changing any hyphens that occur in the element values.

Consider the following data structure:

```

1 Customer-Record.
  2 Customer-Number pic 9(9).
  2 First-Name      pic x(10).
  2 Last-Name       pic x(20).

```

When this data structure is populated with some sample values, and XML is generated directly from it and then formatted using program Pretty (shown in “Example: generating XML” on page 303), the result is as follows:

```

<Customer-Record>
  <Customer-Number>12345</Customer-Number>
  <First-Name>John</First-Name>
  <Last-Name>Smith-Jones</Last-Name>
</Customer-Record>

```

The element names contain hyphens, and the content of the element Last-Name also contains a hyphen.

Assuming that this XML document is the content of data item xmlDoc, and that charcnt is set to the length of this XML document, you can change all the hyphens in the element names to underscores but leave the element values unchanged by using the following code:

```

1 xmldoc          pic x(16384).
1 charcnt  comp pic 9(5).
1 pos      comp pic 9(5).
1 tagstate comp pic 9 value zero.
. . .
dash-to-underscore.
  perform varying pos from 1 by 1
    until pos > charcnt
    if xmldoc(pos:1) = '<'
      move 1 to tagstate
    end-if
    if tagstate = 1 and xmldoc(pos:1) = '-'
      move '_' to xmldoc(pos:1)
    else
      if xmldoc(pos:1) = '>'
        move 0 to tagstate
      end-if
    end-if
  end-perform.

```

The revised XML document in data item `xml doc` has underscores instead of hyphens in the element names, as shown below:

```

<Customer_Record>
  <Customer_Number>12345</Customer_Number>
  <First_Name>John</First_Name>
  <Last_Name>Smith-Jones</Last_Name>
</Customer_Record>

```

Controlling the encoding of generated XML output

When you generate XML output by using the XML GENERATE statement, you can control the encoding of the output by the category of the data item that receives the XML output. The following table shows the possible output formats.

Table 21. Encoding of generated XML output

If you define the receiving XML identifier as:	The generated XML output is encoded in:
Alphanumeric	The CCSID specified by the PROCESS statement CCSID option <code>d</code> — XML GENERATE single-byte data CCSID in effect when the source was compiled. If the CCSID in effect is 65535, the job default CCSID at run time will be used.
National	Unicode (UCS-2, CCSID specified in the National CCSID compiler option or in the NTLCCSID PROCESS option) ¹
1. A <i>byte order mark</i> is not generated.	

For details about how data items are converted to XML and how the XML element names are formed from the COBOL data-names, see the related reference below about the operation of the XML GENERATE statement.

RELATED REFERENCES

Operation of XML GENERATE (*ILE COBOL Language Reference*)

Handling errors in generating XML output

When an error is detected during generation of XML output, an exception condition exists. You can write code to check the special register XML-CODE, which contains a numeric exception code that indicates the error type.

To handle errors, use either or both of the following phrases of the XML GENERATE statement:

- ON EXCEPTION
- COUNT IN

If you code the ON EXCEPTION phrase in the XML GENERATE statement, control is transferred to the imperative statement that you specify. You might code an imperative statement, for example, to display the XML-CODE value. If you do not code an ON EXCEPTION phrase, control is transferred to the end of the XML GENERATE statement.

When an error occurs, one problem might be that the data item that receives the XML output is not large enough. In that case, the XML output is not complete, and special register XML-CODE contains error code 400.

You can examine the generated XML output by doing these steps:

1. Code the COUNT IN phrase in the XML GENERATE statement.
The count field that you specify holds a count of the XML character positions that are filled during XML generation. If you define the XML output as national, the count is in national character positions (UCS-2 character encoding units); otherwise the count is in bytes.
2. Use the count field with reference modification to refer to the substring of the receiving data item that contains the generated XML output.
For example, if XML-OUTPUT is the data item that receives the XML output, and XML-CHAR-COUNT is the count field, then XML-OUTPUT(1:XML-CHAR-COUNT) references the XML output.

Use the contents of XML-CODE to determine what corrective action to take. For a list of the exceptions that can occur during XML generation, see the related reference below.

RELATED REFERENCES

“XML generate exceptions” on page 645

Chapter 13. Calling and Sharing Data with Other Languages

ILE COBOL can call or be called by other ILE, OPM, and EPM languages.

This chapter describes:

- How to call and pass data to another language from ILE COBOL
- How control is returned to ILE COBOL from the other language
- How to issue a CL command from an ILE COBOL program
- How to include Structured Query Language (SQL) statements in your ILE COBOL program.

In general:

- If your ILE COBOL program is **calling** another language, use a CALL statement with the USING phrase that points to the items that will constitute the parameter list. Control is returned to your program at the next statement after the CALL statement (unless the called program or any program called by it terminates the run unit).
- If your ILE COBOL program is **being called** with parameters by another language, you need a USING phrase on the PROCEDURE DIVISION statement, and a Linkage Section that describes the parameters to be received. Your ILE COBOL program can return control to the calling program with a GOBACK statement or an EXIT PROGRAM statement.
- Your ILE COBOL program can terminate the run unit with a STOP RUN statement or GOBACK statement provided that the nearest control boundary is a hard control boundary; the run unit will not be terminated if the nearest control boundary is a soft control boundary.

For a full description of how to call an ILE COBOL program from another language, refer to that language's programming guide.

One consideration for you when passing or receiving data with non-ILE COBOL programs is the matching of the parameter lists. If your ILE COBOL program is calling a non-ILE COBOL program, you must understand what is expected in the way of input, and set up your data items accordingly. If your program is being called, you must know what will be passed, and set up your Linkage Section to accept it.

Another consideration for you is the treatment of the RETURN-CODE special register. The RETURN-CODE special register cannot be set by a non-ILE COBOL program. When the RETURN-CODE special register contains an incorrect value after control has been returned from a called program, set the RETURN-CODE special register to a meaningful value before your ILE COBOL program returns control to its caller. Otherwise, an incorrect return code will be passed back to its caller.

Calling ILE C and VisualAge C++ Programs and Procedures

Note: All references to ILE C in this section also apply to VisualAge C++.

An ILE COBOL program can call ILE C programs and procedures using dynamic program calls or static procedure calls.

When a dynamic program call is used to call an ILE C program, the ILE C program must be compiled and bound as a separate program object. When a static procedure call is used to call an ILE C procedure, the ILE C procedure must first be compiled into a module object and then bound to the calling ILE COBOL program. In ILE C, an ILE procedure corresponds to an ILE C function. Refer to the *IBM Rational Development Studio for i: ILE C/C++ Programmer's Guide* for a description of compiling and binding ILE C programs and procedures.

You call an ILE C program or procedure from an ILE COBOL program by using the *CALL literal* statement (where *literal* is the name of the ILE C program or procedure). To call the ILE C program or procedure, you write the *CALL literal* statement in the same way as you would if you were calling another ILE COBOL subprogram. See "Using Static Procedure Calls and Dynamic Program Calls" on page 220 for detailed information about how to write the *CALL* statement in your ILE COBOL program to call an ILE C program using dynamic program calls or static procedure calls.

You can also call an ILE C program from an ILE COBOL program by using the *CALL identifier* statement. See "Using *CALL identifier*" on page 223 for more information on *CALL identifier*.

Alternately, you can use *CALL procedure-pointer* to call an ILE C program or procedure from an ILE COBOL program. See "Using *CALL procedure-pointer*" on page 224 for more information on *CALL procedure-pointer*. A procedure-pointer in ILE COBOL is similar to a function pointer in ILE C. You can pass a procedure-pointer as a argument on the *CALL* statement from ILE COBOL to an ILE C function and have the ILE C function define its parameter as a function pointer.

You can only call an ILE C function that returns a value if the *RETURNING* phrase of the ILE COBOL *CALL* statement has been specified.

Two or more ILE C programs in the same activation group can interact with each other's runtime resources. Refer to the *IBM Rational Development Studio for i: ILE C/C++ Programmer's Guide* for a description of how this is accomplished. Therefore, you should ensure that the ILE C programs you call from your ILE COBOL program are designed to work together in the same activation group. Two ILE C programs in the same activation group can share things like *errno*, signal vectors, and storage pools. If your ILE COBOL program needs to call more than one ILE C programs that are not designed to share the same run time then specify a different name for the activation group in which the ILE C program will run.

ILE C allows recursion but ILE COBOL does not for default program type. You need to use a *RECURSIVE* clause in *PROGRAM-ID* paragraph to make a COBOL program become a recursive program. If an ILE C function calls an ILE COBOL non recursive program recursively, a runtime error message will be generated from the ILE COBOL program.

To call an ILE C function from an ILE COBOL program, the name of the ILE C function being called may need to be case-sensitive, longer than 10 characters (up to 256 characters), and contain some special characters. In this case, use a static procedure call and compile your ILE COBOL program with the **NOMONOPRC* value of the *OPTION* parameter of the *CRTCBLMOD* or *CRTBNDCBL* commands.

When a ILE C++ compiler procedure is called from ILE COBOL, the keywords *extern "COBOL"* or *extern "C"* should be placed on the ILE C++ compiler function

prototype, to prevent the mangling of the ILE C++ compiler function name. Use extern "C" if ILE COBOL is passing BY VALUE arguments to ILE C++ compiler.

Passing Data to an ILE C Program or Procedure

You can pass data to a called ILE C program or procedure by using CALL...BY REFERENCE, CALL...BY VALUE, or CALL...BY CONTENT. Refer to "Passing Data Using CALL...BY REFERENCE, BY VALUE, or BY CONTENT" on page 233 for a description of how to use CALL...BY REFERENCE, CALL...BY VALUE or CALL...BY CONTENT.

When data is passed to the ILE C program using CALL...BY REFERENCE, a pointer to the data item is placed into the argument list that is accepted by the ILE C program.

When data is passed to the ILE C program using CALL...BY CONTENT, the value of the data item is copied to a temporary location and then a pointer containing the address of the copy's temporary location is placed into the argument list that is accepted by the ILE C program.

For CALL...BY VALUE, the value of the item is placed into the argument list that is accepted by the ILE C program. CALL...BY VALUE can be used to call ILE C procedures but not ILE C program objects.

In your ILE COBOL program, you describe the arguments that you want to pass to the ILE C program or procedure, in the Data Division in the same manner as you describe other data items in the Data Division. Refer to "Passing and Sharing Data Between Programs" on page 232 for a description of how to describe the arguments that you want to pass.

When the called ILE C program object begins running, the function `main` is automatically called. Every ILE C program object must have one function named `main`. When you pass parameters to the ILE C program object, you must declare two parameters with the function `main`. Although any name can be given to these parameters, they are usually referred to as `argc` and `argv`. The first parameter, `argc` (argument count), has type `int` and indicates how many arguments were supplied on the CALL statement that called the ILE C program object. The second parameter, `argv` (argument vector), has type array of pointers to char array objects.

The value of `argc` indicates the number of pointers in the array `argv`. If a program name is available, the first element in `argv` points to a character array that contains the program name of the invocation name of the ILE C program that is being run. The remaining elements in `argv` contain pointers to the parameters being passed to the called ILE C program. The last element, `argv[argc]`, always contains NULL.

Refer to the *IBM Rational Development Studio for i: ILE C/C++ Programmer's Guide* for further information on describing parameters in the called ILE C program or procedure.

Data Type Compatibility between ILE C and ILE COBOL

ILE C and ILE COBOL have different data types. When you want to pass data between programs written in ILE C and ILE COBOL, you must be aware of these differences. Some data types in ILE C and ILE COBOL have no direct equivalent in the other language.

An ILE C program always expects character strings to terminate with a null character; you should make sure that the string data passed to the ILE C program is null-terminated. Refer to “Manipulating null-terminated strings” on page 208 for further information.

Table 22 shows the ILE COBOL data type compatibility with ILE C.

Table 22. ILE COBOL Data Type Compatibility with ILE C

ILE COBOL	ILE C declaration in prototype	Length	Description
PIC X(n).	char[n] or char *	n	A character field where n=1 to 16 711 568
FORMAT DATE literal.	char[6]	6	A date field.
FORMAT TIME literal.	char[8]	8	A time field.
FORMAT TIMESTAMP.	char[n]	26	A timestamp field.
PIC G(n)	char[2n]	2n	A graphic field.
PIC 1 INDIC ..	char	1	An indicator.
PIC S9(n) DISPLAY	char[n]	n	A zoned decimal.
PIC S9(n-p)V9(p) COMP-3	decimal(n,p)	n/2+1	A packed decimal.
PIC S9(n-p)V9(p) PACKED-DECIMAL.	decimal(n,p)	n/2+1	A packed decimal.
PIC S9(4) COMP-4 BINARY	short int	2	A 2-byte signed integer with a range of -9999 to +9999.
PIC S9(4) COMP-4 with *NOSTDTRUNC PIC S9(4) BINARY with *NOSTDTRUNC PIC S9(4) COMP-5	short int	2	A 2-byte signed integer with a range of -32768 to +32767.
PIC S9(4) COMP-5 	unsigned short int	2	A 2-byte unsigned integer with a range of 0 to 65535.
PIC S9(9) COMP-4 PIC S9(9) BINARY 	int long int	4	A 4-byte signed integer with a range of -999999999 to +999999999.
PIC S9(9) COMP-4 with *NOSTDTRUNC PIC S9(9) BINARY with *NOSTDTRUNC PIC S9(9) COMP-5	int long int	4	A 4-byte signed integer with a range of -2147483648 to +2147483647.
PIC S9(9) COMP-5 	unsigned int	4	A 4-byte unsigned integer with a range of 0 to 4294967295.
PIC S9(18) COMP-4 PIC S9(18) BINARY	long long	8	An 8-byte integer.
PIC S9(18) COMP-4 with *NOSTDTRUNC PIC S9(18) BINARY with *NOSTDTRUNC PIC S9(18) COMP-5	long long	8	An 8-byte integer.
PIC S9(18) COMP-5 	unsigned int	8	A 8-byte unsigned integer.
05 VL-FIELD. 10 i PIC S9(4) COMP-4. 10 data PIC X(n).	_Packed struct {short i; char[n]}	n+2	A variable length field where i is the intended length and n is the maximum length.

Table 22. ILE COBOL Data Type Compatibility with ILE C (continued)

ILE COBOL	ILE C declaration in prototype	Length	Description
05 n PIC 9(9) COMP-4. 05 x redefines n PIC X(4).	struct {unsigned int : n};	4	Bitfields can be manipulated using hex literals.
01 record 05 field1... 05 field2...	struct	n	A structure. Use the <code>_Packed</code> qualifier on the struct. Structures passed should be passed as a pointer to the structure if you want to change the contents of the structure.
USAGE IS POINTER	*	16	A pointer.
PROCEDURE-POINTER	pointer to function	16	A 16-byte pointer to a procedure.
USAGE IS INDEX	int	4	A 4-byte integer.
REDEFINES	union.element	n	An element of a union.
OCCURS	data_type[n]	n*(length of data_type)	An array.
USAGE IS COMP-1	float	4	A 4-byte floating-point.
USAGE IS COMP-2	double	8	An 8-byte floating-point.
Not supported.	long double	8	An 8-byte long double.
Not supported.	enum	1, 2, 4	Enumeration.

Sharing External Data with an ILE C Program or Procedure

External data can be shared between an ILE COBOL program and an ILE C program. In order for the data item to be shared, it must be defined with the same name and description in the ILE COBOL program and the ILE C program. If the external data that is to be shared between the ILE C program and the ILE COBOL program is defined with different sizes in the programs, then the size of the external data item will be forced to that defined with the `extern` keyword in the ILE C program.

The ILE COBOL program and the ILE C program must be statically bound together in order for the external data item to be shared.

In the ILE COBOL program, the data item must be described with the `EXTERNAL` clause in the Working Storage Section. See “Sharing EXTERNAL Data” on page 237 or refer to the section on the `EXTERNAL` clause in the ILE C for a further description of how external data is used in an ILE COBOL program.

In the ILE C program, the data item must be declared using the `extern` keyword. Refer to *IBM Rational Development Studio for i: ILE C/C++ Programmer’s Guide* for a detailed description of the `extern` keyword.

Returning Control from an ILE C Program or Procedure

The `return` keyword in ILE C causes control to be returned to the caller. If the ILE C `return` keyword returns something other than `void`, the ILE COBOL `CALL` statement must have a returning phrase. In addition, the data type and length of the item returned from ILE C must match the data type and length of the `RETURNING` phrase identifier of the COBOL call statement.

When return is issued from an ILE C program, it may cause the ILE activation group in which the called ILE C program is running to end. If the ILE C program was defined to run in a *NEW activation group then when return is issued near a hard control boundary, the activation group in which the ILE C program was running is ended. If the ILE C program was defined to run in a *CALLER activation group or a named activation group then when return is issued, the activation group in which the ILE C program was running remains active. A subsequent call to the ILE C program in this activation group will find the ILE C program in its last used state.

The `exit(n)` function can cause control to be returned to the nearest control boundary. An exception condition can cause an exception handler to be invoked or cause control to be returned to the nearest control boundary.

When the ILE C program is running in a different named activation group than the calling ILE COBOL program, `exit(n)` or an unhandled exception cause the following to happen. If `exit(n)` or an unhandled exception occur near a hard control boundary, the activation group in which the ILE C program is running is ended. If they occur near a soft control boundary, the activation group remains active. If an unhandled exception ends the activation group in which the ILE C program is running, the CEE9901 escape message is raised in the calling ILE COBOL program's activation group.

When the ILE C program and the calling ILE COBOL program are running in the same activation group, `exit(n)` or an unhandled exception cause the following to happen. If `exit(n)` or an unhandled exception occur near a hard control boundary, the activation group, including the ILE COBOL program, is ended. If an unhandled exception ends the activation group in which both the ILE C program and the ILE COBOL program are running, the CEE9901 escape message is issued to the program prior to the hard control boundary. If `exit(n)` or an unhandled exception occur near a soft control boundary, all programs and procedures, including the ILE COBOL program, between the ILE C program from which the `exit(n)` is made and the program at the soft control boundary, are ended.

Control is returned to your ILE COBOL program at the next statement after the CALL statement if the called program ends without an exception. If the called program ends with an exception then an exception handling procedure identified in your ILE COBOL program may be invoked. Refer to Chapter 16, "ILE COBOL Error and Exception Handling," on page 369 for a full description of transferring control to an exception handling procedure.

The called program can also send an escape message past the calling ILE COBOL program skipping it altogether. In this case, the invocation of the ILE COBOL program is canceled. Canceling the invocation is similar to returning from the ILE COBOL program.

Examples of an ILE C Procedure Call from an ILE COBOL Program

Each example consists of an ILE COBOL program that calls an ILE C procedure.

Sample Code for ILE C Procedure Call Example 1

Example 1 has two code samples:

C1 QCSRC

An ILE C procedure that is bound to the ILE COBOL program.

CBL1 QCBLLSRC

An ILE COBOL procedure that calls the bound ILE C procedure.

The sample code for C1 QCSRC is shown in Figure 70.

```
/* C1 QCSRC --- ILE C Procedure */
#include <stdio.h>;
#include <stdlib.h>;
void C1(char *result)
{
    *(result+9) = '*';
    *(result+10) = '#';
    return;
}
```

Figure 70. Source code for C1 QCSRC

The sample code for CBL1 QCBLLSRC is shown in Figure 71.

```
*****
* cbl1 qcbllsrc
*
* Description:
*
*   COBOL source with ILE C procedure call.
*
*****
Identification Division.
  Program-Id.    cbl1.
  Author.       Author's Name.
  Installation.  IBM Toronto Lab
  Date-Written. July 14, 1998.
  Date-Compiled. Will be replaced by compile date.
Environment Division.
  Configuration Section.
    Source-Computer.  IBM-ISERIES.
    Object-Computer.  IBM-ISERIES.
    Special-Names.
INPUT-OUTPUT SECTION.

File-Control.
Data Division.
  Working-Storage Section.
    01 RESULT-STRING  PIC X(20)      VALUE ALL "X".

Procedure Division.

TEST1-INIT.
  DISPLAY RESULT-STRING.
  CALL PROCEDURE "C1" USING RESULT-STRING.
  DISPLAY RESULT-STRING.
  STOP RUN.

*-----
* Output before call
* XXXXXXXXXXXXXXXXXXXX
* Output after call
* XXXXXXXX*#XXXXXXXXXX
```

Figure 71. Source code for CBL1 QCBLLSRC

Sample Code for ILE C Procedure Call Example 2

Example 2 has two code samples:

CPROC1 QCSRC

An ILE C procedure that is bound to the ILE COBOL program.

VARG1 QCBLLSRC

An ILE COBOL procedure that calls the bound ILE C procedure.

The sample code for CPROC1 QSRC is shown in Figure 72.

```
/* CPROC1 QCSRC --- ILE C Procedure */
#include <stdio.h>

int inner(va_list);

int CPROC1(int p0, ...)
{
    int rc;
    va_list args;
    va_start(args,p0);
    rc = inner(args);
    va_end(args);
    return rc;
}

int inner(va_list v) {
    int p1,p2,p3=0;
    int p4,p5,p6=0;
    int p7,p8,p9=0;
    p1 = va_arg(v,int);
    p2 = va_arg(v,int);
    p3 = va_arg(v,int);
    p4 = va_arg(v,int);
    p5 = va_arg(v,int);
    p6 = va_arg(v,int);
    p7 = va_arg(v,int);
    p8 = va_arg(v,int);
    p9 = va_arg(v,int);
    printf("In inner with p1=%d p2=%d p3=%d\n",
           p1, p2, p3);
    printf("In inner with p4=%d p5=%d p6=%d\n",
           p4, p5, p6);
    printf("In inner with p7=%d p8=%d p9=%d\n",
           p7, p8, p9);
    return(p1 + p2 + p3 + p4 + p5 + p6 + p7 + p8 + p9);
}
```

Figure 72. Source code for CPROC1 QSRC

The sample code for VARG1 QCBLLSRC is shown in Figure 73 on page 321.

```

*****
* cb11 qcb1lesrc
*
* Description:
*
*   COBOL source with ILE C procedure call.
*
*****
      IDENTIFICATION DIVISION.

      PROGRAM-ID.       VARG1.

      *** This program demonstrates how to call a C procedure
      *** using variable-length argument list.

      AUTHOR.
      INSTALLATION.    IBM Toronto Lab.
      DATE-WRITTEN.
      DATE-COMPILED.

      ENVIRONMENT DIVISION.

      CONFIGURATION SECTION.

      SOURCE-COMPUTER. IBM-ISERIES.
      OBJECT-COMPUTER. IBM-ISERIES.
      SPECIAL-NAMES. LINKAGE PROCEDURE FOR "CPROC1"
                     USING ALL DESCRIBED.

      INPUT-OUTPUT SECTION.
      FILE-CONTROL.
      DATA DIVISION.
      FILE SECTION.
      WORKING-STORAGE SECTION.

      01 PARM0          PIC S9(9) BINARY VALUE 0.
      01 PARM1          PIC S9(9) BINARY VALUE 1.
      01 PARM2          PIC S9(9) BINARY VALUE 2.
      01 PARM3          PIC S9(9) BINARY VALUE 3.
      01 PARM4          PIC S9(9) BINARY VALUE 4.
      01 PARM5          PIC S9(9) BINARY VALUE 5.
      01 PARM6          PIC S9(9) BINARY VALUE 6.
      01 PARM7          PIC S9(9) BINARY VALUE 7.
      01 PARM8          PIC S9(9) BINARY VALUE 8.
      01 PARM9          PIC S9(9) BINARY VALUE 9.
      01 RC1            PIC S9(9) BINARY VALUE 0.

      PROCEDURE DIVISION.

      MAIN.

          CALL PROCEDURE "CPROC1" USING BY VALUE
              PARM0
              PARM1
              PARM2
              PARM3
              PARM4
              PARM5
              PARM6
              PARM7
              PARM8
              PARM9

          RETURNING INTO RC1.
      GOBACK.

```

Figure 73. Source code for VARG1 QCBLLSRC

Creating and Running the ILE C Procedure Call Examples

To create and run ILE C procedure call example 1, follow these steps:

1. Create one ILE COBOL module and one ILE C module.
 - To create the ILE COBOL module CBL1, type
CRTCBMOD MODULE(CBL1) SRCFILE(*CURLIB/QCBLLESRC)
 - To create the ILE C module C1, type
CRTCMOD MODULE(C1) SRCFILE(*CURLIB/QCSRC)
2. Create a program using the two modules
CRTPGM PGM(CBL1) MODULE(*CURLIB/CBL1 *CURLIB/C1)
3. Call the program
CALL PGM(*CURLIB/CBL1)

To create and run ILE C procedure call example 2, follow these steps:

1. Create one ILE COBOL module and one ILE C module:
 - To create the ILE COBOL module VARG1, type
CRTCBMOD MODULE(VARG1) SRCFILE(*CURLIB/QCBLLESRC)
 - To create the ILE C module CPROC1, type
CRTCMOD MODULE(CPROC1) SRCFILE(*CURLIB/QCSRC)
2. Create a program using the two modules:
CRTPGM PGM(VARG1) MODULE(*CURLIB/VARG1 *CURLIB/CPROC1)
3. Call the program:
CALL PGM(*CURLIB/VARG1)

Example of an ILE C Program Call from an ILE COBOL Program

This example consists of an ILE COBOL program that calls an ILE C program.

Sample Code for ILE C Program Call Example

The example has two code samples:

C2 QCSRC

An ILE C program.

CBL2 QCBLESRC

An ILE COBOL program with an ILE C program call.

The sample code for C2 QCSRC is shown in Figure 74.

```
/* C2 QCSRC --- ILE C Program */
#include <stdio.h>;
#include <stdlib.h>;
void main(int argc, char *argv[])
{
    *(argv[1]+9) = '*';
    *(argv[1]+10) = '#';
    return;
}
```

Figure 74. Source code for C2 QCSRC

The sample code for CBL2 QCBLLESRC is shown in Figure 75.

```
*****
* cbl2 qcbllsrc
*
* Description:
*
*   COBOL source with ILE C program call.
*
*****
Identification Division.
  Program-Id.    cbl2.
  Author.       Author's Name.
  Installation.  IBM Toronto Lab
  Date-Written.  July 14, 1998.
  Date-Compiled. Will be replaced by compile date.
Environment Division.
  Configuration Section.
    Source-Computer.  IBM-ISERIES.
    Object-Computer.  IBM-ISERIES.
    Special-Names.
INPUT-OUTPUT SECTION.

File-Control.
Data Division.
  Working-Storage Section.
    01 RESULT-STRING  PIC X(20)          VALUE ALL "X".

Procedure Division.

TEST1-INIT.
  DISPLAY RESULT-STRING.
  CALL "C2" USING BY REFERENCE RESULT-STRING.
  DISPLAY RESULT-STRING.
  STOP run.

*-----
* Output before call
* XXXXXXXXXXXXXXXXXXXX
* Output after call
* XXXXXXXX*#XXXXXXXXXX
```

Figure 75. Source code for CBL2 QCBLLESRC

Creating and Running the ILE C Program Call Example

To create and run the ILE C program call example, follow these steps:

1. Create one ILE COBOL program and one ILE C program

- To create the ILE COBOL program CBL2, type
CRTBNDCBL PGM(CBL2) SRCFILE(*CURLIB/QCBLLESRC)
- To create the ILE C program C2, type
CRTBNDC PGM(C2) SRCFILE(*CURLIB/QCSRC)

2. Call the ILE COBOL program

```
CALL PGM(*CURLIB/CBL2)
```

Calling ILE RPG Programs and Procedures

An ILE COBOL program can call ILE RPG programs and procedures using dynamic program calls or static procedure calls.

When a dynamic program call is used to call an ILE RPG program, the ILE RPG program must be compiled and bound as a separate program object. When a static

procedure call is used to call an ILE RPG procedure, the ILE RPG procedure must first be compiled into a module object and then bound to the calling ILE COBOL program. Refer to the *IBM Rational Development Studio for i: ILE RPG Programmer's Guide* for a description of compiling and binding ILE RPG programs and procedures.

You call an ILE RPG program or procedure from an ILE COBOL program by using the *CALL literal* statement (where *literal* is the name of the ILE RPG program or procedure). To call the ILE RPG program or procedure, you write the *CALL literal* statement in the same way as you would if you were calling another ILE COBOL subprogram. See "Using Static Procedure Calls and Dynamic Program Calls" on page 220 for detailed information about how to write the *CALL* statement in your ILE COBOL program to call an ILE RPG program using dynamic program calls or static procedure calls.

You can also call an ILE RPG program from an ILE COBOL program by using the *CALL identifier* statement. See "Using *CALL identifier*" on page 223 for more information on *CALL identifier*.

Passing Data to an ILE RPG Program or Procedure

You can pass data to a called ILE RPG program or procedure by using *CALL...BY REFERENCE*, *CALL...BY VALUE*, or *CALL...BY CONTENT*. Refer to "Passing Data Using *CALL...BY REFERENCE*, *BY VALUE*, or *BY CONTENT*" on page 233 for a description of how to use *CALL...BY REFERENCE*, *CALL...BY VALUE* or *CALL...BY CONTENT*.

When data is passed to the ILE RPG program using *CALL...BY REFERENCE*, a pointer to the data item is placed into the argument list that is accepted by the ILE RPG program. When data is passed to the ILE RPG program using *CALL...BY CONTENT*, the value of the data item is copied to a temporary location and then a pointer containing the address of the copy's temporary location is placed into the argument list that is accepted by the ILE RPG program. For *CALL...BY VALUE*, the value of the item is placed into the argument list that is accepted by the ILE RPG program. *CALL...BY VALUE* can be used to call ILE RPG procedures but not ILE RPG program objects.

In your ILE COBOL program, you describe the arguments that you want to pass to the ILE RPG program or procedure, in the Data Division in the same manner as you describe other data items in the Data Division. Refer to "Passing and Sharing Data Between Programs" on page 232 for a description of how to describe the arguments that you want to pass.

In the called ILE RPG program, you describe the parameters that you want to receive from the ILE COBOL program using the *PARM* operation. Each receiving parameter is defined in a separate *PARM* operation. You specify the name of the parameter in the Result field. The Factor 1 and Factor 2 entries are optional and indicate variables or literals. The value from the Factor 1 field is transferred from the Result field entry when the call occurs. The value from the Factor 2 field is placed in the Result field entry upon return.

Another method of defining parameters in an ILE RPG program is to specify them in a prototype. Each parameter is defined on a separate definition specification. For parameters passed *BY REFERENCE*, no special keywords are necessary. For parameters passed *BY VALUE*, the *VALUE* keyword is used. Refer to the *IBM*

Rational Development Studio for i: ILE RPG Programmer's Guide for more information on how to describe the arguments in an ILE RPG program.

Data Type Compatibility between ILE RPG and ILE COBOL

ILE RPG and ILE COBOL have different data types. When you want to pass data between programs written in ILE RPG and ILE COBOL, you must be aware of these differences. Some data types in ILE RPG and ILE COBOL have no direct equivalent in the other language.

Table 23 shows the ILE COBOL data type compatibility with ILE RPG.

Table 23. ILE COBOL Data Type Compatibility with ILE RPG

ILE COBOL	ILE RPG I-Spec, D-Spec, or C-Spec	Length	Description
PIC X(n).	blank or A in data type column, n in length column, and blank in decimal position column	n	A character field where n=1 to 32 767
PIC 1 INDIC ..	*INxxxx	1	An indicator.
PIC S9(n) DISPLAY	S in data type column or blank in data type column, n in length column, and 0 in decimal position column	n	A zoned decimal.
PIC S9(n-p)V9(p) COMP-3	P in data type column, n in length column, and p in decimal position column	n/2 + 1	A packed decimal.
PIC S9(n-p)V9(p) PACKED-DECIMAL.	P in data type column, n in length column, and p in decimal position column	n/2 + 1	A packed decimal.
Not supported	I in data type column, 3 in length column, and 0 in decimal position column	1	A 1-byte signed integer with a range of -128 to 127
Not supported	U in data type column, 3 in length column, and 0 in decimal position column	1	A 1-byte unsigned integer with a range of 0 to 255
PIC S9(4) COMP-4 BINARY	B in data type column, 4 in length column, and 0 in decimal position column	2	A 2-byte signed integer with a range of -9999 to +9999.
PIC S9(4) BINARY with *NOSTDTRUNC PIC S9(4) COMP-5	I in data type column, 5 in length column, and 0 in decimal position column	2	A 2-byte signed integer with a range of -32768 to 32767
PIC 9(4) COMP-5	U in data type column, 5 in length column, and 0 in decimal position column	2	A 2-byte unsigned integer with a range of 0 to 65535
PIC S9(9) COMP-4 PIC S9(9) BINARY	B in data type column, 9 in length column, and 0 in decimal position column	4	A 4-byte signed integer with a range of -999999999 to +999999999.
PIC S9(9) BINARY with *NOSTDTRUNC PIC S9(9) COMP-5	I in data type column, 10 in length column, and 0 in decimal position column	4	A 4-byte signed integer with a range of -2147483648 to 2147483647
PIC 9(9) COMP-5	U in data type column, 10 in length column, and 0 in decimal position column	4	A 4-byte unsigned integer with a range of 0 to 4294967295

Table 23. ILE COBOL Data Type Compatibility with ILE RPG (continued)

ILE COBOL	ILE RPG I-Spec, D-Spec, or C-Spec	Length	Description
PIC S9(18) COMP-4 PIC S9(18) BINARY PIC S9(18) COMP-5	I in data type column, 20 in length column, and 0 in decimal position column	8	An 8-byte signed integer with a range of -9223372036854775808 to 9223372036854775807.
PIC 9(18) COMP-5	U in data type column, 20 in length column, and 0 in decimal position column	8	An 8-byte unsigned integer with a range of 0 to 18446744073709551615.
USAGE IS COMP-1	F in data type column, 4 in length column	4	A 4-byte internal floating-point field.
USAGE IS COMP-2	F in data type column, 8 in length column.	8	An 8-byte internal floating-point field.
05 VL-FIELD. 10 i PIC S9(4) COMP-4. 10 data PIC X(n).	A in data type column, n in length column. Keyword VARYING.	n+2	A variable length field where i is the intended length and n is the maximum length.
05 n PIC 9(9) COMP-4. 05 x redefines n PIC X(4).	U in data type column, 4 in length column. To manipulate move to unsigned field in data structure overlaid by character array and use bit operations on each byte.	4	Bitfields can be manipulated using hex literals.
01 record 05 field1... 05 field2...	data structure	n	A structure. Structures passed should be passed as a pointer to the structure if you want to change the contents of the structure.
USAGE IS POINTER	* in data type column	16	A pointer.
PROCEDURE-POINTER	* in data type column and keyword PROCPTR	16	A 16-byte pointer to a procedure.
USAGE IS INDEX	I in data type column, length is 10, 0 in decimal position	4	A 4-byte integer.
REDEFINES	data structure subfield	n	An element of a union.
OCCURS	Keyword OCCURS or keyword DIM	n*(length of data_type)	An array.
FORMAT DATE	D in data type column	n	A date data type.
FORMAT TIME	T in data type column	n	A time data type.
FORMAT TIMESTAMP	Z in data type column	n	A timestamp data type.
PIC G(n)	G in data type column	n*2	A graphic (double-byte) data type.
Not supported	C in data type column	n*2	A UCS-2 (Universal Character Set) data type.

Returning Control from an ILE RPG Program or Procedure

When returning from an ILE RPG main procedure, the RETURN operation code causes control to be returned to the caller. If, prior to executing the RETURN operation code, the SETON operation code is used to set the LR indicator, the called ILE RPG program is reset to its initial state upon return to the caller. Otherwise, the called ILE RPG program is left in its last used state.

When returning from an ILE RPG subprocedure, the RETURN operation code causes control to be returned to the caller. If the procedure returns a value, the returned value is specified in the extended factor 2 of the RETURN operation. If the subprocedure returns a value, the COBOL CALL statement should have a RETURNING phrase.

Note: The LR indicator has no meaning when returning from a subprocedure.

Control is returned to your ILE COBOL program at the next statement after the CALL statement if the called program ends without an exception. If the called program ends with an exception then control is returned to the exception handling procedure identified in your ILE COBOL program. Refer to Chapter 16, “ILE COBOL Error and Exception Handling,” on page 369 for a full description of transferring control to an exception handling procedure.

The called program can also send an escape message past the calling ILE COBOL program skipping it altogether. In this case, the invocation of the ILE COBOL program is canceled. Canceling the invocation is similar to returning from the ILE COBOL program.

Calling ILE CL Programs and Procedures

An ILE COBOL program can call ILE CL programs and procedures using dynamic program calls or static procedure calls.

When a dynamic program call is used to call an ILE CL program, the ILE CL program must be compiled and bound as a separate program object. When a static procedure call is used to call an ILE CL procedure, the ILE CL procedure must first be compiled into a module object and then bound to the calling ILE COBOL program. Refer to the *CL Programming* for a description of compiling and binding ILE CL programs and procedures.

You call an ILE CL program or procedure from an ILE COBOL program by using the CALL *literal* statement (where *literal* is the name of the ILE CL program or procedure). To call the ILE CL program or procedure, you write the CALL *literal* statement in the same way as you would if you were calling another ILE COBOL subprogram. See “Using Static Procedure Calls and Dynamic Program Calls” on page 220 for detailed information about how to write the CALL statement in your ILE COBOL program to call an ILE CL program using dynamic program calls or static procedure calls.

You can also call an ILE CL program from an ILE COBOL program by using the CALL *identifier* statement. See “Using CALL identifier” on page 223 for more information on CALL *identifier*.

Passing Data to an ILE CL Program or Procedure

You can pass data to a called ILE CL program or procedure by using CALL...BY REFERENCE or CALL...BY CONTENT. Refer to “Passing Data Using CALL...BY REFERENCE, BY VALUE, or BY CONTENT” on page 233 for a description of how to use CALL...BY REFERENCE or CALL...BY CONTENT.

When data is passed to the ILE CL program using CALL...BY REFERENCE, a pointer to the data item is placed into the argument list that is accepted by the ILE CL program. When data is passed to the ILE CL program using CALL...BY CONTENT, the value of the data item is copied to a temporary location and then a

pointer containing the address of the copy's temporary location is placed into the argument list that is accepted by the ILE CL program.

In your ILE COBOL program, you describe the arguments that you want to pass to the ILE CL program or procedure, in the Data Division in the same manner as you describe other data items in the Data Division. Refer to "Passing and Sharing Data Between Programs" on page 232 for a description of how to describe the arguments that you want to pass.

In the called ILE CL program, you describe the parameters that you want to receive from the ILE COBOL program on the PARM parameter of the PGM statement. The order in which the receiving parameters are listed in the PARM parameter must be the same as the order in which they are listed on the CALL statement in the ILE COBOL program. In addition to the position of the parameters, you must pay careful attention to their length and type. Parameters listed in the called ILE CL program must be declared as the same length and type as they are in the calling ILE COBOL program.

You use DCL statements to describe the receiving parameters in the called ILE CL program. The order of the DCL statements is not important. Only the order in which the parameters are specified on the PGM statement determines what variables are received. The following example shows how parameters are described in the called ILE CL program.

```
PGM PARM(&P1 &P2);
DCL VAR(&P1); TYPE(*CHAR) LEN(32)
DCL VAR(&P2); TYPE(*DEC) LEN(15 5)
.
.
.
RETURN
ENDPGM
```

Refer to the *CL Programming* for a description of how to describe parameters in an ILE CL program.

Data Type Compatibility between ILE CL and ILE COBOL

ILE CL and ILE COBOL have different data types. When you want to pass data between programs written in ILE CL and ILE COBOL, you must be aware of these differences. Some data types in ILE CL and ILE COBOL have no direct equivalent in the other language.

Table 24 shows the ILE COBOL data type compatibility with ILE CL.

Table 24. ILE COBOL Data Type Compatibility with ILE CL

ILE COBOL	ILE CL	Length	Description
PIC X(n).	TYPE(*CHAR) LEN(n)	n	A character field where n=1 to 32 766.
01 flag PIC 1. 88 flag-on VALUE B'1'. 88 flag-off VALUE B'0'.	TYPE(*LGL)	1	Holds '1' or '0'.
PIC S9(n-p)V9(p) COMP-3.	TYPE(*DEC) LEN(n p)	n/2+1	A packed decimal. Maximum value for n=15. Maximum value for p=9.
PIC S9(n-p)V9(p) PACKED-DECIMAL.	TYPE(*DEC) LEN(n p)	n/2+1	A packed decimal. Maximum value for n=15. Maximum value for p=9.

Table 24. ILE COBOL Data Type Compatibility with ILE CL (continued)

ILE COBOL	ILE CL	Length	Description
USAGE IS COMP-1	Not Supported.	4	A 4-byte internal floating-point.
USAGE IS COMP-2	Not Supported.	8	An 8-byte internal floating-point.

Returning Control from an ILE CL Program or Procedure

The RETURN command in ILE CL causes control to be returned to the caller.

Control is returned to your ILE COBOL program at the next statement after the CALL statement if the called program ends without an exception. If the called program ends with an exception then control is returned to the exception handling procedure identified in your ILE COBOL program. Refer to Chapter 16, “ILE COBOL Error and Exception Handling,” on page 369 for a full description of transferring control to an exception handling procedure.

The called program can also send an escape message past the calling ILE COBOL program skipping it altogether. In this case, the invocation of the ILE COBOL program is canceled. Canceling the invocation is similar to returning from the ILE COBOL program.

Calling OPM Languages

Programs written in OPM languages such as OPM COBOL/400 or OPM RPG/400® can only be called from ILE COBOL using dynamic program calls. OPM programs cannot be statically bound to ILE COBOL programs. If you attempt to call an OPM program using a static procedure call, you will receive an error message. At bind time, you will receive a warning message from the binder for an unresolved reference to the static procedure call. If you disregard the warning message and create the ILE program object, you will get an exception when the static procedure call is attempted at run time.

You call an OPM program from an ILE COBOL program by using the CALL *literal* statement (where *literal* is the name of the OPM program). To call the OPM program, you write the CALL *literal* statement in the same way as you would if you were calling another ILE COBOL subprogram using a dynamic program call. See “Performing Dynamic Program Calls using CALL *literal*” on page 222 for detailed information about how to write the CALL statement in your ILE COBOL program to call an OPM program using dynamic program calls.

You can also call an OPM program from an ILE COBOL program by using the CALL *identifier* statement. See “Using CALL *identifier*” on page 223 for more information on CALL *identifier*.

Programs written in OPM languages can only be run in the Default Activation Group (*DFTACTGRP).

You can call an ILE COBOL program from an OPM program by using the same call semantics as you would for calling another OPM program.

External data cannot be shared between OPM programs and ILE COBOL programs.

Calling OPM COBOL/400 Programs

OPM COBOL/400 programs can only be run in the Default Activation Group (*DFTACTGRP). ILE COBOL programs can be run in the Default Activation Group (*DFTACTGRP), in *NEW ILE activation groups, and in named ILE activation groups.

Note: CRTPGM does not allow *DFTACTGRP to be explicitly specified in the ACTGRP parameter but it does allow *CALLER to be specified in the ACTGRP parameter. Specifying *CALLER in the ACTGRP parameter allows an ILE COBOL program called from an OPM COBOL/400 program (or any OPM program) to be run in the Default Activation Group. This is the only way for an ILE COBOL program to run in the Default Activation Group. An ILE COBOL program cannot be the hard control boundary in the Default Activation Group.

When a mixed language application of OPM COBOL/400 and ILE COBOL programs is run, the following scenario must be adhered to in order to most closely mimic an OPM COBOL/400 run unit:

- All participating programs must run in the Default Activation Group (*DFTACTGRP).
- The first COBOL program to be activated in the activation group must be an OPM COBOL/400 program.
- STOP RUN must be issued by an OPM COBOL/400 program or GOBACK must be issued by an OPM COBOL/400 main program, to end the run unit.
- An exception causing an implicit STOP RUN, if any, must be handled in such a way that the implicit STOP RUN is triggered by OPM COBOL/400.

For a mixed language application of OPM COBOL/400 and ILE COBOL programs running in the Default Activation Group, each ILE COBOL program is considered to be a non-COBOL program by the OPM COBOL/400 programs and each OPM COBOL/400 program is considered to be a non-COBOL program by the ILE COBOL programs. Also, each ILE COBOL program that is called by an OPM COBOL/400 program generates a soft control boundary by which the scope of the STOP RUN issued by the ILE COBOL program is bound.

When STOP RUN is issued by the ILE COBOL program, control is returned to the OPM COBOL/400 program without refreshing the state of the ILE COBOL program and the OPM COBOL/400 run unit is not ended. When STOP RUN is issued from an OPM COBOL/400 program, control is returned to the caller of the current main OPM COBOL/400 program and the OPM COBOL/400 run unit is ended.

For a mixed language application of OPM COBOL/400 and ILE COBOL programs where an ILE COBOL program is running in a *NEW or named ILE activation group and the OPM COBOL/400 program is running in the Default Activation Group, the effect of STOP RUN issued by the ILE COBOL program depends on whether the nearest control boundary is a hard control boundary or a soft control boundary. If it is a hard control boundary then control is returned to the caller of the hard control boundary and its *NEW or named ILE activation group is ended. If it is a soft control boundary then control is returned to the caller of the soft control boundary but the *NEW or named ILE activation group of the ILE COBOL program remains active.

Note: This scenario does not conform to an OPM COBOL/400 run unit.

Calling EPM Languages

Programs written in EPM languages such as EPM C/400, Pascal, and FORTRAN can be called from an ILE COBOL program through a CALL to QPXXCALL.

In the following example, an ILE COBOL program uses QPXXCALL to call a Pascal procedure.

```
5722WDS V5R4M0 060210 LN  IBM ILE COBOL          CBLGUIDE/COBTOPAS      ISERIES1  06/02/15 13:38:36      Page    2
                               S o u r c e
STMT PL SEQNBR -A 1 B..+....2....+....3....+....4....+....5....+....6....+....7..IDENTFCN  S COPYNAME  CHG DATE
 1  000100 IDENTIFICATION DIVISION.
 2  000200 PROGRAM-ID. COBTOPAS.
 3  000300 ENVIRONMENT DIVISION.
 4  000400 CONFIGURATION SECTION.
 5  000500 SOURCE-COMPUTER. IBM-ISERIES
 6  000600 OBJECT-COMPUTER. IBM-ISERIES
 7  000700 DATA DIVISION.
 8  000800 WORKING-STORAGE SECTION.
 9  000900 01 PARAMETER-LIST.
10  001000 05 ENTRY-NAME PIC X(100) VALUE "SQUARE".
11  001100 05 ENTRY-ID PIC X(10) VALUE "*MAIN".
12  001200 05 PROG-NAME PIC X(20) VALUE "MATH".
13  001300 05 A-REAL PIC S9(9) COMP-4 VALUE 0.
14  001400 05 CLEAN PIC S9(9) COMP-4 VALUE 0.
15  001500 05 INPT PIC S99 VALUE 0.
16  001600 PROCEDURE DIVISION.
    001700 MAINLINE.
17  001800 DISPLAY "ENTER AREA NUMBER:".
18  001900 ACCEPT INPT.
19  002000 MOVE INPT TO A-REAL.
20  002100 CALL "QPXXCALL" USING ENTRY-NAME
    002200 ENTRY-ID
    002300 PROG-NAME
    002400 A-REAL.
21  002500 DISPLAY A-REAL.
22  002600 CALL "QPXXDLTE" USING CLEAN.
23  002700 STOP RUN.
    002800
                               * * * * * E N D   O F   S O U R C E * * * * *
```

Figure 76. Calling a Pascal procedure from an ILE COBOL program.

```
segment MATH;
procedure SQUARE ( var X : integer ) ; external ;
procedure SQUARE;
begin
  X := X * X
end; .
```

Figure 77. Pascal entry-point that is to be called from an ILE COBOL program.

Pascal allows an ILE COBOL program to call a Pascal procedure as a subprogram. To do this, specify the Pascal procedure with the EXTERNAL directive (see Figure 77). In the above example, the first invocation the ENTRY-ID parameter of QPXXCALL will establish a Pascal Main Environment. You can use QPXXDLTE to clean up Pascal Reentrant and Main Environments. Passing zero in the CLEAN parameter to QPXXDLTE causes the current Pascal Main Environment to be deleted.

You can call an ILE COBOL program from an EPM program by using the same call semantics as you would for calling another EPM program.

External data cannot be shared between EPM programs and ILE COBOL programs.

Issuing a CL Command from an ILE COBOL Program

You can issue a CL command from an ILE COBOL program through a dynamic program call to QCMDEXC.

In the following example program, the CALL to QCMDEXC (at sequence number 000160) results in the processing of the Add Library List Entry (ADDLIBLE) CL command (at sequence number 000110). The successful completion of the CL command results in the addition of the library, COBOLTEST, to the library list.

```
5722WDS V5R4M0 060210 LN  IBM ILE COBOL      CBLGUIDE/CMDXMPLE  ISERIES1  06/02/15 13:40:28  Page   2
                               S o u r c e
STMT PL SEQNBR -A 1 B..+...2...+...3...+...4...+...5...+...6...+...7..IDENTFCN  S COPYNAME  CHG DATE
 1      000100 IDENTIFICATION DIVISION.
 2      000200 PROGRAM-ID. CMDXMPLE.
 3      000300 ENVIRONMENT DIVISION.
 4      000400 CONFIGURATION SECTION.
 5      000500 SOURCE-COMPUTER. IBM-ISERIES
 6      000600 OBJECT-COMPUTER. IBM-ISERIES
 7      000700 DATA DIVISION.
 8      000800 WORKING-STORAGE SECTION.
 9      000900 01 PROGRAM-VARIABLES.
10      001000 05 CL-CMD          PIC X(33)
11      001100          VALUE "ADDLIBLE COBOLTEST".
12      001200 05 PACK-VAL      PIC 9(10)V9(5) COMP-3
13      001300          VALUE 18.
14      001400 PROCEDURE DIVISION.
15      001500 MAINLINE.
16      001600 CALL "QCMDEXC" USING CL-CMD PACK-VAL.
17      001700 STOP RUN.
18      001800
                               * * * * *  E N D   O F   S O U R C E  * * * * *
```

Figure 78. Issuing a CL command from an ILE COBOL program.

For more information about QCMDEXC, see the *CL Programming*.

Including Structured Query Language (SQL) Statements in Your ILE COBOL Program

The syntax for SQL statements embedded in an ILE COBOL source program is:

Imbedding SQL Statements

►►—EXEC SQL—*sql-statement*—END-EXEC.—◄◄

If the member type for the source program is SQLCBLLE when the COBOL syntax checker encounters an SQL statement, the statement is passed to the SQL syntax checker. If an error is detected, a message is returned.

If an SQL statement is encountered, and if the member type is not SQLCBLLE a message is returned indicating that an ILE COBOL statement is in error.

If there are errors in the embedded SQL statement as well as errors in the preceding ILE COBOL statements, the SQL error message will only be displayed after the preceding COBOL errors are corrected.

You can create SQL programs for use with your ILE COBOL programs. The SQL cursor used by your ILE COBOL program may be scoped to either the module object or the activation group. You specify the SQL cursor scoping through the CLOSQCSR parameter of the Create SQL Program commands (CRTSQLxxx).

```
# For more information about SQL statements and SQL cursors, refer to the DB2
# Universal Database for AS/400 section of the Database and File Systems category in the
# i5/OS Information Center at this Web site - http://www.ibm.com/systems/i/
# infocenter/.
```

Calling an ILE API to Retrieve Current Century

The following example, Figure 79, shows how to retrieve a four-digit year using the ILE bindable API, Get Current Local Time(CEELOCT). This API retrieves the current local time in three formats. The third format is the Gregorian date, the first four characters of which are the year.

The next section, "Using Intrinsic Functions or the ACCEPT Statement to Retrieve Current Century," discusses how you can also use several of the intrinsic functions, and the ACCEPT statement to do the same thing.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. DATE1.
* Example program to get the 4 digit year in ILE COBOL for ISERIES
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. IBM-ISERIES
OBJECT-COMPUTER. IBM-ISERIES
DATA DIVISION.
WORKING-STORAGE SECTION.
01 date-vars.
   05 lilian                pic 9(9) usage binary.
   05 lilian-time-stamp     usage comp-2.
   05 gregorian-date.
       10 greg-year         pic x(4).
       10 greg-month        pic x(2).
       10 greg-day          pic x(2).
       10 greg-time         pic x(9).
       10 filler            pic x(6).
PROCEDURE DIVISION.
TEST-PARA.
   call procedure "CEELOCT" using
       lilian lilian-time-stamp
       gregorian-date.
   display "date is " gregorian-date.
   display "year " greg-year.
   display "month " greg-month.
STOP RUN.
```

Figure 79. Example of Retrieving Current Century.

Using Intrinsic Functions or the ACCEPT Statement to Retrieve Current Century

You can also use one of the following intrinsic functions to retrieve the current century or a 4-digit year:

CURRENT-DATE

Returns the current 4-digit year, as well as other information about the current date and time.

DATE-OF-INTEGERS

Takes an integer date, the number of days since December 31, 1600, and returns a Gregorian date with a 4-digit year in the form YYYYMMDD.

DAY-OF-INTEGER

Takes an integer date, the number of days since December 31, 1600, and returns a Julian date with a 4-digit year in the form YYYYDDD.

DATE-TO-YYYYMMDD

Converts a 2-digit year Gregorian date to a 4-digit year Gregorian date.

DAY-TO-YYYYDDD

Converts a 2-digit year Julian date to a 4-digit year Julian date.

EXTRACT-DATE-TIME

Extracts a part of the date or time information contained in a date, time or timestamp item. The year is extracted as a 4-digit year or a 1-digit century.

YEAR-TO-YYYY

Converts a 2-digit year to a 4-digit year.

The FROM DATE YYYYMMDD phrase of the ACCEPT statement can be used to retrieve a 4-digit year Gregorian date from the system. The FROM DAY YYYYDDD phrase of the ACCEPT statement can be used to retrieve a 4-digit year Julian date from the system.

Calling IFS API

You can call the IFS API from an ILE COBOL program. The IFS API should be checked to ensure whether operational descriptors are needed in the COBOL program or not. If needed, the operational descriptors should be specified in the SPECIAL-NAMES paragraph.

Figure 80 is an example of a call to IFS API:

```
SPECIAL NAMES.  
LINKAGE TYPE PROCEDURE FOR "open" USING ALL DESCRIBED.  
  
... more declaration and procedure statements  
  
CALL "open" USING ...
```

Figure 80. Calling IFS API

Chapter 14. Using Pointers in an ILE COBOL Program

You can use a **pointer** (a data item in which address values can be stored) within an ILE COBOL program when you want to pass and receive addresses of data items, ILE procedures, or program objects.

This chapter describes:

- How to define and redefine pointers
- How to initialize pointers
- How to read and write pointers
- How to manipulate data using pointers.

Defining Pointers

You can define pointers in two ways:

- A pointer to a data item. This pointer is defined with the **USAGE POINTER** clause. The resulting data item is called a **pointer data item**.
- A pointer to an ILE COBOL program, an ILE procedure, or a program object. This pointer is defined with the **USAGE PROCEDURE-POINTER** clause. The resulting data item is called a **procedure-pointer data item**.

On the AS/400 system, pointers are 16 bytes long.

ILE COBOL pointer data items point to system space objects. One part of the pointer describes its attributes, such as to which AS/400 space object it is pointing. Another part of the pointer contains the offset into the AS/400 system space object.

ILE COBOL procedure-pointer data items are AS/400 **open pointers**. Open pointers have the ability to be used as other types of AS/400 pointers. In particular, when an ILE COBOL procedure-pointer data item is set to a program object, the open pointer will contain an AS/400 system pointer. When an ILE COBOL procedure-pointer data item is set to an ILE procedure, the open pointer will contain an AS/400 procedure pointer.

A pointer a 16-byte elementary item that can be compared for equality, or used to set the value of other pointer items.

A pointer data item can be used only in:

- A SET statement (Formats 5 and 7 only)
- A relation condition
- The USING phrase of a CALL statement, or the Procedure Division header
- The operand for the LENGTH OF and ADDRESS OF special registers.

A procedure-pointer data item can be used only in:

- A SET statement (Format 6 only)
- A relation condition
- The USING phrase of a CALL statement, or the Procedure Division header
- The operand for the LENGTH OF and ADDRESS OF special registers
- The CALL statement as a target.

If pointers are used in a relational condition, the only valid operators are equal to, or not equal to.

Because pointer data items are not simply binary numbers on the AS/400 system, manipulating pointers as integers does not work.

Pointer data items are defined explicitly with the USAGE IS POINTER clause, and are implicit when using an ADDRESS OF special register or the ADDRESS OF an item.

If a group item is described with the USAGE IS POINTER clause, the elementary items within the group item are pointer data items. The group itself is not a pointer data item, and cannot be used in the syntax where a pointer data item is allowed. If a group item is described with the USAGE PROCEDURE-POINTER clause, the same rules apply. The USAGE clause of an elementary item cannot contradict the USAGE clause of a group to which the item belongs.

Pointers can be part of a group that is referred to in a MOVE statement or an input/output statement; however, if a pointer is part of a group, there is no conversion of pointer values to another form of internal representation when the statement is executed.

Using ILE C and other languages, you can declare pointers to teraspace memory. ILE C requires a special compile-time option to address this type of storage, but ILE COBOL can always address this storage if compiled with a target release of V4R4M0 or later. For more information on pointers in teraspace, see the *ILE Concepts* publication.

Pointer Alignment

Pointers can be defined at any level (except 88) in the File, Working-Storage, or Linkage sections of a program.

When a pointer is referenced on the i5/OS, it must be on a 16-byte storage
boundary. **Pointer alignment** refers to the ILE COBOL compiler's process of
positioning pointer items within a group item to offsets that are multiples of 16
bytes from the beginning of the record. If a pointer item is not on a 16-byte
boundary, a pointer alignment exception is sent to the ILE COBOL program. In
general, pointer alignment exceptions occur in the Linkage Section, where it is up
to the user to align these items.

In the File and Working-Storage sections, the compiler ensures that this exception does not occur by adding implicit FILLER items. Every time an implicit FILLER item is added by the compiler, a warning is issued. In the Linkage Section, no implicit FILLER items are added by the compiler; however, warnings are issued indicating how many bytes of FILLER would have been added had the group item appeared in the File or Working-Storage sections.

You can define a data item as a pointer by specifying the USAGE IS POINTER clause or the USAGE IS PROCEDURE-POINTER clause as shown in the following example:

```
ID DIVISION.  
PROGRAM-ID. PROGA.  
WORKING-STORAGE SECTION.  
    77 APTR USAGE POINTER.  
    77 APROC-PTR USAGE PROCEDURE-POINTER.  
    01 AB.  
        05 BPTR USAGE POINTER.  
        05 BVAR PIC S9(3) PACKED-DECIMAL.  
LINKAGE SECTION.
```

```

01 AVAR.
   05 CVAR PIC X(30).
PROCEDURE DIVISION.
   SET APTR TO ADDRESS OF AVAR.
   SET APROC-PTR TO ENTRY "PROGA".

```

In the above example, AVAR is an 01-level data item, so the ADDRESS OF AVAR is the ADDRESS OF special register. Because a special register is an actual storage area, the SET statement moves the contents of ADDRESS OF AVAR into pointer data item APTR.

In the above example, if the SET statement used ADDRESS OF CVAR, no special register exists. Instead, the pointer data item APTR is assigned the calculated address of CVAR.

In the above example, the second SET statement is setting procedure-pointer data item APROC-PTR to the outermost ILE COBOL program "PROGA".

Writing the File Section and Working-Storage Section for Pointer Alignment

In the File Section and Working-Storage Section, all 01-level items and 77-level items (and some 66-level items) are placed on 16-byte boundaries.

Within a group structure, pointers must also occur on a 16-byte boundary. To ensure this, the ILE COBOL compiler adds FILLER items immediately before the pointers. To avoid these FILLER items, you should place pointers at the beginning of a group item.

If the pointer is part of a table, the first item in the table is placed on a 16-byte boundary. To ensure that all subsequent occurrences of the pointer fall on a 16-byte boundary, a FILLER item is added to the end of the table if necessary.

An example of pointer alignment follows:

```

WORKING-STORAGE SECTION.
  77 APTR USAGE POINTER.
  01 AB.
     05 ALPHA-NUM PIC X(10).
     05 BPTR USAGE PROCEDURE-POINTER.
  01 EF.
     05 ARRAY-1 OCCURS 3 TIMES.
        10 ALPHA-NUM-TWO PIC X(14).
        10 CPTR USAGE POINTER.
        10 ALPHA-NUM-THREE PIC X(5).

```

In the above example, APTR is a pointer data item. The 77-level item, therefore, is placed on a 16-byte boundary. The group item AB is an 01-level item and is automatically placed on a 16-byte boundary. Within the group item AB, BPTR is not on a 16-byte boundary. To align it properly, the compiler inserts a 6-byte FILLER item after ALPHA-NUM. Finally, CPTR requires a FILLER of 2 bytes to align its first occurrence. Because ALPHA-NUM-THREE is only 5 bytes long, another 11-byte FILLER must be added to the end of ARRAY-1 to align all subsequent occurrences of CPTR.

When a pointer is defined in the File Section, and a file does not have blocking in effect, each 01-level item will be on a 16-byte boundary. If a file has blocking in effect, only the first record of a block is guaranteed to be on a 16-byte boundary.

Thus pointers should not be defined for files with blocking in effect. For more information on blocking, refer to “Unblocking Input Records and Blocking Output Records” on page 415.

Redefining Pointers

A pointer data item or procedure-pointer data item may be the subject or object of a REDEFINES clause.

When a pointer is the subject of a REDEFINES clause, the object data item must be on a 16-byte boundary. For example:

```
WORKING-STORAGE SECTION.  
01 AB.  
    05 ALPHA-NUM PIC X(16).  
    05 APTR REDEFINES ALPHA-NUM USAGE POINTER.  
    05 BPTR  USAGE POINTER.  
    05 CPTR REDEFINES BPTR USAGE POINTER.
```

In the above example, both APTR and CPTR are pointer data items that redefine 16-byte aligned items. In the following example, the redefined item would result in a severe compiler error:

```
WORKING-STORAGE SECTION.  
01 EF.  
    05 ALPHA-NUM PIC X(5).  
    05 HI.  
        10 ALPHA-NUM-TWO PIC X(11).  
        10 APTR  USAGE POINTER.  
    05 BPTR REDEFINES HI USAGE POINTER.
```

In the above example, APTR is aligned on a 16-byte boundary. That is, the ILE COBOL compiler did not need to add FILLER items to align APTR. The group item HI is not on a 16-byte boundary, and so neither is pointer data item BPTR. Because the ILE COBOL compiler cannot add FILLER items to place BPTR on a 16-byte boundary, a severe error will result.

In the following example, similar to the above, the ILE COBOL compiler is able to place the pointer data item on a 16-byte boundary:

```
WORKING-STORAGE SECTION.  
01 EF.  
    05 ALPHA-NUM PIC X(5).  
    05 HI.  
        10 ALPHA-NUM-TWO PIC X(11).  
        10 APTR  USAGE POINTER.  
        10 ALPHA-NUM-THREE PIC X(5).  
    05 KL REDEFINES HI.  
        10 BPTR USAGE POINTER.
```

In the above example, group item KL is not on a 16-byte boundary; however, the compiler adds an 11-byte FILLER before pointer data item BPTR to ensure that it falls on a 16-byte boundary.

Initializing Pointers Using the NULL Figurative Constant

The NULL figurative constant represents a value used to indicate that data items defined with USAGE IS POINTER, USAGE IS PROCEDURE-POINTER, ADDRESS OF, or the ADDRESS OF special register do not contain a valid address. For example:

```

WORKING-STORAGE SECTION.
  77 APTR  USAGE POINTER VALUE NULL.
PROCEDURE DIVISION.
  IF APTR = NULL THEN
    DISPLAY 'APTR IS NULL'
  END-IF.

```

In the above example, pointer APTR is set to NULL in the Working-Storage section. The comparison in the procedure division will be true and the display statement is executed.

On the AS/400 system, the initial value of a pointer data item or procedure-pointer data item with or without a VALUE clause of NULL, equals NULL.

Reading and Writing Pointers

Pointers can be defined in the File Section, and can be set and used as can any other Working-Storage pointer data items. There are, however, some restrictions:

- If a file has blocking in effect, only the first record of a block is guaranteed to be on a 16-byte boundary. Thus pointers should not be defined for files with blocking in effect.
- A record containing pointers can be written to a file; however, on subsequent reading of that record, the pointer data items and procedure-pointer data items equal NULL.

Using the LENGTH OF Special Register with Pointers

The LENGTH OF special register contains the number of bytes used by an identifier. It returns a value of 16 for a pointer data item or procedure-pointer data item.

You can use LENGTH OF in the Procedure Division anywhere a numeric data item having the same definition as the implied definition of the LENGTH OF special register is used; however, LENGTH OF cannot be used as a subscript or a receiving data item. LENGTH OF has the implicit definition:

```
USAGE IS BINARY, PICTURE 9(9)
```

The following example shows how you can use LENGTH OF with pointers:

```

WORKING-STORAGE SECTION.
  77 APTR  USAGE POINTER.
  01 AB.
    05 BPTR  USAGE PROCEDURE-POINTER.
    05 BVAR  PIC S9(3) PACKED-DECIMAL.
    05 CVAR  PIC S9(3) PACKED-DECIMAL.
PROCEDURE DIVISION.
  MOVE LENGTH OF AB TO BVAR.
  MOVE LENGTH OF BPTR TO CVAR.

```

In the above example, the length of group item AB is moved to variable BVAR. BVAR has a value of 20 because BPTR is 16 bytes long, and both variables BVAR and CVAR are 2 bytes long. CVAR receives a value of 16.

You can also use the LENGTH OF special register to set up data structures within user spaces, or to increment addresses received from another program. To see an example of a program that uses the LENGTH OF special register to define data structures within user spaces, refer to Figure 82 on page 343.

Setting the Address of Linkage Section Items

Generally, when one ILE COBOL program calls another, operands are passed from the calling ILE COBOL program to the called ILE COBOL program in the following manner:

- The calling program uses the CALL USING statement to pass operands to the called program, and
- The called program specifies the USING phrase in the Procedure Division header.

For each operand that is listed on the CALL USING statement in the calling ILE program, there must be a corresponding operand that is specified by the USING phrase in the Procedure Division of the called program.

When using the ADDRESS OF special register, you no longer need to ensure a one-to-one mapping between the USING phrases of the two programs. For those data items in the Linkage Section that are not specified in the USING phrase of the Procedure Division header, you can use a SET statement to specify the starting address of the data structure. Once the SET statement is run, the data item can be freely referred to since the address of the data item is already set. For an example of a SET statement used in this manner, refer to Figure 83 on page 344. In Figure 83 on page 344, **15** and **16** illustrates how the SET statement is used to set the starting address of the data structures *ls-header-record* and *ls-user-space* at the beginning of the user space.

Using ADDRESS OF and the ADDRESS OF Special Register

When you specify ADDRESS OF in an ILE COBOL program, the compiler determines whether to use the calculated address of a data item, referred to as ADDRESS OF, or the ADDRESS OF special register. The ADDRESS OF special register is the starting address of the data structure from which all calculated addresses are determined. Because the ADDRESS OF special register is the starting address of a structure, it must be an 01-level or 77-level data item. If you reference modify this data item, it is no longer the starting address of the data structure. It is a calculated address, or ADDRESS OF. If you are taking the ADDRESS OF an elementary item, and the ADDRESS OF of the 01-level item has been set to NULL, a pointer exception (MCH3601) results.

You cannot use the calculated ADDRESS OF where an item can be changed. Only the ADDRESS OF special register can be changed. For example, in Figure 83 on page 344, the SET statement at **17** uses the ADDRESS OF special register because it is an 01-level item. At **18**, ADDRESS OF is used because, although it is an 01-level item, it is reference-modified.

Using Pointers in a MOVE Statement

Elementary pointers cannot be moved using the MOVE statement; a SET statement must be used; however, pointers are implicitly moved when they are part of a group item.

When compiling a MOVE statement, the ILE COBOL compiler generates code to maintain (a pointer MOVE) or not maintain (a non-pointer MOVE) pointers within a group item.

A pointer MOVE is done when all of the following conditions are met:

1. The source or receiver of a MOVE statement contains a pointer

2. Both of the items are at least 16 bytes long
3. The data items are properly aligned
4. The data items are alphanumeric or group items.

Of the conditions listed above, determining if two data items are properly aligned can be the most difficult.

Note: A pointer MOVE is slower than using the SET statement to move a pointer.

Items must be on the same offset relative to a 16-byte boundary for a pointer MOVE to occur. (A warning is issued if this is not true.)

The following example shows three data structures, and the results when a MOVE statement is issued:

```

WORKING-STORAGE SECTION.
01 A.
   05 B          PIC X(10).
   05 C.
       10 D      PIC X(6).
       10 E      POINTER.

01 A2.
   05 B2        PIC X(6).
   05 C2.
       10 D2    PIC X(10).
       10 E2    POINTER.

01 A3.
   05 B3        PIC X(22).
   05 C3.
       10 D3    PIC X(10).
       10 E3    POINTER.

PROCEDURE DIVISION.
MOVE A to A2. 1
MOVE A to A3. 1
MOVE C to C2. 2
MOVE C2 to C3. 3

```

Figure 81. Using Pointers in a MOVE Statement

- 1** This results in a pointer MOVE because the offset of each group item to be moved is zero. Pointer integrity is maintained.
- 2** This results in a non-pointer MOVE, because the offsets do not match. The offset of group item C is 10, and the offset of group item C2 is 6. Pointer integrity is not maintained.
- 3** This results in a pointer MOVE, because the offset of group item C2 is 6, and the offset of C3 relative to a 16-byte boundary is also 6. (When the offset is greater than 16, the offset relative to a 16-byte boundary is calculated by dividing the offset by 16. The remainder is the relative offset. In this case, the offset was 22, which, when divided by 16, leaves a remainder, or relative offset, of 6.) Pointer integrity is maintained.

If a group item contains a pointer, and the ILE COBOL compiler cannot determine the offset relative to a 16-byte boundary, the ILE COBOL compiler issues a warning message, and the pointer move is attempted. However, pointer integrity may not be maintained. The ILE COBOL compiler cannot determine the offset if the item is defined in the Linkage

Section, or if the item is reference-modified with an unknown starting position. You must ensure that pointer alignment is maintained, or a machine check error may result.

The ILE COBOL compiler places all 01-level and 77-level items on a 16-byte boundary whether or not they contain pointers.

Using Pointers in a CALL Statement

When a pointer data item or procedure-pointer data item is passed in a CALL statement, the item is treated as all other USING items. In other words, when a pointer data item is passed BY REFERENCE (or BY CONTENT), a pointer to the pointer data item (or copy of the pointer data item) is passed to the called program. When a pointer data item is passed BY VALUE the contents of the pointer data item is placed on the call stack. Procedure-pointer data items are passed similarly.

Special consideration must be given when a CALL statement with the BY CONTENT phrase is used to pass pointers and group items containing pointers. This is similar to the case of a MOVE statement. For a CALL...BY CONTENT, an implicit MOVE of an item is done to create it in a temporary area. To ensure pointer alignment on this pointer MOVE, the ILE COBOL compiler or run time must determine the offset of the BY CONTENT item relative to the 16-byte boundary. For the best performance, the BY CONTENT item should be coded in such a way that the ILE COBOL compiler can determine this offset.

The ILE COBOL run time has to determine the offset of an item relative to a 16-byte boundary when the BY CONTENT item is:

- Reference modified with an unknown starting position, or
- Defined in the Linkage Section.

When an operand is reference-modified, the offset is the reference modification starting position minus one, plus the operand's offset within the data structure. When an operand is in the Linkage Section, its offset can be determined from the calling program.

To avoid pointer alignment problems, pass items using BY REFERENCE.

Adjusting the Value of Pointers

The following example shows you how to adjust the value of a pointer by increasing it UP BY or decreasing it DOWN BY an integer value. This method of changing the value of a pointer can be useful when you are accessing items in a table that is referenced by a pointer data item.

```
WORKING-STORAGE SECTION.  
  01 A.  
    05 ARRAY-USER-INFO OCCURS 300 TIMES.  
      10 USER-NAME PIC X(10).  
      10 USER-ID PIC 9(7).  
    01 ARRAY-PTR USAGE IS POINTER.  
LINKAGE SECTION.  
  01 USER-INFO.  
    05 USER-NAME LIKE USER-NAME OF ARRAY-USER-INFO.  
    05 USER-ID LIKE USER-ID OF ARRAY-USER-INFO.  
PROCEDURE DIVISION.  
  SET ARRAY-PTR TO ADDRESS OF ARRAY-USER-INFO(200).1  
  SET ADDRESS OF USER-INFO TO ARRAY-PTR.2
```

```

SET ARRAY-PTR UP BY LENGTH OF USER-INFO. 3
SET ADDRESS OF USER-INFO TO ARRAY-PTR. 4
MOVE "NEW NAME" TO USER-NAME OF USER-INFO. 5

```

Notes:

1. The first SET statement places the address of the 200th element of the ARRAY-USER-INFO array into the pointer ARRAY-PTR.
2. The second SET statement gives data item USER-INFO the same address as the 200th element of the ARRAY-USER-INFO array.
3. The third SET statement increments the address contained in pointer ARRAY-PTR by the length of one element of the array.
4. The fourth SET statement gives data item USER-INFO the same address as the 201st element of the ARRAY-USER-INFO array (in other words, up one element from the second SET statement).
5. This move is the same as:

```
MOVE "NEW NAME" to USER-NAME OF ARRAY-USER-INFO (201).
```

For a complete definition of the SET statement, refer to the *IBM Rational Development Studio for i: ILE COBOL Reference*.

Accessing User Spaces Using Pointers and APIs

The following example shows how you can use pointers to access user spaces and to chain records together.

POINTA is a program that reads customer names and addresses into a user space, and then displays the information in a list. The program assumes that the customer information exists in a file called POINTACU.

The customer address field is a variable-length field, to allow for lengthy addresses.

```

.....+.....1.....+.....2.....+.....3.....+.....4.....+.....5.....+.....6.....+.....7.....+.....8
A* THIS IS THE CUSTOMER INFORMATION FILE - POINTACUST
A
A
A      R FSCUST          TEXT('CUSTOMER MASTER RECORD')
A      FS_CUST_NO      8S00  TEXT('CUSTOMER NUMBER')
A      FS_CUST_NM      20    ALIAS(FS_CUST_NUMBER)
A      FS_CUST_NM      20    TEXT('CUSTOMER NAME')
A      FS_CUST_AD      100   ALIAS(FS_CUST_NAME)
A      FS_CUST_AD      100   TEXT('CUSTOMER ADDRESS')
A      FS_CUST_AD      100   ALIAS(FS_CUST_ADDRESS)
A      FS_CUST_AD      100   VARLEN

```

Figure 82. Example Using Pointers to Access User Spaces -- DDS

```

Source
STMT PL SEQNBR -A 1 B.+....2....+....3....+....4....+....5....+....6....+....7..IDENTFCN S COPYNAME CHG DATE
000100 PROCESS varchar 1
000200 ID DIVISION.
000300* This program reads in a file of variable length records
000400* into a user space. It then shows the records on
000500* the display.
000600 PROGRAM-ID. pointa.
000700 ENVIRONMENT DIVISION.
000800 CONFIGURATION SECTION.
000900 SPECIAL-NAMES. CONSOLE IS CRT,
001000 CRT STATUS IS ws-crt-status. 2
001100 INPUT-OUTPUT SECTION.
001200 FILE-CONTROL.
001300 SELECT cust-file ASSIGN TO DATABASE-pointacu
001400 ORGANIZATION IS SEQUENTIAL
001500 FILE STATUS IS ws-file-status.
001600 DATA DIVISION.
001700 FILE SECTION.
001800 FD cust-file.
001900 01 fs-cust-record.
002000* copy in field names turning underscores to dashes
002100* and using alias names
002200 COPY DDR-ALL-FORMATS-I OF pointacu.
18 +000001 05 POINTACU-RECORD PIC X(130). <-ALL-FMTS
+000002* I-O FORMAT:FSCUST FROM FILE POINTACU OF LIBRARY CBLGUIDE <-ALL-FMTS
+000003* CUSTOMER MASTER RECORD <-ALL-FMTS
19 +000004 05 FSCUST REDEFINES POINTACU-RECORD. <-ALL-FMTS
20 +000005 06 FS-CUST-NUMBER PIC S9(8). <-ALL-FMTS
+000006* CUSTOMER NUMBER <-ALL-FMTS
21 +000007 06 FS-CUST-NAME PIC X(20). <-ALL-FMTS
+000008* CUSTOMER NAME <-ALL-FMTS
22 +000009 06 FS-CUST-ADDRESS. 3 <-ALL-FMTS
+000010* (Variable length field) <-ALL-FMTS
23 +000011 49 FS-CUST-ADDRESS-LENGTH <-ALL-FMTS
+000012 PIC S9(4) COMP-4. <-ALL-FMTS
24 +000013 49 FS-CUST-ADDRESS-DATA <-ALL-FMTS
+000014 PIC X(100). <-ALL-FMTS
+000015* CUSTOMER ADDRESS <-ALL-FMTS
25 002300 WORKING-STORAGE SECTION.
26 002400 01 ws-file-status.
27 002500 05 ws-file-status-1 PIC X.
28 002600 88 ws-file-stat-good VALUE "0".
29 002700 88 ws-file-stat-at-end VALUE "1".
30 002800 05 ws-file-status-2 PIC X.
31 002900 01 ws-crt-status. 4
32 003000 05 ws-status-1 PIC 9(2).
33 003100 88 ws-status-1-ok VALUE 0.
34 003200 88 ws-status-1-func-key VALUE 1.
35 003300 88 ws-status-1-error VALUE 9.
36 003400 05 ws-status-2 PIC 9(2).
37 003500 88 ws-func-03 VALUE 3.
38 003600 88 ws-func-07 VALUE 7.
39 003700 88 ws-func-08 VALUE 8.
40 003800 05 ws-status-3 PIC 9(2).

```

Figure 83. Example Using Pointers to Access User Spaces (Part 1 of 8)

```

5722WDS V5R4M0 060210 LN IBM ILE COBOL CBLGUIDE/POINTA ISERIES1 06/02/15 13:43:25 Page 3
STMT PL SEQNBR -A 1 B...2...+...3...+...4...+...5...+...6...+...7..IDENTFCN S COPYNAME CHG DATE
41 003900 01 ws-params. 5
42 004000 05 ws-space-ptr POINTER. 6
43 004100 05 ws-space.
44 004200 10 ws-space-name PIC X(10) VALUE "MYSPACE".
45 004300 10 ws-space-lib PIC X(10) VALUE "QTEMP".
46 004400 05 ws-attr PIC X(10) VALUE "PF".
47 004500 05 ws-init-size PIC S9(5) VALUE 32000 BINARY.
48 004600 05 ws-init-char PIC X VALUE SPACE.
49 004700 05 ws-auth PIC X(10) VALUE "*ALL".
50 004800 05 ws-text PIC X(50) VALUE
004900 "Customer Information Records".
51 005000 05 ws-replace PIC X(10) VALUE "*YES".
52 005100 05 ws-err-data. 7
53 005200 10 ws-input-l PIC S9(6) BINARY VALUE 16.
54 005300 10 ws-output-l PIC S9(6) BINARY.
55 005400 10 ws-exception-id PIC X(7).
56 005500 10 ws-reserved PIC X(1).
005600
57 005700 77 ws-accept-data PIC X VALUE SPACE.
58 005800 88 ws-acc-blank VALUE SPACE.
59 005900 88 ws-acc-create-space VALUE "Y", "y".
60 006000 88 ws-acc-use-prv-space VALUE "N", "n".
61 006100 88 ws-acc-delete-space VALUE "Y", "y".
62 006200 88 ws-acc-save-space VALUE "N", "n".
006300
63 006400 77 ws-prog-indicator PIC X VALUE "G".
64 006500 88 ws-prog-continue VALUE "G".
65 006600 88 ws-prog-end VALUE "C".
66 006700 88 ws-prog-loop VALUE "L".
006800
67 006900 77 ws-line PIC 99.
007000* error message line
68 007100 77 ws-error-msg PIC X(50) VALUE SPACES.
007200* more address information indicator
69 007300 77 ws-plus PIC X.
007400* length of address information to display
70 007500 77 ws-temp-size PIC 9(2).
007600
71 007700 77 ws-current-rec PIC S9(4) VALUE 1.
72 007800 77 ws-old-rec PIC S9(4) VALUE 1.
73 007900 77 ws-old-space-ptr POINTER.
008000* max number of lines to display
74 008100 77 ws-displayed-lines PIC S99 VALUE 20.
008200* line on which to start displaying records
75 008300 77 ws-start-line PIC S99 VALUE 5.
008400* variables to create new record in space
76 008500 77 ws-addr-inc PIC S9(4) PACKED-DECIMAL.
77 008600 77 ws-temp PIC S9(4) PACKED-DECIMAL.
78 008700 77 ws-temp-2 PIC S9(4) PACKED-DECIMAL.
008800* pointer to previous record
79 008900 77 ws-cust-prev-ptr POINTER VALUE NULL.
80 009000 LINKAGE SECTION.
81 009100 01 ls-header-record. 8
82 009200 05 ls-hdr-cust-ptr USAGE POINTER.
009300* number of records read in from file

```

Figure 83. Example Using Pointers to Access User Spaces (Part 2 of 8)

```

STMT PL SEQNBR -A 1 B.+...2....+...3....+...4....+...5....+...6....+...7..IDENTFCN S COPYNAME CHG DATE
83 009400 05 ls-record-counter PIC S9(3) BINARY.
84 009500 05 FILLER PIC X(14). 9
85 009600 01 ls-user-space. 10
86 009700 05 ls-customer-rec.
009800* pointer to previous customer record
87 009900 10 ls-cust-prev-ptr USAGE POINTER.
88 010000 10 ls-cust-rec-length PIC S9(4) BINARY.
89 010100 10 ls-cust-name PIC X(20).
90 010200 10 ls-cust-number PIC S9(8).
010300* total length of this record including filler bytes
010400* to make sure next record on 16 byte boundary
91 010500 10 ls-cust-address-length PIC S9(4) BINARY.
92 010600 05 ls-cust-address-data PIC X(116).
010700
010800* Size of ls-user-space is 16 more than actually needed.
010900* This allows the start address of the next record
011000* to be established without exceeding the declared size.
011100* The size is 16 bigger to allow for pointer alignment.
011200
93 011300 PROCEDURE DIVISION.
011400* note no need for "USING" entry on PROC... DIV.
94 011500 DECLARATIVES.
011600 cust-file-para SECTION.
011700 USE AFTER ERROR PROCEDURE ON cust-file.
011800 cust-file-para-2.
95 011900 MOVE "Error XX on file pointacu" TO ws-error-msg.
96 012000 MOVE ws-file-status TO ws-error-msg(7:2).
012100 END DECLARATIVES.
012200
012300 main-program section.
012400 mainline.
012500* keep reading initial display until entered data correct
97 012600 SET ws-prog-loop TO TRUE.
98 012700 PERFORM initial-display THRU read-initial-display
012800 UNTIL NOT ws-prog-loop.
012900* if want to continue with program and want to create
013000* customer information area, fill the space with
013100* records from the customer file
99 013200 IF ws-prog-continue AND
013300 ws-acc-create-space THEN
100 013400 PERFORM read-customer-file
101 013500 MOVE 1 TO ws-current-rec
013600* set ptr to header record
102 013700 SET ADDRESS OF ls-header-record TO ws-space-ptr
013800* set to first customer record in space
103 013900 SET ADDRESS OF ls-user-space TO ls-hdr-cust-ptr
014000 END-IF.
104 014100 IF ws-prog-continue THEN
105 014200 PERFORM main-loop UNTIL ws-prog-end
014300 END-IF.
014400 end-program.
106 014500 PERFORM clean-up.
107 014600 STOP RUN.
014700
014800 initial-display. 11

```

Figure 83. Example Using Pointers to Access User Spaces (Part 3 of 8)

```

5722WDS V5R4M0 060210 LN IBM ILE COBOL CBLGUIDE/POINTA ISERIES1 06/02/15 13:43:25 Page 5
STMT PL SEQNBR -A 1 B..+...2...+...3...+...4...+...5...+...6...+...7..IDENTFCN S COPYNAME CHG DATE
108 014900 DISPLAY "Create Customer Information Area" AT 0118 WITH
    015000 BLANK SCREEN REVERSE-VIDEO
    015100 "Create customer information area (Y/N)=> <="
    015200 AT 1015
    015300 "F3=Exit" AT 2202.
109 015400 IF ws-error-msg NOT = SPACES THEN
110 015500 DISPLAY ws-error-msg at 2302 with beep highlight
111 015600 MOVE SPACES TO ws-error-msg
    015700 END-IF.
    015800
    015900 read-initial-display. 12
112 016000 ACCEPT ws-accept-data AT 1056 WITH REVERSE-VIDEO
    016100 ON EXCEPTION
113 016200 IF ws-status-1-func-key THEN
114 016300 IF ws-func-03 THEN
115 016400 SET ws-prog-end TO TRUE
    016500 ELSE
116 016600 MOVE "Invalid Function Key" TO ws-error-msg
    016700 END-IF
    016800 ELSE
117 016900 MOVE "Unknown Error" TO ws-error-msg
    017000 END-IF
    017100 NOT ON EXCEPTION
118 017200 IF ws-acc-create-space THEN
119 017300 PERFORM create-space THRU set-space-ptrs
120 017400 SET ws-prog-continue TO TRUE
    017500 ELSE
121 017600 IF ws-acc-use-prv-space THEN
122 017700 PERFORM get-space
123 017800 IF ws-space-ptr = NULL
124 017900 MOVE "No Customer Information Area" TO ws-error-msg
    018000 ELSE
125 018100 PERFORM set-space-ptrs
126 018200 SET ws-prog-continue TO TRUE
    018300 END-IF
    018400 ELSE
127 018500 MOVE "Invalid Character Entered" TO ws-error-msg
    018600 END-IF
    018700 END-IF
    018800 END-ACCEPT.
    018900
    019000 create-space.
128 019100 CALL "QUSCRTUS" USING ws-space, ws-attr, ws-init-size, 13
    019200 ws-init-char, ws-auth, ws-text,
    019300 ws-replace, ws-err-data.
    019400
    019500* checks for errors in creating the space could be added here
    019600
    019700 get-space.
129 019800 CALL "QUSPTRUS" USING ws-space, ws-space-ptr, ws-err-data. 14
    019900
    020000 set-space-ptrs.
    020100* set header record to beginning of space
130 020200 SET ADDRESS OF ls-header-record 15
    020300 ADDRESS OF ls-user-space 16

```

Figure 83. Example Using Pointers to Access User Spaces (Part 4 of 8)

```

STMT PL SEQNBR -A 1 B..+...2...+...3...+...4...+...5...+...6...+...7..IDENTFCN S COPYNAME CHG DATE
020400      TO ws-space-ptr.
020500* set first customer record after header record
131 020600      SET ADDRESS OF ls-user-space TO 17
020700      ADDRESS OF ls-user-space(LENGTH OF ls-header-record 18
020800      + 1:1).
020900* save ptr to first record in header record
132 021000      SET ls-hdr-cust-ptr TO ADDRESS OF ls-user-space.
021100
021200 delete-space.
133 021300      CALL "QUSDLTUS" USING ws-space, ws-err-data. 19
021400
021500 read-customer-file.
021600* read all records from customer file and move into space
134 021700      OPEN INPUT cust-file.
135 021800      IF ws-file-stat-good THEN
136 021900      READ cust-file AT END CONTINUE
022000      END-READ
138 022100      PERFORM VARYING ls-record-counter FROM 1 BY 1
022200      UNTIL not ws-file-stat-good
139 022300      SET ls-cust-prev-ptr TO ws-cust-prev-ptr
022400* Move information from file into space
140 022500      MOVE fs-cust-name TO ls-cust-name
141 022600      MOVE fs-cust-number TO ls-cust-number
142 022700      MOVE fs-cust-address-length TO ls-cust-address-length
143 022800      MOVE fs-cust-address-data(1:fs-cust-address-length)
022900      TO ls-cust-address-data(1:ls-cust-address-length)
023000* Save ptr to current record
144 023100      SET ws-cust-prev-ptr TO ADDRESS OF ls-user-space
023200* Make sure next record on 16 byte boundary
145 023300      ADD LENGTH OF ls-customer-rec 20
023400      ls-cust-address-length TO 1 GIVING ws-addr-inc
146 023500      DIVIDE ws-addr-inc BY 16 GIVING ws-temp
023600      REMAINDER ws-temp-2
147 023700      SUBTRACT ws-temp-2 FROM 16 GIVING ws-temp
023800* Save total record length in user space
148 023900      ADD ws-addr-inc TO ws-temp GIVING ls-cust-rec-length
149 024000      SET ADDRESS OF ls-user-space
024100      TO ADDRESS OF ls-user-space(ls-cust-rec-length + 1:1)
024200* Get next record from file
150 024300      READ cust-file AT END CONTINUE
024400      END-READ
024500      END-PERFORM
024600* At the end of the loop have one more record than really
024700* have
152 024800      SUBTRACT 1 FROM ls-record-counter
024900      END-IF.
153 025000      CLOSE cust-file.
025100
025200 main-loop. 21
025300* write the records to the display until F3 entered
154 025400      DISPLAY "Customer Information" AT 0124 WITH
025500      BLANK SCREEN REVERSE-VIDEO
025600      "Cust Customer Name Customer"
025700      AT 0305
025800      " Address"

```

Figure 83. Example Using Pointers to Access User Spaces (Part 5 of 8)

```

5722WDS V5R4M0 060210 LN IBM ILE COBOL CBLGUIDE/POINTA ISERIES1 06/02/15 13:43:25 Page 7
STMT PL SEQNBR -A 1 B.+...2...+...3...+...4...+...5...+...6...+...7..IDENTFCN S COPYNAME CHG DATE
025900 "Number" AT 0405
026000 "F3=Exit" AT 2202.
026100* if a pending error put on the display
155 026200 IF ws-error-msg NOT = SPACES THEN
156 026300 DISPLAY ws-error-msg at 2302 with beep highlight
157 026400 MOVE SPACES TO ws-error-msg
026500 END-IF.
026600* if in the middle of the list put F7 on the display
158 026700 IF ws-current-rec > 1 THEN 22
159 026800 DISPLAY "F7=Back" AT 2240
026900 END-IF.
027000* save the current record
160 027100 MOVE ws-current-rec TO ws-old-rec.
161 027200 SET ws-old-space-ptr TO ADDRESS OF ls-user-space. 23
027300* move each record to the display
162 027400 PERFORM VARYING ws-line FROM ws-start-line BY 1
027500 UNTIL ws-line > ws-displayed-lines or
027600 ws-current-rec > ls-record-counter
027700* if address is greater than display width show "+"
163 027800 IF ls-cust-address-length > 40 THEN
164 027900 MOVE "+" TO ws-plus
165 028000 MOVE 40 TO ws-temp-size
028100 ELSE
166 028200 MOVE ls-cust-address-length TO ws-temp-size
167 028300 MOVE SPACE TO ws-plus
028400 END-IF
168 028500 DISPLAY ls-cust-number at line ws-line column 5
028600 ls-cust-name ls-cust-address-data with
028700 size ws-temp-size ws-plus at line
028800 ws-line column 78
028900* get next record in the space
169 029000 ADD 1 TO ws-current-rec
170 029100 SET ADDRESS OF ls-user-space
029200 TO ADDRESS OF ls-user-space
029300 (ls-cust-rec-length + 1:1)
029400 END-PERFORM.
029500* if can go forward put F8 on the display
171 029600 IF ws-current-rec < ls-record-counter THEN 22
172 029700 DISPLAY "F8=Forward" AT 2250
029800 END-IF.
029900* check to see if continue, exit, or get next records or
030000* previous records
173 030100 SET ws-acc-blank to TRUE.
174 030200 ACCEPT ws-accept-data WITH SECURE 24
030300 ON EXCEPTION
175 030400 IF ws-status-1-func-key THEN
176 030500 IF ws-func-03 THEN
177 030600 SET ws-prog-end TO TRUE
030700 ELSE
178 030800 IF ws-func-07 THEN
179 030900 PERFORM back-screen
031000 ELSE
180 031100 IF ws-func-08 THEN
181 031200 PERFORM forward-screen
031300 ELSE

```

Figure 83. Example Using Pointers to Access User Spaces (Part 6 of 8)

```

STMT PL SEQNBR -A 1 B..+...2...+...3...+...4...+...5...+...6...+...7..IDENTFCN S COPYNAME CHG DATE
182 031400 MOVE "Invalid Function Key" TO ws-error-msg
183 031500 MOVE ws-old-rec TO ws-current-rec
184 031600 SET ADDRESS OF ls-user-space TO ws-old-space-ptr
031700 END-IF
031800 END-IF
031900 ELSE
185 032000 MOVE "Unknown Error" TO ws-error-msg
186 032100 MOVE ws-old-rec TO ws-current-rec
187 032200 SET ADDRESS OF ls-user-space TO ws-old-space-ptr
032300 END-IF
032400 NOT ON EXCEPTION
188 032500 MOVE ws-old-rec TO ws-current-rec
189 032600 SET ADDRESS OF ls-user-space TO ws-old-space-ptr
032700 END-ACCEPT.
032800
032900 clean-up.
033000* do clean up for program
033100* keep reading end display until entered data correct
190 033200 SET ws-prog-loop to TRUE.
191 033300 SET ws-acc-blank to TRUE.
192 033400 PERFORM final-display THRU read-final-display 25
033500 UNTIL NOT ws-prog-loop.
033600
033700 final-display.
193 033800 DISPLAY "Delete Customer Information Area" AT 0118 WITH 26
033900 BLANK SCREEN REVERSE-VIDEO
034000 "Delete customer information area (Y/N)=> <="
034100 AT 1015
034200 "F3=Exit" AT 2202.
194 034300 IF ws-error-msg NOT = SPACES THEN
195 034400 DISPLAY ws-error-msg at 2302 with beep highlight
196 034500 MOVE SPACES TO ws-error-msg
034600 END-IF.
034700
034800 read-final-display.
197 034900 ACCEPT ws-accept-data AT 1056 WITH REVERSE-VIDEO
035000 ON EXCEPTION
198 035100 IF ws-status-1-func-key THEN
199 035200 IF ws-func-03 THEN
200 035300 SET ws-prog-end TO TRUE
035400 ELSE
201 035500 MOVE "Invalid Function Key" TO ws-error-msg
035600 END-IF
035700 ELSE
202 035800 MOVE "Unknown Error" TO ws-error-msg
035900 END-IF
036000 NOT ON EXCEPTION
203 036100 IF ws-acc-delete-space THEN
204 036200 PERFORM delete-space
205 036300 SET ws-prog-continue TO TRUE
036400 ELSE
206 036500 IF ws-acc-save-space THEN
207 036600 SET ws-prog-continue TO TRUE
036700 ELSE
208 036800 MOVE "Invalid Character Entered" TO ws-error-msg

```

Figure 83. Example Using Pointers to Access User Spaces (Part 7 of 8)

```

5722WDS V5R4M0 060210 LN  IBM ILE COBOL          CBLGUIDE/POINTA      ISERIES1 06/02/15 13:43:25   Page    9
STMT PL SEQNBR -A 1 B.+...2....+...3....+...4....+...5....+...6....+...7..IDENTFCN  S COPYNAME  CHG DATE
      036900          END-IF
      037000          END-IF
      037100          END-ACCEPT.
      037200
      037300 back-screen.  27
209   037400          IF ws-old-rec <= 1 THEN
210   037500              MOVE "Top of customer records" TO ws-error-msg
211   037600              MOVE ws-old-rec TO ws-current-rec  28
212   037700              SET ADDRESS OF ls-user-space TO ws-old-space-ptr
      037800          ELSE
213   037900              MOVE ws-old-rec TO ws-current-rec  28
214   038000              SET ADDRESS OF ls-user-space TO ws-old-space-ptr
215   038100              PERFORM VARYING ws-line FROM ws-start-line BY 1
      038200                  UNTIL ws-line > ws-displayed-lines or
      038300                      ws-current-rec <= 1
      038400* Back up one record at a time
216   038500              SET ws-cust-prev-ptr TO ls-cust-prev-ptr  29
217   038600              SET ADDRESS OF ls-user-space TO ws-cust-prev-ptr
218   038700              SUBTRACT 1 FROM ws-current-rec
      038800          END-PERFORM
      038900          END-IF.
      039000
      039100 forward-screen.  30
      039200* if current record greater or equal to the max records
      039300* print error, have reached max records
219   039400          IF ws-current-rec >= ls-record-counter
220   039500              MOVE "No more customer records" TO ws-error-msg
221   039600              MOVE ws-old-rec TO ws-current-rec
222   039700              SET ADDRESS OF ls-user-space TO ws-old-space-ptr
      039800          ELSE
223   039900              MOVE ws-current-rec TO ws-old-rec
224   040000              SET ws-old-space-ptr TO ADDRESS OF ls-user-space
      040100          END-IF.
      040200
          * * * * *  E N D   O F   S O U R C E   * * * * *

```

Figure 83. Example Using Pointers to Access User Spaces (Part 8 of 8)

- 2 CRT STATUS IS specifies a data name into which a status value is placed after the termination of an extended ACCEPT statement. In this example, the STATUS key value is used to determine which function key was pressed.
- 3 *fs-cust-address* is a variable-length field. To see meaningful names here rather than FILLER, specify *VARCHAR for the CVTOPT parameter of the CRTCBMOD or CRTBNDCBL commands, or VARCHAR in the PROCESS statement, as shown in 1. For more information about variable-length fields, refer to “Declaring Data Items Using SAA Data Types” on page 438.
- 4 CRT STATUS as mentioned in 2 is defined here.
- 5 The *ws-params* structure contains the parameters used when calling the APIs to access user spaces.
- 6 *ws-space-ptr* defines a pointer data item set by the API QUSPTRUS. This points to the beginning of the user space, and is used to set the addresses of items in the Linkage Section.
- 7 *ws-err-data* is the structure for the error parameter for the user space APIs. Note that the *ws-input-l* is zero, meaning that any exceptions are signalled to the program, and not passed in the error code parameter. For more information on error code parameters, refer to the *CL and APIs* section of the *Programming* category in the **i5/OS Information Center** at this Web site -<http://www.ibm.com/systems/i/infocenter/>.
- 8 The first data structure (*ls-header-record*) to be defined in the user space.
- 9 FILLER is used to maintain pointer alignment, because it makes *ls-header-record* a multiple of 16 bytes long.

#

- 10** The second data structure (*ls-user-space*) to be defined in the user space.
- 11** *initial-display* shows the Create Customer Information Area display.
- 12** *read-initial-display* reads the first display, and determines if the user chooses to continue or end the program. If the user continues the program by pressing Enter, then the program checks *ws-accept-data* to see if the customer information area is to be created.
- 13** QUSCRTUS is an API used to create user spaces.
- 14** QUSPTRUS is an API used to return a pointer to the beginning of a user space.
- 15** Maps the first data structure (*ls-header-record*) over the beginning of the user space.
- 16** Maps the second data structure (*ls-user-space*) over the beginning of the user space.
- 17** Uses ADDRESS OF special register
- 18** Uses ADDRESS OF, not the ADDRESS OF special register, because it is reference modified.
- 19** QUSDLTUS is an API used to delete a user space.
- 20** The following four arithmetic statements calculate the total length of each record, and ensure that each record is a multiple of 16 bytes in length.
- 21** *main-loop* puts up the Customer Information display.
- 22** These statements determine if the program should display function keys F7 and F8.
- 23** Saves a pointer to the first customer record on the display.
- 24** This ACCEPT statement waits for input from the Customer Information display. Based on the function key pressed, it calls the appropriate paragraph to display the next set of records (*forward-screen*), or the previous set of records (*back-screen*), or sets an indicator to end the routine if F3 is pressed.
- 25** The clean up routine displays the Delete Customer Information Area display until an appropriate key is pressed.
- 26** This statement puts up the Delete Customer Information Area display.
- 27** Each record contains a pointer to the previous customer record. The ADDRESS OF special register points to the current customer record. By changing the ADDRESS OF special register, the current customer record is changed.

back-screen moves the current record pointer backward one record at a time **29**, by moving the pointer to the previous customer record into the pointer to the current customer record (ADDRESS OF). Before moving backward one record at a time, the program sets the current customer record to the first record currently displayed **28**.
- 30** *forward-screen* sets *ws-old-space-ptr* (which points to the first record in the display) to point to the current record (which is after the last record displayed.)

A user space always begins on a 16-byte boundary, so the method illustrated here ensures that **all** records are aligned. *ls-cust-rec-length* is also used to chain the records together.

When you run POINTA, you see the following displays:

```
CMDSTR                               Start Commands
Select one of the following:
Commands
  1. Start QSH                               QSH
  2. Start RPC Binder Daemon                 RPCBIND
  4. Start AppDict Services/400             STRADS
  7. Start AFP Utilities                     STRAFPU
  8. Start Advanced Print Function          STRAPF

 10. Start BEST/1 Planner                   STRBEST
 11. Start BGU                             STRBGU
 12. Start Calendar Service                 STRCALSRV
 13. Start COBOL Debug                     STRCBLDBG
 14. Start CICS/400                        STRCICS
                                           More...

Selection or command
===>call pointa

F3=Exit  F4=Prompt  F9=Retrieve  F12=Cancel  F16=Major menu
(C) COPYRIGHT IBM CORP. 1980, 1998.
Output file POINTSCREE created in library HORNER.      +
```

Create Customer Information Area

Create customer information area (Y/N)=> y <=

F3=Exit

Customer Information

Cust Number	Customer Name	Customer Address	
00000001	Bakery Unlimited	30 Bake Way, North York	
00000002	Window World	150 Eglinton Ave E., North York, Ontario	
00000003	Jons Clothes	101 Park St, North Bay, Ontario, Canada	
00000004	Pizza World	254 Main Street, Toronto, Ontario	+
00000005	Marv's Auto Body	9 George St, Peterborough, Ontario, Cana	+
00000006	Jack's Snacks	23 North St, Timmins, Ontario, Canada	
00000007	Video World	14 Robson St, Vancouver, B.C, Canada	
00000008	Pat's Daycare	8 Kingston Rd, Pickering, Ontario, Canad	+
00000009	Mary's Pies	3 Front St, Toronto, Ontario, Canada	
00000010	Carol's Fashions	19 Spark St, Ottawa, Ontario, Canada	
00000011	Grey Optical	5 Lundy's Lane, Niagara Falls, Ont. Cana	+
00000012	Fred's Forage	33 Dufferin St, Toronto, Ontario, Canada	+
00000013	Dave's Trucking	15 Water St, Guelph, Ontario, Canada	
00000014	Doug's Music	101 Queen St. Toronto, Ontario, Canada	+
00000015	Anytime Copiers	300 Warden Ave, Scarborough, Ontario, Ca	+
00000016	Rosa's Ribs	440 Avenue Rd, Toronto, Ontario, Canada	

F3=Exit F8=Forward


```

CMDSTR                               Start Commands
Select one of the following:
Commands
  1. Start QSH                               QSH
  2. Start RPC Binder Daemon                RPCBIND
  4. Start AppDict Services/400            STRADS
  7. Start AFP Utilities                    STRAFP
  8. Start Advanced Print Function          STRAFP
  10. Start BEST/1 Planner                  STRBEST
  11. Start BGU                             STRBGU
  12. Start Calendar Service                STRCALSRV
  13. Start COBOL Debug                     STRCBLDBG
  14. Start CICS/400                        STRCICS
                                           More...

Selection or command
===> endcpyscn

F3=Exit  F4=Prompt  F9=Retrieve  F12=Cancel  F16=Major menu
(C) COPYRIGHT IBM CORP. 1980, 1998.

```

Processing a Chained List Using Pointers

A typical application for using pointer data items is in processing a chained list (a series of records where each one points to the next).

For this example, picture a chained list of data that is composed of individual salary records. Figure 84 shows one way to visualize how these records are linked in storage:

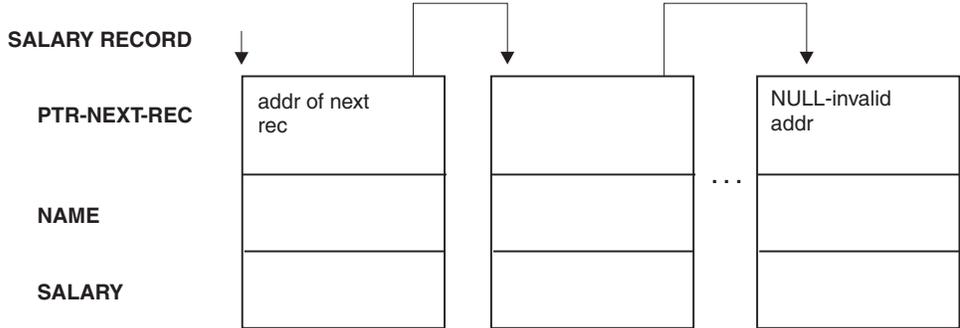


Figure 84. Representation of a Chained List Ending with NULL

The first item in each record (except for the last record) points to the next record. The first item in the last record, in order to indicate that it is the last record, contains a null value instead of an address.

The high-level logic of an application that processes these records might look something like this:

```

OBTAIN ADDRESS OF FIRST RECORD IN CHAINED LIST FROM ROUTINE
CHECK FOR END OF THE CHAINED LIST
DO UNTIL END OF THE CHAINED LIST
  PROCESS RECORD
  GO ON TO THE NEXT RECORD
END

```

Figure 85 on page 356 contains an outline of the processing program, CHAINLST, used in this example of processing a chained list.

```

5722WDS V5R4M0 060210 LN  IBM ILE COBOL          CBLGUIDE/CHAINLST      ISERIES1  06/02/15 13:45:02      Page      2
                               S o u r c e
STMT PL SEQNBR -A 1 B.+....2....+....3....+....4....+....5....+....6....+....7..IDENTFCN  S COPYNAME  CHG DATE
 1      000100 IDENTIFICATION DIVISION.
 2      000200 PROGRAM-ID. CHAINLST.
 3      000300 ENVIRONMENT DIVISION.
 4      000400 DATA DIVISION.
        000500*
 5      000600 WORKING-STORAGE SECTION.
 6      000700 77 PTR-FIRST          POINTER VALUE IS NULL.
 7      000800 77 DEPT-TOTAL        PIC 9(4) VALUE IS 0.
        000900*
 8      001000 LINKAGE SECTION.
 9      001100 01 SALARY-REC.
10      001200 05 PTR-NEXT-REC      POINTER.
11      001300 05 NAME              PIC X(20).
12      001400 05 DEPT              PIC 9(4).
13      001500 05 SALARY            PIC 9(6).
14      001600 01 DEPT-X            PIC 9(4).
        001700*
15      001800 PROCEDURE DIVISION USING DEPT-X.
        001900 CHAINLST-PROGRAM SECTION.
        002000 MAINLINE.
        002100*
        002200* FOR EVERYONE IN THE DEPARTMENT RECEIVED AS DEPT-X,
        002300* GO THROUGH ALL OF THE RECORDS IN THE CHAINED LIST BASED ON THE
        002400* ADDRESS OBTAINED FROM THE PROGRAM CHAINANC
        002500* AND ACCUMULATE THE SALARIES.
        002600* IN EACH RECORD, PTR-NEXT-REC IS A POINTER TO THE NEXT RECORD
        002700* IN THE LIST; IN THE LAST RECORD, PTR-NEXT-REC IS NULL.
        002800* DISPLAY THE TOTAL.
        002900*
16      003000 CALL "CHAINANC" USING PTR-FIRST
17      003100 SET ADDRESS OF SALARY-REC TO PTR-FIRST
        003200*
18      003300 PERFORM WITH TEST BEFORE UNTIL ADDRESS OF SALARY-REC = NULL
19      003400     IF DEPT = DEPT-X THEN
20      003500         ADD SALARY TO DEPT-TOTAL
        003600     END-IF
21      003700     SET ADDRESS OF SALARY-REC TO PTR-NEXT-REC
        003800     END-PERFORM
        003900*
22      004000 DISPLAY DEPT-TOTAL
23      004100 GOBACK.
        004200
                               * * * * *  E N D   O F   S O U R C E  * * * * *

```

Figure 85. ILE COBOL Program for Processing a Chained List

Passing Pointers between Programs and Procedures

To obtain the address of the first SALARY-REC record area, the CHAINLST program calls the program CHAINANC:

```
CALL "CHAINANC" USING PTR-FIRST
```

PTR-FIRST is defined in WORKING-STORAGE in the calling program (CHAINLST) as a pointer data item:

```
WORKING-STORAGE SECTION.
77 PTR-FIRST          POINTER VALUE IS NULL.
```

Upon return from the call to CHAINANC, PTR-FIRST contains the address of the first record in the chained list.

PTR-FIRST is initially defined as having a null value as a logic check. If an error occurs with the call, and PTR-FIRST never receives the value of the address of the first record in the chain, a null value remains in PTR-FIRST and, according to the logic of the program, the records will not be processed.

NULL is a figurative constant used to assign the value of a non-valid address to pointer items. It can be used in the VALUE IS NULL clause, in the SET statement, and as an operand in a relation condition with a pointer.

The Linkage Section of the calling program contains the description of the records in the chained list. It also contains the description of the department code that is passed through the USING phrase of the CALL statement.

```
LINKAGE SECTION.  
01 SALARY-REC.  
    05 PTR-NEXT-REC    POINTER.  
    05 NAME            PIC X(20).  
    05 DEPT            PIC 9(4).  
    05 SALARY          PIC 9(6).  
01 DEPT-X             PIC 9(4).
```

To base the record description SALARY-REC on the address contained in PTR-FIRST, use a SET statement:

```
CALL "CHAINANC" USING PTR-FIRST  
SET ADDRESS OF SALARY-REC TO PTR-FIRST
```

Check for the End of the Chained List

The chained list in this example is set up so that the last record contains an address that is not valid. To do this, the pointer data item in the last record would be assigned the value NULL.

A pointer data item can be assigned the value NULL in three ways:

- A pointer data item can be defined with a VALUE IS NULL clause in its data definition.
- NULL can be the sending field in a SET statement.
- The initial value of a pointer data item with or without a VALUE clause of NULL equals NULL.

In the case of a chained list in which the pointer in the last record contains a null value, the code to check for the end of the list would be:

```
IF PTR-NEXT-REC = NULL  
  
:  
(logic for end of chain)
```

If you have not reached the end of the list, process the record and move on to the next record.

In the program CHAINLST, this test for the end of the chained list is accomplished with a “do while” structure:

```
PERFORM WITH TEST BEFORE UNTIL ADDRESS OF SALARY-REC = NULL  
IF DEPT = DEPT-X  
    THEN ADD SALARY TO DEPT-TOTAL  
    ELSE CONTINUE  
END-IF  
SET ADDRESS OF SALARY-REC TO PTR-NEXT-REC  
END-PERFORM
```

Processing the Next Record

To move on to the next record, set the address of the record in the Linkage Section to be equal to the address of the next record. This is accomplished through the pointer data item sent as the first field in SALARY-REC:

```
SET ADDRESS OF SALARY-REC TO PTR-NEXT-REC
```

Then repeat the record-processing routine, which will process the next record in the chained list.

Incrementing Addresses Received from Another Program

The data passed from a calling program might contain header information that you want to ignore (for example, in data received from a CICS/400® application that is not migrated to the command level).

Because pointer data items are not numeric, you cannot directly perform arithmetic on them. You can, however, use the SET verb to increment the passed address in order to bypass header information.

You could set up the Linkage Section as follows:

```
LINKAGE SECTION.  
01 RECORD-A.  
   05 HEADER          PIC X(16).  
   05 REAL-SALARY-REC PIC X(30).  
  
:  
01 SALARY-REC.  
   05 PTR-NEXT-REC    POINTER.  
   05 NAME            PIC X(20).  
   05 DEPT            PIC 9(4).  
   05 SALARY          PIC 9(6).
```

Within the Procedure Division, base the address of SALARY-REC on the address of REAL-SALARY-REC:

```
SET ADDRESS OF SALARY-REC TO ADDRESS OF REAL-SALARY-REC
```

SALARY-REC is now based on the address of RECORD-A + 16.

Passing Entry Point Addresses with Procedure-Pointers

You can use procedure-pointer data items, defined with the USAGE IS PROCEDURE-POINTER clause, to pass the entry address of a program in a format required by certain ILE callable services.

For example, to have a user-written error handling routine take control when an exception condition occurs during program execution, you must first pass the entry address of an ILE procedure, such as an outermost ILE COBOL program, to CEEHDLR, a condition management ILE callable service, to have it registered.

Procedure-pointer data items can be set to contain the entry address for the following types of programs:

- An outermost ILE COBOL program
- An ILE procedure written in another ILE language
- An ILE program object or an OPM program object.

Note: A procedure-pointer data item cannot be set to the address of a nested ILE COBOL program.

A procedure-pointer data item can only be set using Format 6 of the SET statement.

For a complete definition of the USAGE IS PROCEDURE-POINTER clause and the SET statement, refer to the *IBM Rational Development Studio for i: ILE COBOL Reference*.

Chapter 15. Preparing ILE COBOL Programs for Multithreading

In the i5/OS environment, programs may run within the **threads** of processes. ILE
COBOL supports multithreaded execution by means of the THREAD PROCESS
statement option (see ""THREAD Option"" on page 59). In order to understand
this chapter's discussion of ILE COBOL support for multithreading, you need to be
familiar with the following terms:

Job On the i5/OS, a job represents a process. The operating system and
multithreading applications can handle execution flow within a job.
Multiple jobs can run concurrently, and programs running within a job can
share resources. A job is the container for the memory and resources of the
program.

Thread

Within a job, an application can initiate one or more threads. Within a thread, control is transferred between executing programs.

Run-unit

On the i5/OS, a run-unit represents the program activation group. A run
unit can contain multiple threads. When a COBOL run-unit ends in a
multithreaded environment, the job also ends. Within a run-unit, ILE
COBOL programs can call non-ILE COBOL programs, and vice versa.

Program Invocation Instance

Within a thread, control is transferred between separate ILE COBOL and non-ILE COBOL programs. For example, an ILE COBOL program can CALL another ILE COBOL program or an ILE C program. Each separately invoked (as in, CALLED) program is a program invocation instance. Program invocation instances of a particular program might exist in multiple threads within a given job.

The following illustration shows the relationships between jobs, threads, run-units, and program invocation instances:

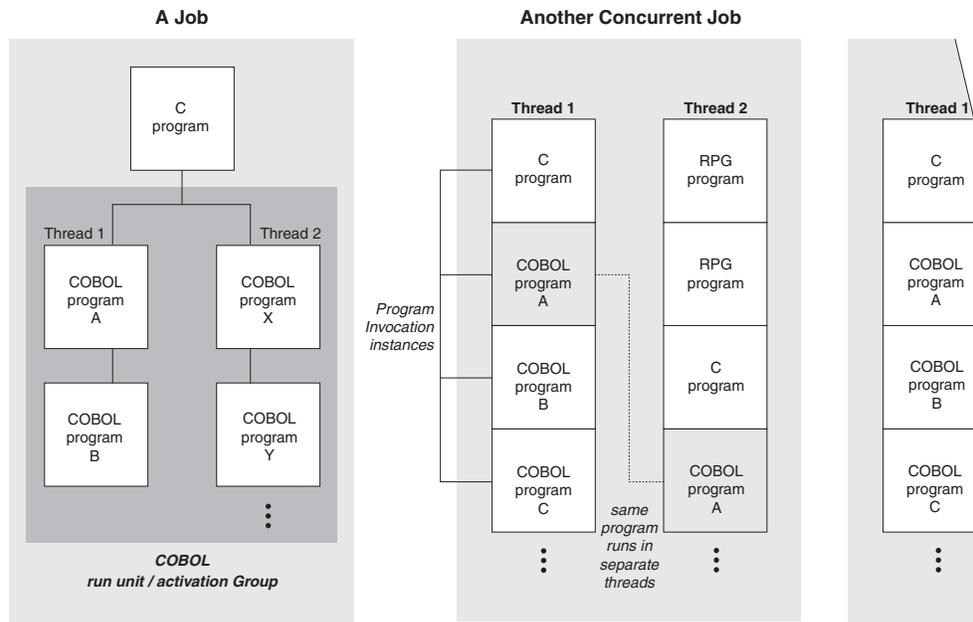


Figure 86. Schematic Illustration of Multithreading Concepts

ILE COBOL does not have a COBOL statement to support initiating or managing program threads, but COBOL programs can use APIs to do this. ILE COBOL programs can run in threads in a multithreaded environment. In other words, ILE COBOL programs can be invoked by other applications such that they are running in multiple threads within a job or as multiple program invocation instances within a thread.

The remainder of this chapter contains information that will help you prepare your ILE COBOL programs for multithreaded environments.

This chapter describes:

- How Language Elements Are Interpreted in a Multithreaded Environment
- When to Choose THREAD for Multithreading Support
- Control Transfer within a Multithreaded Environment
- An Example of Using ILE COBOL in a Multithreaded Environment.

How Language Elements Are Interpreted in a Multithreaded Environment

Because your ILE COBOL programs can be run as separate threads within a job, be aware that language elements might be interpreted in two ways:

Run-unit scope

The language element persists for the duration of the ILE COBOL run-unit execution and is available to other programs within the thread.

Program invocation instance scope

The language element persists only within a particular program invocation instance.

These two types of scope are important in two contexts:

Reference

Describes where an item can be referenced from. For example, if a data item has run-unit reference scope, any program invocation instance in the run unit can reference the data item.

State Describes how long an item persists in storage. For example, if a data item has program invocation instance state scope, it will remain in storage only while the program invocation instance is running.

The following table summarizes the reference and state scope of various ILE COBOL language elements:

Language Element	Reference Scope	State Scope
ADDRESS-OF special register	Same as associated record	Program invocation instance
DB-FORMAT-NAME special register	Run-unit	Program invocation instance
DEBUG-ITEM special register	Syntax checked only	
Files	Run-unit	Run-unit
FORMAT OF special register	Same as associated identifier	Same as associated identifier
Index data	Program	Program invocation instance
LENGTH OF special register	Same as associated identifier	Same as associated identifier
LINAGE-COUNTER special register	Same as associated file	Same as associated file
LINKAGE-SECTION data	Run-unit	Based on scope of underlying data
LOCAL-STORAGE data	Program	Program invocation instance
LOCALE OF special register	Same as associated identifier	Same as associated identifier
RETURN-CODE	Run-unit	Program invocation instance
WHEN-COMPILED special register	Run-unit	Run-unit
WORKING-STORAGE data	Run-unit	Run-unit
SORT-RETURN special register	Run-unit	Program invocation instance

Working with Run-Unit Scoped Elements

Run-unit scoped elements have storage that can be shared across modules. Examples of shared storage are:

- External and shared files
- External data items
- CALL BY REFERENCE between modules

Within a run unit, every module has a lock, and ILE COBOL ensures that only one copy of a module is running at a time in the run unit. If you have resources with run-unit scope, it is your responsibility to synchronize access to that data from multiple threads using logic in the application. You can do one or both of the following:

- Structure the application such that run-unit scoped resources are not accessed simultaneously from multiple threads.
- If you are going to access resources simultaneously from separate threads, synchronize access using facilities provided by C or by platform functions such as the Pthread mutex support or the MI built-in functions for creating and handling mutexes. For more information, refer to the Multithreaded Applications document, listed under the **Programming** topic, at the following URL:

Working with Program Invocation Instance Scoped Elements

With these language elements, storage is allocated for each individual program invocation instance. Therefore, even if a program is invoked multiple times among multiple threads, each time it is invoked it will be allocated separate storage. For example, if program X is invoked in two or more threads, each program invocation instance of X gets its own set of resources, such as storage.

Because the storage associated with these language elements is program invocation instance scoped, data is protected from access across threads and you do not have to concern yourself with access synchronization. However, this data cannot be shared between invocations of programs unless it is explicitly passed.

Choosing THREAD for Multithreading Support

```
# The THREAD(SERIALIZE) PROCESS option must be coded in all modules that
# interact with a multi-threaded Java application. COBOL relies heavily on static
# storage even in programs or procedures that apparently only use automatic
# storage. THREAD(SERIALIZE) is necessary to ensure the correct handling of this
# static storage. This applies not only to modules that contain calls to Java methods,
# but also to any modules that might be called during interactions with Java, when
# the Java part of the application might be running with multiple threads.
```

Select SERIALIZE on the THREAD option of the PROCESS statement for multithreading support. Compiling with SERIALIZE prepares the ILE COBOL run-time environment for threading support. However, compiling with SERIALIZE may reduce program performance. You must compile all of the programs in the run unit with SERIALIZE; you cannot mix programs compiled with SERIALIZE and those compiled with NOTHREAD in one run unit.

The default option is THREAD(NOTHREAD). For more information about the THREAD PROCESS statement options, see “THREAD Option” on page 59.

Language Restrictions under THREAD

When THREAD(SERIALIZE) is in effect, the following language elements are not supported and are flagged by the compiler with a severe error message (of severity 30):

- ALTER statement
- GO TO statement without a procedure name
- INITIAL phrase in PROGRAM-ID paragraph
- STOP literal statement
- STOP RUN
- WITH DEBUGGING MODE clause

Use of DDM data areas is not allowed in a multithreaded environment.

It is recommended that you do not use UPSI switches in a multithreaded environment, since it is possible for one thread to set a switch and another thread to set it again before the first thread has checked it.

Control Transfer within a Multithreaded Environment

Be aware of the following control transfer issues when writing ILE COBOL programs for a multithreaded environment:

CALL and CANCEL

As is the case in single-threaded environments, a program invoked is in its initial state the first time it is called within a run unit and the first time it is called after a CANCEL to the CALLED program.

EXIT PROGRAM

EXIT PROGRAM returns to the caller of the program without terminating the thread in all cases. EXIT PROGRAM from a main program is treated as a comment.

GOBACK

Same as EXIT PROGRAM, except that GOBACK from a main program returns to the caller. This determination can be made if all ILE COBOL programs invoked within the run unit have returned to their invokers via GOBACK or EXIT PROGRAM.

Limitations on ILE COBOL in a Multithreaded Environment

Some ILE COBOL applications depend on subsystems or other applications. In a multithreaded environment, these dependencies result in some limitations on ILE COBOL programs:

SORT/MERGE

SORT and MERGE should only be active in one thread at a time. However, this is not enforced by the COBOL run-time environment— it must be controlled by the application.

External and shared files

External and shared files should not be accessed or updated simultaneously from multiple threads. However, this is not enforced by the COBOL run-time environment— it must be controlled by the application.

In general, synchronizing access to resources visible to an application within a run unit is the responsibility of the application.

Example of Using ILE COBOL in a Multithreaded Environment

This example consists of an ILE COBOL main procedure that creates two ILE COBOL threads, waits for the ILE COBOL threads to finish, then exits.

Sample Code for the Multithreading Example

The example has three code samples:

THRCBL QCBLESRC

An ILE COBOL main procedure that creates the ILE COBOL threads, waits for them to finish, then exits.

SUBA QCBLESRC

An ILE COBOL procedure that is called by the thread created by THRCBL.

SUBB QCBLESRC

A second ILE COBOL procedure that is called by the thread created by THRCBL.

The sample code for THRCBL QCBLLSRC is shown in Figure 87.

```
PROCESS NOMONOPRC OPTIONS THREAD(SERIALIZE).
IDENTIFICATION DIVISION.
PROGRAM-ID. THRCBL.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
    special-names. system-console is oper1.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 pthread_attr_t typedef.
    05 FILLER PIC 9(8) usage binary occurs 4 times.
    05 FILLER USAGE POINTER.
01 pthread_t typedef.
    05 FILLER USAGE POINTER.
    05 FILLER PIC 9(8) usage binary.
    05 FILLER USAGE POINTER.
01 PROC-SUBA-PTR USAGE PROCEDURE-POINTER.
01 PROC-SUBB-PTR USAGE PROCEDURE-POINTER.
01 attr type pthread_attr_t.
01 rc PIC 9(8) usage binary value 0.
01 group1.
    05 thread type pthread_t occurs 10 times.
01 joinStatus0 USAGE POINTER.
01 joinStatus1 USAGE POINTER.
PROCEDURE DIVISION.
TEST1-INIT.
    SET PROC-SUBA-PTR TO ENTRY PROCEDURE "SUBA".
    SET PROC-SUBB-PTR TO ENTRY PROCEDURE "SUBB".

* Create a thread attributes object
    call procedure "pthread_attr_init" using attr
        returning rc.

* Define threads to be joinable
    call procedure "pthread_attr_setdetachstate" using attr
        by value 0 size 4
        returning rc.
```

Figure 87. Source code for THRCBL QCBLLSRC (Part 1 of 2)

```

* Start creating thread(s)
  call procedure "pthread_create" using thread(1) attr
    by value PROC-SUBA-PTR omitted
    returning rc.
  call procedure "pthread_create" using thread(2) attr
    by value PROC-SUBB-PTR omitted
    returning rc.
* Start joining thread(s)
  call procedure "pthread_join" using by value thread(1)
    by reference joinStatus0
    returning rc.
  call procedure "pthread_join" using by value thread(2)
    by reference joinStatus1
    returning rc.

* Destroy thread attributes object
  call procedure "pthread_attr_destroy" using attr
    returning rc.

```

Figure 87. Source code for THRCBL QCBLLSRC (Part 2 of 2)

The sample code for SUBA QCBLLSRC is shown in Figure 88.

```

PROCESS NOMONOPRC OPTIONS THREAD(SERIALIZE).
IDENTIFICATION DIVISION.
PROGRAM-ID. SUBA.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
  special-names. system-console is oper1.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 one-line pic x(11).
PROCEDURE DIVISION.
TEST1-INIT.
  move "IN SUBA" TO ONE-LINE.
  DISPLAY one-line UPON oper1.

```

Figure 88. Source code for SUBA QCBLLSRC

The sample code for SUBB QCBLLSRC is shown in Figure 89

```

PROCESS NOMONOPRC OPTIONS THREAD(SERIALIZE).
IDENTIFICATION DIVISION.
PROGRAM-ID. SUBB.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
  special-names. system-console is oper1.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 one-line pic x(11).
PROCEDURE DIVISION.
TEST1-INIT.
  move "IN SUBB" TO ONE-LINE.
  DISPLAY one-line UPON oper1.

```

Figure 89. Source code for SUBB QCBLLSRC

Creating and Running the Multithreading Example

To create and run the multithreading example, follow these steps:

1. Create three ILE COBOL modules
 - To create the ILE COBOL module THRCBL, type
`CRTCBLMOD MODULE(THRCBL) SRCFILE(*CURLIB/QCBLLESRC) DBGVIEW(*ALL)`
 - To create the ILE COBOL module SUBA, type
`CRTCBLMOD MODULE(SUBA) SRCFILE(*CURLIB/QCBLLESRC) DBGVIEW(*ALL)`
 - To create the ILE COBOL module SUBB, type
`CRTCBLMOD MODULE(SUBB) SRCFILE(*CURLIB/QCBLLESRC) DBGVIEW(*ALL)`
2. Create a program THREAD using the three modules
 - To create the THREAD program, type
`CRTPGM PGM(THREAD) MODULE(*CURLIB/THRCBL *CURLIB/SUBA *CURLIB/SUBB)`
3. Create a SPAWN command to call the multithreaded program THREAD
`SPAWN MYLIB/THREAD DEBUG(2)`

For information on how to create a SPAWN command, refer to the Multithreaded Applications document, listed under the **Programming** topic, at the following URL: <http://www.ibm.com/systems/i/infocenter/>

4. To display the output of the program, type `DSPMSG QSYSOPR`. The output depends on which thread runs first and will show the sequence of the threads as:

```
IN SUBB  
IN SUBA
```

or as,

```
IN SUBA  
IN SUBB
```

Chapter 16. ILE COBOL Error and Exception Handling

ILE COBOL contains special elements to help you anticipate and correct error conditions that can occur when your program is running. Even if your code is flawless, errors may occur in the system facilities that your program uses.

You can anticipate possible error conditions by putting code into your program to handle them. If error-handling code is not present in your program, your program could behave in a manner that you did not anticipate, data files could be corrupted, and incorrect output may be produced. Without error-handling code, you may not even be aware that a problem exists.

The action taken by your error-handling code can vary from attempting to cope with the situation and continue, to issuing a message, to halting the program. At a minimum, coding an error message to identify an error condition is a good idea.

When you run an ILE COBOL program, several types of errors can occur. The ILE COBOL statement active at the time of a given error causes certain ILE COBOL clauses or phrases to be run.

This chapter discusses how to:

- Use error-handling bindable APIs
- Initiate deliberate dumps
- Handle errors in string operations
- Handle errors in arithmetic operations
- Handle errors in input-output operations
- Handle errors in sort/merge operations
- Handle exceptions on the CALL statement
- Create user-written error-handling routines.

ILE Condition Handling

On the AS/400 system, there are several ways that programs can communicate status to one another. One of the main methods is to send an IBM i message.

There are several type of IBM i messages. These include inquiry, informational, completion, escape, and notify. For example, the final message sent by the ILE COBOL compiler when a compilation is successful is LNC0901,

Program program-name created in library library-name on date at time.

Message LNC0901 is a completion message. If a compilation fails, you will receive message LNC9001,

Compile failed. Program-name not created.

Message LNC9001 is an escape message.

An ILE condition and an IBM i message are quite similar. Any escape, status, notify, or function check message is a condition, and every ILE condition has an associated IBM i message.

Like IBM i messages, which can be handled by declaring and enabling a message monitor, an ILE condition can be handled by registering an **ILE condition handler**.

An ILE condition handler allows you to register an exception handling procedure at run time that is given control when an exception occurs. To register an exception handler, use the Register a User-Written Condition Handler (CEEHDLR) bindable API.

When a program object or an ILE procedure is called, a new call stack entry is created. Associated with each call stack entry is a call message queue. This call message queue is a program message queue if a program object is called, or a procedure message queue if an ILE procedure is called. In ILE, you can send a message to a program object or ILE procedure by sending a message to its call stack entry.

Similarly, you can signal a condition to a program object or ILE procedure by signalling a condition to its call stack entry. You can signal a condition to a program object by using ILE bindable APIs. Refer to the section on ILE bindable APIs in *ILE Concepts* for a list of Condition Management bindable APIs.

Each call stack entry can have several ILE condition handlers registered. When multiple ILE condition handlers are registered for the same call stack entry, the system calls these handlers in last-in-first-out (LIFO) order. These ILE condition handlers can also be registered at different priority levels. Only a few of these priorities are available to ILE COBOL. There are approximately ten distinct priorities ranging from 85 to 225. ILE condition handlers are called in increasing priority order.

In ILE, if an exception condition is not handled at a particular call stack entry, the unhandled exception message is percolated to the previous call stack entry message queue. When this happens, exception processing continues at the previous call stack entry. Percolation of an unhandled exception condition continues until either a control boundary is reached or the exception message is handled. An unhandled exception message is converted to a function check when it is percolated to the control boundary.

The function check exception message can then be handled by the call stack entry that issued the original exception condition or it is percolated to the control boundary. If the function check is handled, normal processing continues and exception processing ends. If the function check is percolated to the control boundary, ILE considers the application to have ended with an unexpected error. The generic failure exception message, CEE9901, is sent by ILE to the caller of the control boundary.

When an exception condition occurs in a program object or an ILE procedure, it is first handled by the registered ILE condition handler for the call stack entry of the program object or ILE procedure. If there is no registered ILE condition handler for the call stack entry, then the exception condition is handled by HLL-specific error handlers. HLL-specific error handlers are language features defined for handling errors. HLL-specific error handling in ILE COBOL includes the USE declarative for I/O error handling and imperatives in statement-scoped condition phrases such as ON SIZE ERROR and INVALID KEY.

If the exception condition is not handled by the HLL-specific error handling, then the unhandled exception condition is percolated to the previous call stack entry message queue, as described above.

For more information on ILE condition handling, refer to the sections on error handling, and exception and condition management in the *ILE Concepts* book.

Ending an ILE COBOL Program

An ILE COBOL program can be ended by the following:

- A ILE COBOL statement (EXIT PROGRAM, STOP RUN, or GOBACK)
- A reply to an inquiry message
- An implicit STOP RUN or EXIT PROGRAM statement
- Another ILE language's equivalent of the ILE COBOL STOP RUN statement. For example, ILE C's `exit()` function.
- Another ILE language's equivalent of the ILE COBOL abnormal STOP RUN statement. For example, ILE C's `abort()` function.
- An escape message that is sent past the calling ILE COBOL program by the called ILE procedure or program object.
- Ending, by the called ILE procedure or program object, of the activation group in which the calling ILE COBOL program is running.

A STOP RUN statement is implied when a main ILE COBOL program has no next executable statement (implicit EXIT PROGRAM for a ILE COBOL subprogram), that is, when processing falls through the last statement of a program.

Inquiry messages can be issued in response to a ILE COBOL statement (namely a STOP literal), but they are usually issued when a severe error occurs in a program, or when a ILE COBOL operation does not complete successfully. (Examples are LNR7205, LNR7207, and LNR7208.) Inquiry messages allow you to determine what action to take after an exception error has occurred.

There are four common replies to a COBOL inquiry message: C, D, F, and G (cancel, cancel and dump, cancel and full dump, continue). The first three cause (as their final steps) an implicit abnormal STOP RUN.

An implicit or explicit STOP RUN statement, or a GOBACK statement in the main ILE COBOL program, causes the termination-imminent condition to be signalled to the nearest control boundary. The termination-imminent condition can be handled in two ways:

- Through a registered error handler before it reached the control boundary, or

Note: To register an exception handler, use the Register a User-Written Condition Handler (CEEHDLR) bindable API. Refer to *ILE Concepts* for more information on exception handlers.

- If it reached the control boundary, then all programs after the control boundary are ended, and control returns to the program before the control boundary.

If this control boundary is a hard control boundary, then the activation group (run unit) will end.

If the STOP RUN is abnormal and a hard control boundary is reached, the CEE9901 escape message is issued to the program before the control boundary.

Using Error Handling Bindable Application Programming Interfaces (APIs)

There are two level at which errors can be handled in ILE COBOL. First, the condition handlers registered at each priority level have a chance to handle the condition. If the condition remains unhandled when the control boundary is reached, a function check condition is sent. Each ILE COBOL ILE procedure has an

ILE condition handler registered at priority level 205 to handle a function check. This function check condition handler will issue a COBOL inquiry message, unless handled by the following bindable APIs:

- Retrieve COBOL Error Handler (QlnRtvCobolErrorHandler)
The Retrieve COBOL Error Handler (QlnRtvCobolErrorHandler) API allows you to retrieve the name of the current ILE COBOL error-handling procedure for the activation group from which the API is called.
- Set COBOL Error Handler (QlnSetCobolErrorHandler)
The Set COBOL Error Handler (QlnSetCobolErrorHandler) API allows you to specify the identity of an ILE COBOL error-handling procedure for the activation group from which the API is called.

These APIs only affect exception handling within ILE COBOL programs. For
detailed information on all of these APIs, refer to the section about COBOL APIs in
the *CL and APIs* section of the *Programming* category in the **i5/OS Information**
Center at this Web site -<http://www.ibm.com/systems/i/infocenter/>.

Note: The *NOMONOPRC value must be specified on the OPTION parameter of
the CRTCBMOD or CRTBNDCBL commands in order to use these APIs.

Initiating Deliberate Dumps

You can use the Dump COBOL (QlnDumpCobol) bindable API to deliberately cause a formatted dump of an ILE COBOL program. The QlnDumpCobol API accepts six parameters which define the:

- Program object name
- Library name
- Module object name
- Program object type
- Dump type
- Error code.

The following are some examples of how to call the QlnDumpCobol API and the resultant operations:

```
WORKING-STORAGE SECTION.
01  ERROR-CODE.
    05  BYTES-PROVIDED      PIC S9(6)  BINARY VALUE ZERO.
    05  BYTES-AVAILABLE    PIC S9(6)  BINARY VALUE ZERO.
    05  EXCEPTION-ID       PIC X(7).
    05  RESERVED-X        PIC X.
    05  EXCEPTION-DATA     PIC X(64).
01  PROGRAM-NAME          PIC X(10).
01  LIBRARY-NAME          PIC X(10).
01  MODULE-NAME           PIC X(10).
01  PROGRAM-TYPE          PIC X(10).
01  DUMP-TYPE             PIC X.
PROCEDURE DIVISION.
MOVE LENGTH OF ERROR-CODE TO BYTES-PROVIDED.
MOVE "MYPROGRAM"         TO PROGRAM-NAME.
MOVE "TESTLIB"           TO LIBRARY-NAME.
MOVE "MYMOD1"            TO MODULE-NAME.
MOVE "*PGM"              TO PROGRAM-TYPE.
MOVE "D"                 TO DUMP-TYPE.
CALL PROCEDURE "QlnDumpCobol" USING PROGRAM-NAME,
                                LIBRARY-NAME, MODULE-NAME,
                                PROGRAM-TYPE, DUMP-TYPE,
                                ERROR-CODE.
```

This would provide a formatted dump of COBOL identifiers (option D) for the module object MYMOD1 in program object MYPROGRAM in library TESTLIB.

```
WORKING-STORAGE SECTION.  
01 ERROR-CODE.  
    05 BYTES-PROVIDED    PIC S9(6)  BINARY VALUE ZERO.  
    05 BYTES-AVAILABLE  PIC S9(6)  BINARY VALUE ZERO.  
    05 EXCEPTION-ID     PIC X(7).  
    05 RESERVED-X      PIC X.  
    05 EXCEPTION-DATA   PIC X(64).  
01 PROGRAM-NAME        PIC X(10).  
01 LIBRARY-NAME        PIC X(10).  
01 MODULE-NAME         PIC X(10).  
01 PROGRAM-TYPE        PIC X(10).  
01 DUMP-TYPE           PIC X.  
PROCEDURE DIVISION.  
    MOVE LENGTH OF ERROR-CODE TO BYTES-PROVIDED.  
    MOVE "*SRVPGM"           TO PROGRAM-TYPE.  
    MOVE "F"                 TO DUMP-TYPE.  
    CALL PROCEDURE "QlnDumpCobol" USING OMITTED, OMITTED,  
                                       OMITTED, PROGRAM-TYPE,  
                                       DUMP-TYPE, ERROR-CODE.
```

This would provide a formatted dump of COBOL identifiers and file related information (option F) for the service program that called the QlnDumpCobol API.

If any of the input parameters to the QlnDumpCobol API contain data that is not valid, the dump is not performed and an error message is generated or exception data is returned. An error message is generated if the BYTES-PROVIDED field contains zero. If the BYTES-PROVIDED field contains a value other than zero, then exception data is returned in the ERROR-CODE parameter and no error message is generated.

If you do not want a user to be able to see the values of your program's variables in a formatted dump, do one of the following:

- Ensure that debug data is not present in the program by removing observability.
- Give the user sufficient authority to run the program, but not to perform the formatted dump. This can be done by giving *OBJOPR plus *EXECUTE authority.
- Do not call QlnDumpCobol in the program.

```
# For detailed information on the QlnDumpCobol API, refer to the section about  
# COBOL APIs in the CL and APIs section of the Programming category in the i5/OS  
# Information Center at this Web site -http://www.ibm.com/systems/i/infocenter/.
```

Program Status Structure

The program status structure is a predefined structure containing subfields that provide you with error information when an error occurs in your program. You access these subfields by using the PROGRAM STATUS clause to specify the data item(s) that will receive the error information. Refer to the *WebSphere Development Studio: ILE COBOL Reference* for details on the program status structure and the PROGRAM STATUS clause.

Handling Errors in String Operations

When stringing or unstringing data, an error might occur. Both the STRING and UNSTRING statements provide an ON OVERFLOW phrase to handle typical string overflow error conditions. For the STRING statement, the ON OVERFLOW phrase will be run when the implicit or explicit pointer value is:

- Less than 1
- Greater than the length of the receiving field.

For the UNSTRING statement, the ON OVERFLOW phrase will be run when:

- The implicit or explicit pointer value is less than 1
- The implicit or explicit pointer value is greater than the length of the sending field
- All receiving fields have been acted upon, and the sending field still contains unexamined characters.

Any other error conditions not handled by the ON OVERFLOW phrase will generally result in MCH messages. Such messages will typically be handled by the function check condition handler. To prevent the function check condition handler from being called, you can register your own condition handler, using the CEEHDLR API, to catch the MCH messages.

You use the ON OVERFLOW phrase of the STRING or UNSTRING statement to identify the error-handling steps that you want to perform when an overflow condition occurs. If you do not have an ON OVERFLOW clause on the STRING or UNSTRING statement, control passes to the next sequential statement, and you are not notified of the incomplete operation.

Refer to the STRING and UNSTRING statements in the *IBM Rational Development Studio for i: ILE COBOL Reference* for further information about the ON OVERFLOW phrase.

Handling Errors in Arithmetic Operations

Arithmetic operations can cause certain typical errors to occur. These typical errors generally result in MCH messages.

The ON SIZE ERROR Phrase

The ON SIZE ERROR phrase of the ADD, SUBTRACT, MULTIPLY, DIVIDE, and COMPUTE statement will:

- Enable binary and decimal overflow messages to be issued. The binary and decimal overflow message is MCH1210. The decimal division by zero message is MCH1211.
- Register a condition handler to catch the binary, decimal, and floating-point overflow messages, as well as other arithmetic MCH messages. Floating-point overflow messages include MCH1206 (overflow) and MCH1207 (underflow).

Unlike binary and decimal overflow messages, floating-point overflow is not enabled by the existence of an ON SIZE ERROR phrase. Floating-point overflow is enabled or disabled at the job level. By default, floating-point overflow messages are always issued. Thus, ILE COBOL will ignore these messages, except when an ON SIZE ERROR phrase is coded. To enable or disable floating-point overflow, see the section “Handling Errors in Floating-Point Computations” on page 375.

ILE COBOL registers the above mentioned condition handler at priority level 85. A user condition handler, which is registered at priority level 165, will only receive control if the above mentioned condition handler does not handle the exception.

When no ON SIZE ERROR phrase is coded, the binary and decimal overflow messages will not be issued, and floating-point overflow messages will be ignored. All other arithmetic MCH messages will typically be handled by the function check condition handler unless a user condition handler has been registered using the CEEHDLR API.

A size error condition occurs in the following situations:

- The result of the arithmetic operation is larger than the fixed-point field that is to hold it
- Division by zero
- Zero raised to the zero power
- Zero raised to a negative number
- A negative number raised to a fractional power
- Floating-point overflow or underflow.

During arithmetic operations, typical errors are size errors (MCH1210) and decimal data errors (MCH1202). Most MCH errors are not directly detected by ILE COBOL; they are detected by the operating system and result in system messages. ILE COBOL then monitors for these messages, setting internal bits that determine whether to run a SIZE ERROR imperative statement or issue a runtime message (LNR7200) to end the program.

To prevent the LNR7200 message from being sent, a user condition handler can be registered using the CEEHDLR API to handle the MCH messages or an ILE COBOL error handler can be coded using the COBOL bindable APIs to handle the LNR72xx inquiry messages.

ILE COBOL does detect errors that result from division by zero during an arithmetic operation. If detected by ILE COBOL, these errors cause the SIZE ERROR imperative statement to run.

System message MCH1210 generally occurs when moving one binary or decimal numeric field to another, and the receiver is too small. This error is monitored by ILE COBOL, and also results in the running of the SIZE ERROR imperative statement.

LNR7200 is a run-time message that is usually issued when an unmonitored severe error occurs in your ILE COBOL program.

System message MCH1202 is a typical example of an unmonitored severe error. This kind of error results in the ILE COBOL run-time message LNR7200 (or LNR7204 if the error occurs in a program called by a ILE COBOL program). System messages MCH3601 and MCH0601 are other examples of unmonitored severe errors.

Handling Errors in Floating-Point Computations

IBM i provides a group of Computation Attributes (CA) MI instructions to retrieve information about floating-point operations and to change the way floating-point operations behave. For example, the SETCA (Set Computational Attributes) MI instruction can prevent certain floating-point exceptions from occurring, as well as

indicating whether or not rounding is done. By default, the result of a floating-point operation is *always* rounded, and all of the exceptions, except for Invalid Operand are signalled.

The exceptions that can be prevented are floating-point:

1. Overflow
2. Underflow
3. Zero divide
4. Inexact result
5. Invalid operand

For ON SIZE ERROR phrase handling, ILE COBOL requires that the first 3 exceptions must be signaled.

ILE COBOL also requires rounding to the nearest decimal position to take place, which means if you used the CA MI instructions to prevent rounding, the extra digits would be dropped, leaving you with an inexact result.

Handling Errors in Input-Output Operations

Error handling helps you during the processing of input-output statements by catching severe errors that might not otherwise be noticed. For input-output operations, there are several important error-handling phrases and clauses. These are as follows:

- AT END phrase
- INVALID KEY phrase
- NO DATA phrase
- USE AFTER EXCEPTION/ERROR declarative procedure
- FILE STATUS clause.

During input-output operations, errors are detected by the system, which sends messages; the messages are then monitored by ILE COBOL. As well, ILE COBOL will detect some errors during an input-output operation without system support. Regardless of how an error is detected during an input-output operation, the result will always be an internal file status of other than zero, a runtime message, or both.

An important characteristic of error handling is the issuing of a runtime message when an error occurs during the processing of an input-output statement if there is no AT END/INVALID KEY phrase in the input-output statement, USE AFTER EXCEPTION/ERROR procedure, or FILE STATUS clause in the SELECT statement for the file.

One thing to remember about input-output errors is that you choose whether or not your program will continue running after a less-than-severe input-output error occurs. ILE COBOL does not perform corrective action. If you choose to have your program continue (by incorporating error-handling code into your design), you must also code the appropriate error-recovery procedure.

Besides the error-handling phrases and clauses that specifically relate to input-output statements, user defined ILE condition handlers and ILE COBOL error handling APIs can also be used to handle I/O errors.

For each I/O statement, ILE COBOL registers a condition handler to catch the various I/O related conditions. These condition handlers are registered at priority level 185 which allows user defined condition handlers to receive control first.

As mentioned above, an ILE COBOL runtime message is issued when an error occurs and no appropriate AT END, INVALID KEY, USE procedure, or FILE STATUS clause exists for a file. The message, LNR7057, is an escape message. This message can be handled by a user-defined condition handler. If no condition handler can handle this message, message LNR7057 will be resent as a function check.

ILE COBOL has a function check condition handler that will eventually issue inquiry message LNR7207 unless an ILE COBOL error handling API has been defined.

Processing of Input-Output Verbs

The following diagram shows when the USE procedure and the (NOT) AT END, (NOT) INVALID KEY, and NO DATA imperative statements are run.

The file status shown here refers to the internal file status.

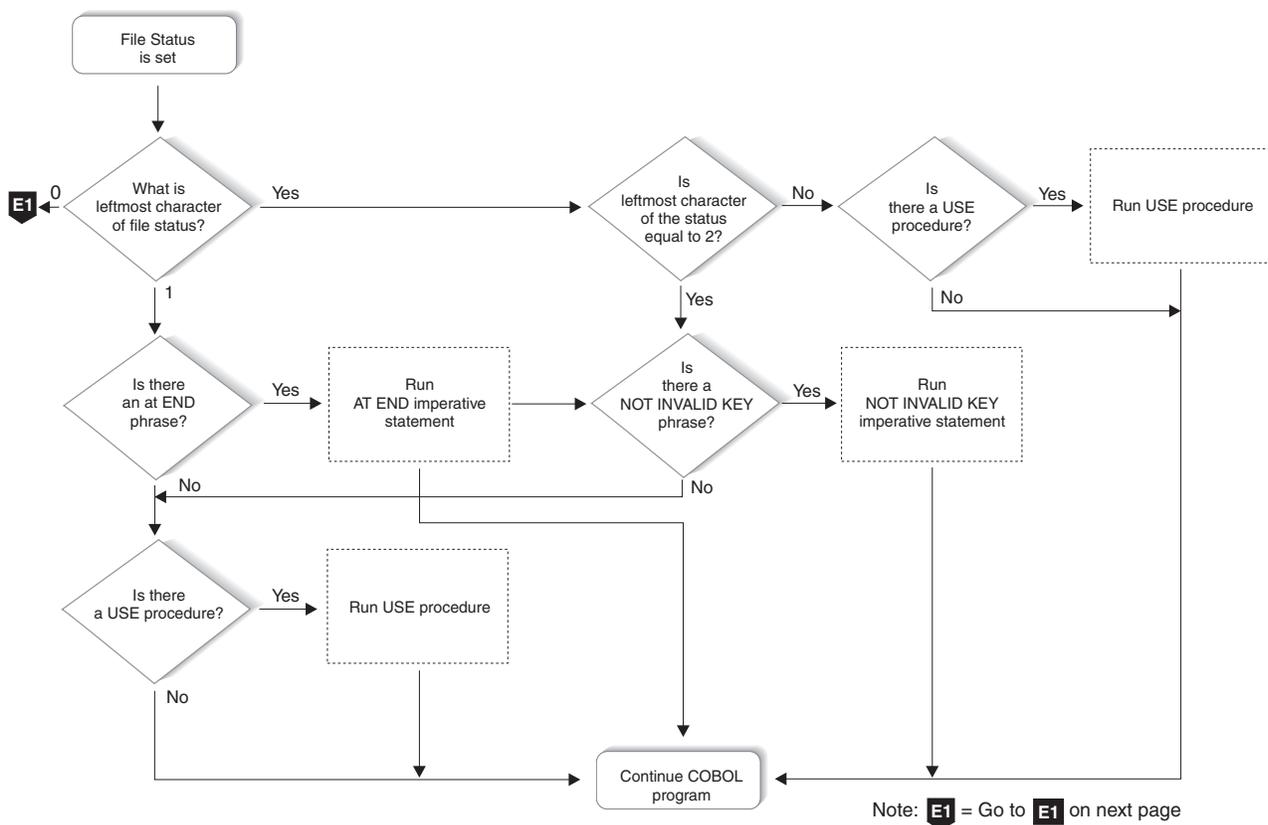


Figure 90. Processing of I/O Verbs (Part 1 of 2)

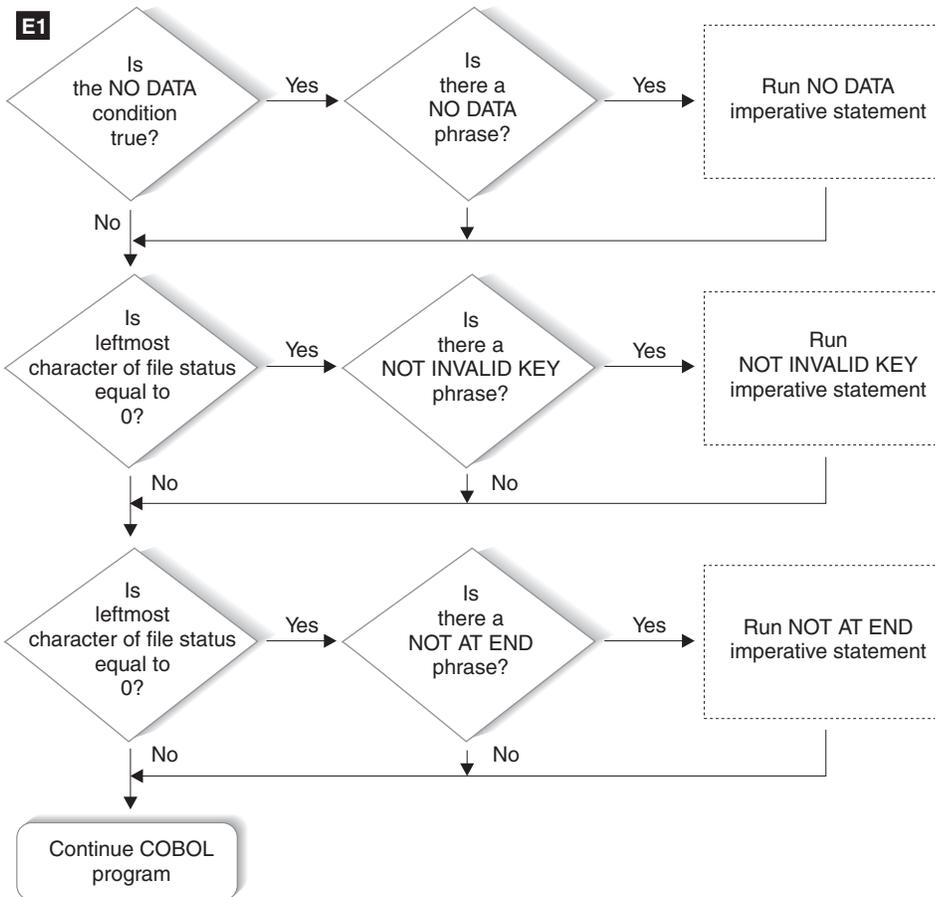


Figure 90. Processing of I/O Verbs (Part 2 of 2)

Note: Follow the parts of the diagram that apply to your statements.

Detecting End-of-File Conditions (AT END Phrase)

An end-of-file condition may or may not represent an error. In many designs, reading sequentially to the end of a file is done intentionally, and the AT END condition is expected.

In some cases, however, the end-of-file condition will reflect an error. You code the AT END phrase of the READ statement to handle either case, according to your program design.

If you code an AT END phrase, the imperative statement identified by the phrase is performed when an end-of-file condition occurs. If you do not code an AT END phrase, the associated USE AFTER EXCEPTION/ERROR declarative is performed.

Any NOT AT END phrase that you code is performed only if the READ statement completed successfully. If the READ operation fails because of any condition other than end-of-file, neither the AT END nor the NOT AT END phrase is performed. Instead, control passes to the end of the READ statement after performing the associated USE AFTER EXCEPTION/ERROR declarative procedure.

If you have coded neither an AT END phrase nor a USE AFTER EXCEPTION/ERROR declarative procedure, but you have coded a STATUS KEY clause for the file, control passes to the next sequential instruction after the

input-output statement that detected the end-of-file condition. At this point, your code should look at the status key and take some appropriate action to handle the error.

Detecting Invalid Key Conditions (INVALID KEY Phrase)

The imperative statement identified by the INVALID KEY phrase will be given control in the event that an input-output error occurs because of a faulty index key or relative key. You can include INVALID KEY phrases on READ, START, WRITE, REWRITE, and DELETE statements for indexed and relative files.

The INVALID KEY phrases differ from USE AFTER EXCEPTION/ERROR declaratives in these ways:

- INVALID KEY phrases operate for only limited types of errors, whereas the USE AFTER EXCEPTION/ERROR declarative encompasses most forms of errors.
- INVALID KEY phrases are coded directly onto the input-output verb, whereas the USE AFTER EXCEPTION/ERROR declaratives are coded separately.
- INVALID KEY phrases are specific to one single input-output operation, whereas the USE AFTER EXCEPTION/ERROR declaratives are more general.

If you specify the INVALID KEY phrase in an input-output statement that causes an invalid key condition, control is transferred to the imperative statement identified by the INVALID KEY phrase. In this case, any USE AFTER EXCEPTION/ERROR declaratives you have coded are not performed.

Any NOT INVALID KEY phrase that you specify is performed only if the statement completes successfully. If the operation fails because of any condition other than invalid key, neither the INVALID KEY nor NOT INVALID KEY phrase is performed. Instead, control passes to the end of the input-output statement after performing any associated USE AFTER EXCEPTION/ERROR declaratives.

Use the FILE STATUS clause in conjunction with the INVALID KEY phrase to evaluate the status key and determine the specific invalid key condition.

For example, assume you have a file containing master customer records and you need to update some of these records with information in a transaction update file. You will read each transaction record, find the corresponding record in the master file, and make the necessary updates. The records in both files each contain a field for a customer number, and each record in the master file has a unique customer number.

The FILE-CONTROL entry for the master file of commuter records includes statements defining indexed organization, random access, MASTER-COMMUTER-NUMBER as the prime record key, and COMMUTER-FILE-STATUS as the file status key. The following example illustrates how you can use the FILE STATUS clause in conjunction with the INVALID KEY phrase to more specifically determine the cause of an input-output statement failure.

```
.  
  . (read the update transaction record)  
  .  
  MOVE "TRUE" TO TRANSACTION-MATCH  
  MOVE UPDATE-COMMUTER-NUMBER TO MASTER-COMMUTER-NUMBER  
  READ MASTER-COMMUTER-FILE INTO WS-CUSTOMER-RECORD  
  INVALID KEY
```

```
DISPLAY "MASTER CUSTOMER RECORD NOT FOUND"  
DISPLAY "FILE STATUS CODE IS: " COMMUTER-FILE-STATUS  
MOVE "FALSE" TO TRANSACTION-MATCH  
END-READ
```

Using EXCEPTION/ERROR Declarative Procedures (USE Statement)

You can code one or more USE AFTER EXCEPTION/ERROR declarative procedures in your ILE COBOL program that will be given control if an input-output error occurs. You can have:

- Individual procedures for each file open mode (whether INPUT, OUTPUT, I-O, or EXTEND)
- Individual procedures for each particular file.
- Individual procedures for groups of files.

Place each such procedure in the declaratives section of the Procedure Division of your program. Refer to the *IBM Rational Development Studio for i: ILE COBOL Reference* for details about how to write a declarative.

In your procedure, you can choose to attempt corrective action, retry the operation, continue, or end the program. You can use the USE AFTER EXCEPTION/ERROR declarative procedure in combination with the status key if you want further analysis of the error.

For GLOBAL files, each ILE COBOL program can have its own USE AFTER EXCEPTION/ERROR declarative procedure.

USE AFTER EXCEPTION/ERROR declarative can themselves be declared GLOBAL. Special precedence rules are followed when multiple declaratives may be performed on an I/O error. In applying these rules, only the first qualifying declarative will be selected for execution. The declarative that is selected must satisfy the rules for execution of that declarative. The order of precedence for selecting a declarative is:

1. A file-specific declarative (one of the form USE AFTER ERROR ON *file-name-1*) within the program that contains the statement that caused the qualifying condition
2. A mode-specific declarative (one of the form USE AFTER ERROR ON INPUT) within the program that contains the statement that caused the qualifying condition
3. A file-specific declarative that specifies the GLOBAL phrase, and is within the program directly containing the program that was last examined for a qualifying condition
4. A mode-specific declarative that specifies the GLOBAL phrase, and is within the program directly containing the program that was last examined for a qualifying condition.
5. Rules 3 and 4 apply recursively back through the parents in the nest of programs.

Write a USE AFTER EXCEPTION/ERROR declarative procedure if you want to return control to your program after an error occurs. If you don't write such a procedure, your job may be cancelled or abnormally ended after an error occurs.

Each USE AFTER EXCEPTION/ERROR declarative procedure runs as a separate invocation from that of other declarative procedures and the non-declarative part of the same ILE COBOL program. Thus, if you call the CEEHDLR API to register

an ILE condition handler from a declarative procedure, that ILE condition handler is invoked only for exceptions that occur in the USE AFTER EXCEPTION/ERROR declarative procedure and not for exceptions that occur in any other part of the ILE COBOL program.

Determining the Type of Error Through the File Status Key

The file status key is updated after each input-output operation on a file by placing values in the two digits of the file status key. In general, a zero in the first digit indicates a successful operation, and a zero in both digits means "nothing abnormal to report".

You must provide a FILE-CONTROL entry to specify the organization and access method for each file used by your ILE COBOL program. You can also code a FILE STATUS clause in this entry.

The FILE STATUS clause designates one or two data items (coded in the WORKING-STORAGE section) to hold a copy of the result of an I/O operation. Your copy of the first of these items is called the external file status. If you use a TRANSACTION file, you have a further record of the result called the external return code, which consists of the external major and minor return codes.

ILE COBOL keeps its information corresponding to these two data items in the ILE COBOL File Field Descriptor (FFD). ILE COBOL's copies of these two data items are called the internal file status and internal return code. In this chapter, *file status* and (*major/minor*) *return code* refer to ILE COBOL's copies unless otherwise specified.

During the processing of an I/O statement, the file status can be updated in one of three ways, as described below. The contents of the file status determine which error handling procedures to run.

Error handling procedures take control after an unsuccessful input or output operation, which is denoted by a file status of other than zero. Before any of these procedures run, the file status is copied into the external file status.

The file status is set in one of three ways:

- Method A (all files):

ILE COBOL checks the contents of variables in file control blocks. If the contents are not what is expected, a file status of other than zero is set. Most file statuses set in this way result from checking the ILE COBOL File Field Descriptor (FFD) and the system User File Control Block (UFCB).

- Method B (transaction files):

ILE COBOL checks the major and minor return codes from the system. If the major return code is not zero, the return code (consisting of major and minor return codes) is translated into a file status. If the major return code is zero, the file status may have been set by Method A or C.

For subfile READ, WRITE, and REWRITE operations, only Methods A and C apply.

For a list of return codes and their corresponding file statuses, see "File Structure Support Summary and Status Key Values" in the *IBM Rational Development Studio for i: ILE COBOL Reference*.

- Method C (all files):

A message is sent by the system when ILE COBOL calls data management to perform an I/O operation. ILE COBOL then monitors for these messages and sets a file status accordingly. Each ILE COBOL I/O operation is handled by a routine within a service program, which is supplied with the ILE COBOL compiler. This routine then calls data management to perform the I/O operation. In most cases, a single message monitor is enabled around these call to the routine in the service program.

The message monitor for each I/O operation handles typical I/O exceptions resulting in CPF messages that begin with The message monitor sets the file status based on the CPF message that it receives. For a list of messages that the message monitor handles, see “File Structure Support Summary and Status Key Values” in the *IBM Rational Development Studio for i: ILE COBOL Reference*.

Through the use of message monitors in this fashion, file status is set consistently for each type of I/O operation regardless of what other types of I/O operations you have in your program. Refer to “Handling Messages through Condition Handlers” on page 384 for more information on message monitors.

MAP 0010: How File Status is Set

001

- Start the I/O operation.
- Reset the internal file status.
- Method A: Check the contents of the variables in the file control blocks. (Check, for example, that the file has been opened properly.)

Are the variables in the file control blocks set as expected?

Yes No

002

- Set internal file status to indicate that an error has occurred.
- Continue at Step 006

003

- Call on data management to perform the I/O operation.

Does data management return an exception?

Yes No

004

- Method A: Check the contents of the variables in the file control blocks.

Are the variables in the file control blocks set as expected?

Yes No

005

- Set internal file status to indicate that an error has occurred.
- Continue at Step 006

006

- Move internal file status to external file status (specified in file status clause).
 - Based on internal file status, run the error handling code.
-

007

Is the file a transaction file?

Yes No

008

- Method C: Set the internal file status according to the CPF message sent by data management.
- Continue at Step 004

009

Are major and minor return codes available from the system?

Yes No

010

- Method C: Set the internal file status according to the CPF message sent by data management.
- Continue at Step 004 on page 383

011

- Method B: Set the internal file status based on the major and minor return codes available from the system.
- Continue at Step 004 on page 383

Interpreting Major and Minor Return Codes

When you specify a TRANSACTION file in your program, the FILE STATUS clause of your SELECT statement can contain two data names: the external file status, and the external (major and minor) return code. As described under “Determining the Type of Error Through the File Status Key” on page 381, a file status can be set in one of three ways; however, return codes are set by the system after any transaction I/O that calls data management. Consequently, most error conditions that result in a system message also have an associated return code.

Return codes are similar to file status values. That is, CPF messages sent by the system are grouped together by the ILE COBOL run time exception handler and each group of CPF messages is used to set one or more file statuses. Similarly, each major return code is also generated by a group of CPF messages. (The minor return code is not necessarily the same.) The main difference between file statuses and return codes is that the grouping of CPF messages is different.

Although ILE COBOL only sets return codes for TRANSACTION files, other types of files (such as printer files) also set return codes. You can access the return codes for these files through an ACCEPT from I-O-FEEDBACK operation.

Handling Messages through Condition Handlers

A condition handler provides a way for an ILE procedure or program object to handle messages sent by the system or by another ILE procedure or program object. A condition handler can handle one or more messages.

In some respects, a condition handler resembles a USE procedure. Similar to the way in which a USE procedure specifies actions to take in response to an I/O error, a condition handler specifies an action to take when an error occurs during the processing of a machine interface (MI) instruction. An MI instruction error is signalled by a system message, and each ILE COBOL statement is composed of one or more MI instructions.

There are two types of condition handlers:

- One type of condition handler is active for the entire program. These condition handlers are designed to handle generic error conditions.
- The other type of condition handler is active on a statement by statement basis. A typical use of these condition handlers would be to monitor I/O operations.

These condition handlers set file statuses and indicate SIZE ERROR, END-OF-PAGE, and OVERFLOW conditions.

Handling Errors in Sort/Merge Operations

You use the SORT-RETURN special register to detect errors in SORT or MERGE operations. The SORT-RETURN special register contains a return code indicating the success or failure of a SORT or MERGE operation. The SORT-RETURN special register contains a return code of 0 if the operation was successful or 16 if the operation was unsuccessful.

You can set the SORT-RETURN special register to 16 in an error declarative or input/output procedure to end a SORT/MERGE operation before all of the records have been processed. The operation ends before a record is returned or released.

The SORT-RETURN special register has the implicit definition:

```
01    SORT-RETURN  GLOBAL  PIC S9(4)  USAGE BINARY VALUE ZERO.
```

When used in a nested program, the SORT-RETURN special register is implicitly defined as GLOBAL in the outermost ILE COBOL program.

Refer to the SORT and MERGE statements in the *IBM Rational Development Studio for i: ILE COBOL Reference* for further information about the SORT-RETURN special register.

Handling Exceptions on the CALL Statement

An exception condition occurs on a CALL statement when the CALL operation itself fails. For example, the system may be out of storage or it may be unable to locate the called program. In this case, if you do not have an ON EXCEPTION or ON OVERFLOW clause on the CALL statement, your application may abnormally end. You use the ON EXCEPTION or ON OVERFLOW clause to detect the exception condition, prevent the abnormal end, and perform your own error-handling routine. For example:

```
CALL "REPORTA"  
   IN LIBRARY "MYLIB"  
   ON EXCEPTION  
     DISPLAY "Program REPORTA not available."  
END-CALL
```

If program REPORTA is unavailable or cannot be found in library MYLIB, control will continue with the ON EXCEPTION clause.

The ON EXCEPTION and ON OVERFLOW phrases handle only the exceptions that result from the failure of the CALL operation itself.

The ON EXCEPTION conditions that are signalled by the CALL operation are handled by a condition handler registered at priority 130. At this priority level, only conditions signalled to the specific call stack entry where the CALL statement exists will be handled. At this priority level, user written condition handlers may not get a chance to see signalled conditions.

If you do not have ON EXCEPTION and ON OVERFLOW phrases on the CALL statements in your application and the CALL statement fails, then the exception is handled by ILE condition handling. See "ILE Condition Handling" on page 369 for an overview of ILE condition handling.

User-Written Error Handling Routines

You can handle most error conditions that might occur when a program is running by using the ON EXCEPTION phrase, the ON SIZE ERROR phrase, and other ILE COBOL language semantics. But in the event of an extraordinary error condition like a machine check, ILE COBOL will issue an inquiry message to allow you to determine what action should be taken after a severe error has occurred.

However, ILE COBOL, in conjunction with ILE provides a mechanism, through user-written ILE condition handlers, whereby extraordinary error conditions can be handled prior to issuing an inquiry message. ILE condition handling gives you the opportunity to write your own error-handling routines to handle error conditions which can allow your program to continue running.

User-written condition handlers have priority level 165. This priority level enables user written condition handlers a chance to see signalled conditions before input-output condition handlers or ILE debugger condition handlers.

In order to have ILE pass control to your own user-written error-handling routine, you must first identify and register its entry point to ILE. To register an exception handler, you pass a procedure-pointer to the Register in a User-Written Condition Handler (CEEHDLR) bindable API. If you want to use an ILE COBOL program as an exception handler, only the outermost ILE COBOL program can be registered. Since ILE COBOL does not allow recursion for non recursive programs, if you register an ILE COBOL program as an exception handler, you must ensure that it can only be called once in an activation group, or that it is a recursive program.

Refer to *ILE Concepts* for more information on exception handlers. Procedure-pointer data items allow you to pass the entry address of procedure entry points to ILE services. For more information on procedure-pointer data items, see "Passing Entry Point Addresses with Procedure-Pointers" on page 358. Any number of user-written condition handlers can be registered. If more than one user-written condition handler is registered, the handlers are given control in last-in-first-out (LIFO) order.

User-written condition handlers can also be unregistered with the Unregister a User-Written Condition Handler (CEEHDLU) API.

Common Exceptions and Some of Their Causes

MCH1202 Decimal data error:

- A numeric elementary item has been used as a source when no valid data has been previously stored in it. The item should have a VALUE clause, or a MOVE statement should be used to initialize its value.
- An attempt has been made to place nonnumeric data in a numeric item.
- Bad data was written to a subfile earlier in the program. The subfile data is not validated until it is written to the display, so the 1202 error can occur on the WRITE of a subfile control record, but the bad data was actually put to the subfile earlier.

MCH0601 Pointer exceptions:

- Part of a Linkage section item extended beyond the space allocated.
For example, if you set the address of a Linkage Section item, and one or more of its elementary data items extend beyond the space with a MOVE to the elementary data item, MCH0601 is issued.

For more information on using pointers, refer to Chapter 14, "Using Pointers in an ILE COBOL Program," on page 335.

MCH0602 Pointer alignment:

- The pointer alignment in the Working-Storage Section of the calling program does not match the alignment in the Linkage Section of the called program. Alignment must be on a 16-byte boundary.

For more information on using pointers, refer to Chapter 14, "Using Pointers in an ILE COBOL Program," on page 335.

MCH0603 Range check error:

- Either the subscript value is less than the lower bound of the array, or greater than the upper bound of the array, or the compound operand defined a character string outside the bounds of the base character string.

MCH3601 Pointer error:

- A reference is made to a record or a field within a record and the associated file has been closed or has never been opened.

For example, the OPEN for the file was unsuccessful and the processing of any other I/O statement for that file is attempted. The file status should be checked before any other I/O is attempted.

CPF2415 End of requests:

- An attempt has been made to accept input from the job input stream while the system is running in batch mode and no input is available.

Recovery After a Failure

Some recovery can take place after a failure. Two areas in which such recovery can take place are:

- Recovery of files with commitment control
- TRANSACTION file recovery.

Recovery of Files with Commitment Control

When the system is restarted after a failure, files under commitment control are automatically restored to their status at the last commitment boundary. For additional information about commitment control, see "Using Commitment Control" on page 417.

Commitment control can be scoped at two levels, the activation group level and the job level. Refer to the section "Commitment Control Scoping" in *ILE Concepts* for further information.

If a job or activation group ends abnormally (either because of user or system error), files under commitment control are restored as part of job or activation group termination to the files' status at the last commitment boundary. The commitment control boundary is determined by the commitment control scope chosen for the program.

Because files under commitment control are rolled back after system or process failure, this feature can be used to help in restarting. You can create a separate record to store data that may be useful should it become necessary to restart a job. This restart data can include items such as totals, counters, record key values, relative key values, and other relevant processing information from an application.

If you keep the restart data mentioned above in a file under commitment control, the restart data will also be permanently stored in the database when a COMMIT statement is issued. When a ROLLBACK occurs after job or process failure, you can retrieve a record of the extent of processing successfully processed before failure. Note that the above method is only a suggested programming technique and will not always be suitable, depending on the application.

TRANSACTION File Recovery

In some cases, you can recover from I/O errors on TRANSACTION files without intervention by the operator, or the varying off/varying on of workstations or communications devices.

For potentially recoverable I/O errors on TRANSACTION files, the system initiates action in addition to the steps that must be taken in the application program to attempt error recovery. For more information about action taken by the system, see the *Communications Management*.

By examining the file status after an I/O operation, the application program can determine whether a recovery from an I/O error on the TRANSACTION file is possible. If the File Status Key has a value of 9N, the application program may be able to recover from the I/O error. A recovery procedure must be coded as part of the application program and varies depending on whether a single device was acquired by the TRANSACTION file or whether multiple devices were attached.

For a file with one acquired device:

1. Close the TRANSACTION file with the I/O error.
2. Reopen the file.
3. Process the steps necessary to retry the failing I/O operation. This may involve a number of steps, depending on the type of program device used. (For example, if the last I/O operation was a READ, you may have to repeat one or more WRITE statements, which were processed prior to the READ statement.) For more information on recovery procedures, see the *ICF Programming* manual.

For a display file with multiple devices acquired:

1. DROP the program device that caused the I/O error on the TRANSACTION file.
2. ACQUIRE the same program device.
3. See Step 3 above.

For an ICF file with multiple devices acquired:

1. ACQUIRE the same program device.
2. See Step 3 above.

For a display file with multiple devices acquired:

Application program recovery attempts should typically be tried only once.

If the recovery attempt fails:

- If the file has only one program device attached, terminate the program through processing of the STOP RUN, EXIT PROGRAM or GOBACK statement, and attempt to locate the source of the error.
- If the file has multiple acquired program devices, you may want to do one of the following:

- Continue processing without the program device that caused the I/O error on the TRANSACTION file, and reacquire the device later.
- End the program.

For a description of major and minor return codes that may help in diagnosing I/O errors on the TRANSACTION file, see the *ICF Programming* manual.

Figure 92 on page 390 gives an example of an error recovery procedure.

```

.....+.....1.....+.....2.....+.....3.....+.....4.....+.....5.....+.....6.....+.....7.....+.....8
A*  DISPLAY FILE FOR ERROR RECOVERY EXAMPLE
A*
A
A          R FORMAT1                INDARA
A          CF01(01 'END OF PROGRAM')
A*
A          12 28'ENTER INPUT '
A          INPUTFLD      5  I 12 42
A          20 26'F1 - TERMINATE'

```

Figure 91. Example of Error Recovery Procedure -- DDS

```

Source
STMT PL SEQNBR -A 1 B.+....2....+....3....+....4....+....5....+....6....+....7..IDENTFCN S COPYNAME CHG DATE
1 000100 IDENTIFICATION DIVISION.
2 000200 PROGRAM-ID. RECOVERY.
3 000300 ENVIRONMENT DIVISION.
4 000400 CONFIGURATION SECTION.
5 000500 SOURCE-COMPUTER. IBM-ISERIES.
6 000600 OBJECT-COMPUTER. IBM-ISERIES.
7 000700 INPUT-OUTPUT SECTION.
8 000800 FILE-CONTROL.
9 000900 SELECT RECOVFILE
10 001000 ASSIGN TO WORKSTATION-RECVFILE-SI
11 001100 ORGANIZATION IS TRANSACTION
12 001200 ACCESS MODE IS SEQUENTIAL
13 001300 FILE STATUS IS STATUS-FLD, STATUS-FLD-2
14 001400 CONTROL-AREA IS CONTROL-FLD.
15 001500 SELECT PRINTER-FILE
16 001600 ASSIGN TO PRINTER-QPRINT.
001700
17 001800 DATA DIVISION.
18 001900 FILE SECTION.
19 002000 FD RECOVFILE.
20 002100 01 RECOV-REC.
002200 COPY DDS-ALL-FORMATS OF RECVFILE.
21 +000001 05 RECVFILE-RECORD PIC X(5). <-ALL-FMFS
+000002* INPUT FORMAT:FORMAT1 FROM FILE RECVFILE OF LIBRARY CBLGUIDE <-ALL-FMFS
+000003* <-ALL-FMFS
22 +000004 05 FORMAT1-I REDEFINES RECVFILE-RECORD. <-ALL-FMFS
23 +000005 06 INPUTFLD PIC X(5). <-ALL-FMFS
+000006* OUTPUT FORMAT:FORMAT1 FROM FILE RECVFILE OF LIBRARY CBLGUIDE <-ALL-FMFS
+000007* <-ALL-FMFS
+000008* 05 FORMAT1-0 REDEFINES RECVFILE-RECORD. <-ALL-FMFS
002300
24 002400 FD PRINTER-FILE.
25 002500 01 PRINTER-REC.
26 002600 05 PRINTER-RECORD PIC X(132).
002700
27 002800 WORKING-STORAGE SECTION.
002900
28 003000 01 I-O-VERB PIC X(10).
29 003100 01 STATUS-FLD PIC X(2).
30 003200 88 NO-ERROR VALUE "00".
31 003300 88 ACQUIRE-FAILED VALUE "9H".
32 003400 88 TEMPORARY-ERROR VALUE "9N".
33 003500 01 STATUS-FLD-2 PIC X(4).
34 003600 01 CONTROL-FLD.
35 003700 05 FUNCTION-KEY PIC X(2).
36 003800 05 PGM-DEVICE-NAME PIC X(10).
37 003900 05 RECORD-FORMAT PIC X(10).
38 004000 01 END-INDICATOR PIC 1 INDICATOR 1
004100 VALUE B"0".
39 004200 88 END-NOT-REQUESTED VALUE B"0".
40 004300 88 END-REQUESTED VALUE B"1".
41 004400 01 USE-PROC-FLAG PIC 1
004500 VALUE B"1".

```

Figure 92. Example of Error Recovery Procedure (Part 1 of 3)

```

5722WDS V5R4M0 060210 LN IBM ILE COBOL CBLGUIDE/RECOVERY ISERIES1 06/02/15 13:48:21 Page 3
STMT PL SEQNBR -A 1 B...2...+...3...+...4...+...5...+...6...+...7..IDENTFCN S COPYNAME CHG DATE
42 004600 88 USE-PROC-NOT-EXECUTED VALUE B"0".
43 004700 88 USE-PROC-EXECUTED VALUE B"1".
44 004800 01 RECOVERY-FLAG PIC 1
004900 VALUE B"0".
45 005000 88 NO-RECOVERY-DONE VALUE B"0".
46 005100 88 RECOVERY-DONE VALUE B"1".
47 005200 01 HEADER-LINE.
48 005300 05 FILLER PIC X(60)
005400 VALUE SPACES.
49 005500 05 FILLER PIC X(72)
005600 VALUE "ERROR REPORT".
50 005700 01 DETAIL-LINE.
51 005800 05 FILLER PIC X(15)
005900 VALUE SPACES.
52 006000 05 DESCRIPTION PIC X(25)
006100 VALUE SPACES.
53 006200 05 DETAIL-VALUE PIC X(92)
006300 VALUE SPACES.
54 006400 01 MESSAGE-LINE.
55 006500 05 FILLER PIC X(15)
006600 VALUE SPACES.
56 006700 05 DESCRIPTION PIC X(117)
006800 VALUE SPACES.
57 006900 PROCEDURE DIVISION.
58 007000 DECLARATIVES.
007100 HANDLE-ERRORS SECTION.
007200 USE AFTER STANDARD ERROR PROCEDURE ON RECOVFILE. 1
007300 DISPLAY-ERROR.
59 007400 SET USE-PROC-EXECUTED TO TRUE.
60 007500 WRITE PRINTER-REC FROM HEADER-LINE
007600 AFTER ADVANCING PAGE
007700 END-WRITE
61 007800 MOVE "ERROR OCCURED IN" TO DESCRIPTION OF DETAIL-LINE.
62 007900 MOVE I-O-VERB TO DETAIL-VALUE OF DETAIL-LINE.
63 008000 WRITE PRINTER-REC FROM DETAIL-LINE
008100 AFTER ADVANCING 5 LINES
008200 END-WRITE
64 008300 MOVE "FILE STATUS =" TO DESCRIPTION OF DETAIL-LINE.
65 008400 MOVE STATUS-FLD TO DETAIL-VALUE OF DETAIL-LINE. 2
66 008500 WRITE PRINTER-REC FROM DETAIL-LINE
008600 AFTER ADVANCING 2 LINES
008700 END-WRITE
67 008800 MOVE "EXTENDED FILE STATUS =" TO DESCRIPTION OF DETAIL-LINE.
68 008900 MOVE STATUS-FLD-2 TO DETAIL-VALUE OF DETAIL-LINE.
69 009000 WRITE PRINTER-REC FROM DETAIL-LINE
009100 AFTER ADVANCING 2 LINES
009200 END-WRITE
70 009300 MOVE "CONTROL-AREA =" TO DESCRIPTION OF DETAIL-LINE.
71 009400 MOVE CONTROL-FLD TO DETAIL-VALUE OF DETAIL-LINE.
72 009500 WRITE PRINTER-REC FROM DETAIL-LINE
009600 AFTER ADVANCING 2 LINES
009700 END-WRITE.
009800 CHECK-ERROR.
73 009900 IF TEMPORARY-ERROR AND NO-RECOVERY-DONE THEN
74 010000 MOVE "***ERROR RECOVERY BEING ATTEMPTED***" 3

```

Figure 92. Example of Error Recovery Procedure (Part 2 of 3)

```

5722WDS V5R4M0 060210 LN  IBM ILE COBOL                CBLGUIDE/RECOVERY      ISERIES1  06/02/15 13:48:21      Page 4
STMT PL SEQNBR -A 1 B..+...2...+...3...+...4...+...5...+...6...+...7..IDENTFCN  S COPYNAME  CHG DATE
010100          TO DESCRIPTION OF MESSAGE-LINE
75  010200      WRITE PRINTER-REC FROM MESSAGE-LINE
010300          AFTER ADVANCING 3 LINES
010400          END-WRITE
76  010500      SET RECOVERY-DONE TO TRUE
77  010600      DROP PGM-DEVICE-NAME FROM RECOVFILE
78  010700      ACQUIRE PGM-DEVICE-NAME FOR RECOVFILE 4
010800      ELSE
79  010900          IF RECOVERY-DONE THEN 5
80  011000              MOVE "***ERROR AROSE FROM RETRY AFTER RECOVERY***"
011100              TO DESCRIPTION OF MESSAGE-LINE
81  011200          WRITE PRINTER-REC FROM MESSAGE-LINE
011300          AFTER ADVANCING 3 LINES
011400          END-WRITE
82  011500              MOVE "***PROGRAM ENDED***"
011600              TO DESCRIPTION OF MESSAGE-LINE
83  011700          WRITE PRINTER-REC FROM MESSAGE-LINE
011800          AFTER ADVANCING 2 LINES
011900          END-WRITE
84  012000          CLOSE RECOVFILE
012100          PRINTER-FILE
85  012200          STOP RUN
012300      ELSE
86  012400          SET NO-RECOVERY-DONE TO TRUE
012500          END-IF
012600      END-IF
87  012700          MOVE "***EXECUTION CONTINUES***"
012800          TO DESCRIPTION OF MESSAGE-LINE.
88  012900          WRITE PRINTER-REC FROM MESSAGE-LINE
013000          AFTER ADVANCING 2 LINES
013100          END-WRITE.
013200      END DECLARATIVES.
013300
013400      MAIN-PROGRAM SECTION.
013500      MAINLINE.
89  013600          MOVE "OPEN" TO I-O-VERB.
90  013700          OPEN I-O RECOVFILE
013800          OUTPUT PRINTER-FILE.
91  013900          PERFORM I-O-PARAGRAPH UNTIL END-REQUESTED. 6
92  014000          CLOSE RECOVFILE
014100          PRINTER-FILE.
93  014200          STOP RUN.
014300
014400      I-O-PARAGRAPH.
94  014500          PERFORM UNTIL USE-PROC-NOT-EXECUTED OR NO-RECOVERY-DONE 7
95  014600              MOVE "WRITE" TO I-O-VERB
96  014700              SET USE-PROC-NOT-EXECUTED TO TRUE
97  014800              WRITE RECOV-REC FORMAT IS "FORMAT1"
014900              INDICATOR IS END-INDICATOR
015000          END-WRITE
015100          END-PERFORM
98  015200          MOVE "READ" TO I-O-VERB.
99  015300          SET USE-PROC-NOT-EXECUTED TO TRUE.
100 015400          SET NO-RECOVERY-DONE TO TRUE.
101 015500          READ RECOVFILE FORMAT IS "FORMAT1"
015600          INDICATOR IS END-INDICATOR 8
015700          END-READ
102 015800          IF NO-ERROR THEN
103 015900              PERFORM SOME-PROCESSING
016000          END-IF.
016100
016200      SOME-PROCESSING.
016300*      (Insert some database processing, for example.)
016400
          * * * * * E N D   O F   S O U R C E * * * * *

```

Figure 92. Example of Error Recovery Procedure (Part 3 of 3)

- 1** This defines processing that takes place when an I/O error occurs on RECOVFILE.
- 2** This prints out information to help in diagnosing the problem.
- 3** If the file-status equals 9N (temporary error), and no previous error recovery has been attempted for this I/O operation, error recovery is now attempted.

- 4** Recovery consists of dropping, then reacquiring, the program device on which the I/O error occurred.
- 5** To avoid program looping, recovery is not attempted now if it was attempted previously.
- 6** The mainline of the program consists of writing to and reading from a device until the user signals an end to the program by pressing F1.
- 7** If the WRITE operation failed but recovery was done, the WRITE is attempted again.
- 8** If the READ operation failed, processing will continue by writing to the device again, and then attempting the READ again.

Handling Errors in Operations Using Null-Capable Fields

When a null-capable field is referenced in a program, the ILE COBOL compiler does not check if the field is actually null or not. It is the responsibility of the programmer to ensure that fields referenced as null-capable actually contain or do not contain null values (in other words, a 0 or 1) in the null map and null key map for the fields. If a field is defined in a program as null-capable, but is not defined as null-capable in the database, no checking is done by ILE COBOL, and whatever is in the field is used at the time of execution. At program initialization, fields for externally described files are set to zero. For program described files, it is the programmer's responsibility to ensure that their null-capable fields are set to zero at program initialization.

If the file *is* null-capable, and the ALWNULL attribute has *not* been specified, when you attempt to read a record that has a null value, the read will fail with a file status of 90.

If the file *is not* null-capable and the ALWNULL attribute of the ASSIGN clause is specified, the null map and null key map are returned from the database as zeros. And, when the null maps and null key maps are passed to the database, they are ignored.

Handling Errors in Locale Operations

There are three types of locales in ILE COBOL:

- DEFAULT locale
- CURRENT locale
- Specific locales.

Specific locales are referenced in the SPECIAL-NAMES paragraph and in the SET LOCALE statement. An example of a specific locale in the SPECIAL-NAMES paragraph is:

```
SPECIAL-NAMES.  LOCALE "MYLOCALE" IN LIBRARY "MYLIB" IS newlocale.  
DATA DIVISION.  
WORKING-STORAGE SECTION.  
01 group-item.  
   05 num-edit PIC $99.99 SIZE 8 LOCALE newlocale.  
PROCEDURE DIVISION.  
   MOVE 40 to num-edit.
```

In the above example a specific locale mnemonic-name *newlocale* has been defined. This mnemonic-name is used in the definition of variable *num-edit*. Since

the mnemonic-name is referenced in the program, the first time the above program is called, the ILE COBOL runtime tries to find the locale MYLOCALE in library MYLIB and load it into memory.

```
# A locale on the i5/OS is an object of type *LOCALE, and like other i5/OS objects
# exists within a library and has a specific authority assigned to it. Any locale
# mnemonic-name that is defined and referenced in the COBOL program will be
# resolved the first time the program is called. The possible types of failures include:
# • Locale does not exist in the specified library
# • Library for locale does not exist
# • Not enough authority to the locale or locale library.
```

These types of failures are typical of most other i5/OS objects. In any of the above scenarios an escape message (usually LNR7096) is issued. Once a locale object is located it must be loaded by the ILE COBOL run-time. Loading a locale object requires the allocation of various spaces, if space is not available an escape message is issued (usually LNR7070).

The SET LOCALE has several possible forms, the two basic forms that can reference a specific locale are:

```
SPECIAL-NAMES. LOCALE "ALocale" IS alocale.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 group-item.
   05 num-edit PIC +$9(4).99 SIZE 10 LOCALE alocale.
* num-edit2 is based on the current locale
   05 num-edit2 PIC +$9(4).99 SIZE 10 LOCALE.
   05 locale-name PIC X(10) VALUE "FRANCE".
   05 locale-lib PIC X(10) VALUE "MYLIB".
   MOVE 345.67 TO num-edit.
* set the current locale to "ALocale" in library "*LIBL".
   SET LOCALE LC_ALL FROM alocale.
   MOVE 678.02 TO num-edit2.

* set the current locale to "FRANCE" in library "MYLIB".
   SET LOCALE LC_ALL FROM locale-name
   IN LIBRARY locale-lib.
   MOVE 678.02 TO num-edit2.
```

The first form references a locale mnemonic-name in the SPECIAL-NAMES paragraph, and just like in the previous example is resolved and loaded the first time the program is called. In the second SET statement, the locale name is taken from the contents of identifier locale-name and the library where the locale exists is taken from the contents of identifier locale-lib. In this case the resolve and load of the locale object is done when the SET statement is run. With this form of the SET statement if the locale can not be resolved an escape message (usually LNR7098) is issued. It is issued for the same type of reasons as LNR7096 mentioned previously.

Part 3. ILE COBOL Input-Output Considerations

Chapter 17. Defining Files

This chapter describes how to:

- Define program-described files
- Define externally described files
- Describe files using Data Description Specifications (DDS)
- Use externally described files in an ILE COBOL program.

Types of File Descriptions

The key element for all I/O operations on the i5/OS is the file. The operating
system maintains a description of each file that is used by a program. The
description of the file to the operating system includes information about the type
of file, such as database or a device, the length of the records in the file, and a
description of each field and its attributes. The file is described at the field level to
the operating system through IDDU, SQL/400® commands, or DDS. If you create a
file (for instance, by using the CRTPF command) without specifying DDS for it, the
file still has a field description. The single field has the same name as the file, and
has the record length you specified in the create command.

You can define a file in two ways:

- A **program-described file** is described by the programmer at the field level in the Data Division within the ILE COBOL program.
- For an **externally described file**, the ILE COBOL compiler uses the description of the file on the system to generate the ILE COBOL source statements in the Data Division that describe the file at the field level within the ILE COBOL program. The file must be created before you compile the program.

Both externally described files and program-described files must be defined in the ILE COBOL program within the INPUT-OUTPUT SECTION and the FILE SECTION. Record descriptions in the FILE SECTION for externally described files are defined with the Format 2 COPY statement. Only field-level descriptions are extracted. When EXTERNALLY-DESCRIBED-KEY is specified as RECORD KEY, the fields that make up RECORD KEY are also extracted from DDS. For more information on the Format 2 COPY statement, see *IBM Rational Development Studio for i: ILE COBOL Reference*.

Actual file processing within the Procedure Division is the same, if the file is externally described or program-described.

Defining Program-Described Files

Records and fields for a program-described file are described by coding record descriptions directly in the FILE SECTION of the ILE COBOL program instead of using the Format 2 COPY statement.

The file must exist on the system before the program can run. The only exception is when you use dynamic file creation, by specifying OPTION(*CRTF) on the CRTCLMOD/CRTBNDCBL command. For more information, refer to the description of the OPTION parameter in “Parameters of the CRTCLMOD Command” on page 28.

To create a file, use one of the Create File commands. DDS can be used with the Create File commands. For an ILE COBOL indexed file, a keyed access path must be created. Specify a key in DDS when the file is created. The record key in the ILE COBOL program must match the key defined when the file was created. If these key values do not match, the file operation may still proceed, but with the wrong record key being passed to the system. If the wrong record key happens to contain an apparently correct key value, the input/output operation will be performed successfully, but on the wrong data. Thus, the integrity of your data may be compromised. To prevent this problem from happening, you should use externally described files whenever possible.

Defining Externally Described Files

The external description for a 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 come from the external file description and from the IBM i command you use to create the file.

Externally described files offer the following advantages over program-described files:

- Less coding in ILE COBOL programs. If the same file is used by many programs, the fields can be defined once to the operating system, and then used by all the programs. This eliminates the need to code a separate record description for each program that uses the file.
- Reduces the chance of programming error. You can often update programs by changing the file's record format and then recompiling the programs that use the file without changing any coding in the program.
- Level checking of the file description. A level check of the description of the file in the ILE COBOL program and the actual file on the system is performed when the file is opened (unless LVLCHK(*NO) is specified on the create file command or an override command). If the description of the file in the program does not match the actual file, the open operation will fail with a file status of 39.
- For indexed files, if EXTERNALLY-DESCRIBED-KEY is specified in the RECORD KEY clause, you can ensure that the record key occupies the same position in the actual file as in your ILE COBOL program's description of the file. Also, you can use noncontiguous keys, which is not possible with program-described files.
- Improved documentation. Programs using the same files use consistent record format and field names.
- Any editing to be processed on externally described output files can be specified in DDS.

Before you can use an externally described file in your program, you must create a DDS to describe the file and create the actual file itself.

Describing Files Using Data Description Specifications (DDS)

You can use Data Description Specifications (DDS) to describe files at the field level to the operating system. In DDS, each record format in an externally described file is identified by a unique record format name.

The record format specifications 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 DDS. The field description generally includes the field name, the field type (character, binary, external decimal, internal decimal, internal floating-point), and the field length (including the number of decimal positions in a numeric field). Instead of being specified in the record format for a physical or logical file, the field attributes can be defined in a field reference file. (See Figure 93 on page 400.)

The keys for a record format are specified in DDS. When you use a Format 2 COPY statement, a table of comments is generated in the source program listing showing how the keys for the format are defined in DDS.

In addition, DDS keywords can be used to:

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

#

For a complete list of the DDS keywords that are valid for a database file, refer to the *Database and File Systems* category in the **i5/OS Information Center** at this Web site -<http://www.ibm.com/systems/i/infocenter/>.

.....	1.....	2.....	3.....	4.....	5.....	6.....	7.....	8.....
A**FLDREF	DSTREF	DISTRIBUTION APPLICATION FIELDS REFERENCE						
A	R DSTREF	TEXT('DISTRIBUTION FIELD REF')						
A* COMMON FIELDS USED AS REFERENCE								
1	A	BASDAT	6	0	EDTCDE(Y)			
A					TEXT('BASE DATE FIELD')			
A* FIELDS USED BY CUSTOMER MASTER FILE								
2	A	CUST	5		CHECK(MF)			
A					COLHDG('CUSTOMER' 'NUMBER')			
3	A	NAME	20		COLHDG('CUSTOMER NAME')			
A		ADDR	R		REFFLD(NAME)			
A					COLHDG('CUSTOMER ADDRESS')			
A		CITY	R		REFFLD(NAME)			
A					COLHDG('CUSTOMER CITY')			
2	A	STATE	2		CHECK(MF)			
A					COLHDG('STATE')			
A		SRHCOD	6		CHECK(MF)			
A					COLHDG('SEARCH' 'CODE')			
A					TEXT('CUSTOMER NUMBER SEARCH CODE')			
2	A	ZIP	5	0	CHECK(MF)			
A					COLHDG('ZIP' 'CODE')			
4	A	CUSTYP	1	0	RANGE(1 5)			
A					COLHDG('CUST' 'TYPE')			
A					TEXT('CUSTOMER TYPE 1=GOV 2=SCH 3=B+			
A					US 4=PT 5=OTH')			
5	A	ARBAL	8	2	COLHDG('ACCTS REC' 'BALANCE')			
A					EDTCDE(J)			
6	A	ORDBAL	R		REFFLD(ARBAL)			
A					COLHDG('A/R AMT IN' 'ORDER FILE')			
A		LSTAMT	R		REFFLD(ARBAL)			
A					COLHDG('LAST' 'AMOUNT' 'PAID')			
7	A	LSTDAT	R		TEXT('LAST AMOUNT PAID IN A/R')			
A					REFFLD(ARBAL)			
A					COLHDG('LAST' 'DATE' 'PAID ')			
A					TEXT('LAST DATE PAID IN A/R')			
A		CRDLMT	8	2	COLHDG('CUSTOMER' 'CREDIT' 'LIMIT')			
A					EDTCDE(J)			
A		SLSYR	10	2	COLHDG('CUSTOMER' 'SALES' 'THIS YEAR')			
A					EDTCDE(J)			
A		SLSLYR	10	2	COLHDG('CUSTOMER' 'SALES' 'LAST YEAR')			
A					EDTCDE(J)			

Figure 93. Example of a Field Reference File

This example of a field reference file (Figure 93) shows the definitions of the fields that are used by the CUSMSTL (customer master logical) file, which is shown in Figure 94 on page 401. 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 a ILE COBOL program, the COBOL-generated field is compatible with the data type specified in the DDS. The field is edited when the record is written. When the field is used in a program-described output file, compatibility with the DDS fields in the file is the user's responsibility. When DDS is not used to create the file, appropriate editing of the field in the ILE COBOL program is also the user's responsibility.
- 2** The CHECK(MF) entry specifies that the field is a mandatory fill field when it is entered from a display workstation. Mandatory fill means that all characters for the field must be entered from the display workstation.

- 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 Application Development ToolSet tools.
- 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.

Using Externally Described Files in an ILE COBOL Program

You can incorporate the file description in your program by coding a Format 2 COPY statement. The information from the external description is then retrieved by the ILE COBOL compiler, and an ILE COBOL data structure is generated.

The following pages provide examples of DDS usage and the ILE COBOL code that would result from the use of a Format 2 COPY statement. (See the *IBM Rational Development Studio for i: ILE COBOL Reference* for a detailed description of the Format 2 COPY statement.)

- Figure 94 shows the DDS for a logical file and Figure 95 on page 402 shows the ILE COBOL code generated.
- Figure 96 on page 403 describes the same file but includes the ALIAS keyword, and Figure 97 on page 404 shows the ILE COBOL code generated.

```

...+...1...+...2...+...3...+...4...+...5...+...6...+...7...+...8
1 A** LOGICAL CUSMSTL  CUSTOMER MASTER FILE
A
A           3 R CUSREC
A
A           CUST
A           NAME
A           ADDR
A           CITY
A           STATE
A           ZIP
A           SRHCO
A           CUSTYP
A           ARBAL
A           ORDBAL
A           LSTAMT
A           LSTDAT
A           CRDLMT
A           SLSYR   5
A           SLSLYR
A           4 K CUST

```

Figure 94. Example of Data Description Specifications for a Logical File

- 1** A logical file for processing the customer master physical file (CUSMSTP) is defined and named CUSMSTL.
- 2** The UNIQUE keyword indicates that duplicate key values for this file are not allowed.

- 3 One record format (CUSREC) is defined for the CUSMSTL file, which is to be based upon the physical file CUSMSTP.
- 4 The CUST field is identified as the key field for this file.
- 5 If field attributes (such as length, data type, and decimal positions) are not specified in the DDS for a logical file, the attributes are obtained from the corresponding field in the physical file. Any field attributes specified in the DDS for the logical file override the attributes for the corresponding field in the physical file. The definition of the fields in the physical file could refer to a field reference file. A field reference file is a data description file consisting of field names and their definitions, such as size and type. When a field reference file is used, the same fields that are used in multiple record formats have to be defined only once in the field reference file. For more information on field reference files, see the *DB2 Universal Database for AS/400* section of the *Database and File Systems* category in the **i5/OS Information Center** at this Web site - <http://www.ibm.com/systems/i/infocenter/>.

#

Figure 93 on page 400 shows an example of a field reference file that defines the attributes of the fields used in the database file.

S o u r c e			
STMT	PL	SEQNBR	-A 1 B.+. . . . 2. . . . +. . . . 3. . . . +. . . . 4. . . . +. . . . 5. . . . +. . . . 6. . . . +. . . . 7. . . . IDENTFCN S COPYNAME CHG DATE
18	000200	01	CUSTOMER-INVOICE-RECORD.
	000210		COPY DDS-CUSREC OF CUSMSTL.
	+000001*		I-O FORMAT:CUSREC FROM FILE CUSMSTL OF LIBRARY TESTLIB CUSREC
	+000002*		CUSTOMER MASTER RECORD CUSREC
	+000003*		USER SUPPLIED KEY BY RECORD KEY CLAUSE CUSREC
19	+000004	05	CUSREC. CUSREC
20	+000005	06	CUST PIC X(5). CUSREC
	+000006*		CUSTOMER NUMBER CUSREC
21	+000007	06	NAME PIC X(25). CUSREC
	+000008*		CUSTOMER NAME CUSREC
22	+000009	06	ADDR PIC X(20). CUSREC
	+000010*		CUSTOMER ADDRESS CUSREC
23	+000011	06	CITY PIC X(20). CUSREC
	+000012*		CUSTOMER CITY CUSREC
24	+000013	06	STATE PIC X(2). CUSREC
	+000014*		STATE CUSREC
25	+000015	06	ZIP PIC S9(5) COMP-3. CUSREC
	+000016*		ZIP CODE CUSREC
26	+000017	06	SRHCOD PIC X(6). CUSREC
	+000018*		CUSTOMER NUMBER SEARCH CODE CUSREC
27	+000019	06	CUSTYP PIC S9(1) COMP-3. CUSREC
	+000020*		CUSTOMER TYPE 1=GOV 2=SCH 3=BUS 4=PVT 5=OT CUSREC
28	+000021	06	ARBAL PIC S9(6)V9(2) COMP-3. CUSREC
	+000022*		ACCOUNTS REC. BALANCE CUSREC
29	+000023	06	ORDBAL PIC S9(6)V9(2) COMP-3. CUSREC
	+000024*		A/R AMT. IN ORDER FILE CUSREC
30	+000025	06	LSTAMT PIC S9(6)V9(2) COMP-3. CUSREC
	+000026*		LAST AMT. PAID IN A/R CUSREC
31	+000027	06	LSTDAT PIC S9(6) COMP-3. CUSREC
	+000028*		LAST DATE PAID IN A/R CUSREC
32	+000029	06	CRDLMT PIC S9(6)V9(2) COMP-3. CUSREC
	+000030*		CUSTOMER CREDIT LIMIT CUSREC
33	+000031	06	SLSYR PIC S9(8)V9(2) COMP-3. CUSREC
	+000032*		CUSTOMER SALES THIS YEAR CUSREC
34	+000033	06	SLSLYR PIC S9(8)V9(2) COMP-3. CUSREC
	+000034*		CUSTOMER SALES LAST YEAR CUSREC

Figure 95. Example of the Results of the Format 2 COPY Statement (DDS)

```

.....+.....1.....+.....2.....+.....3.....+.....4.....+.....5.....+.....6.....+.....7.....+.....8
A** LOGICAL CUSMSTL  CUSTOMER MASTER FILE
A                                     UNIQUE
A          R CUSREC                                     PFILE(CUSMSTP)
A                                     TEXT('CUSTOMER MASTER RECORD')
A          CUST                                     ALIAS(CUSTOMER_NUMBER)
A          NAME                                     1 ALIAS(CUSTOMER_NAME)
A          ADDR                                     ALIAS(ADDRESS)
A          CITY
A          STATE
A          ZIP
A          SRHCOB                                     ALIAS(SEARCH_CODE)
A          CUSTYP                                     ALIAS(CUSTOMER_TYPE)
A          ARBAL                                     ALIAS(ACCT_REC_BALANCE)
A          ORDBAL
A          LSTAMT
A          LSTDAT
A          CRDLMT
A          SLSYR
A          SLSLYR
A          K CUST

```

Figure 96. Example of Data Description Specifications with ALIAS

- 1 This is the name associated with the ALIAS keyword, which will be included in the program. Available through the DDS ALIAS option, an alias is an alternative name that allows a data name of up to 30 characters to be included in an ILE COBOL program.

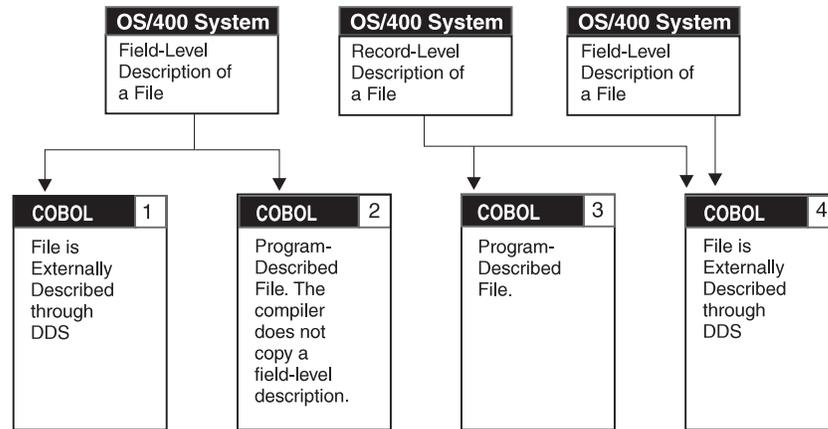
Source						
STMT	PL	SEQNBR	-A 1 B..+...2...+...3...+...4...+...5...+...6...+...7..	IDENTFCN	S COPYNAME	CHG DATE
18		002000 01	CUSTOMER-INVOICE-RECORD.			
		002100	COPY DDS-CUSREC OF CUSMSTL ALIAS.			
		+000001*	I-O FORMAT:CUSREC FROM FILE CUSMSTL OF LIBRARY TESTLIB		CUSREC	
		+000002*	CUSTOMER MASTER RECORD		CUSREC	
		+000003*	USER SUPPLIED KEY BY RECORD KEY CLAUSE		CUSREC	
19		+000004	05 CUSREC.		CUSREC	
20		+000005	06 CUSTOMER-NUMBER PIC X(5).		CUSREC	
		+000006*	CUSTOMER NUMBER		CUSREC	
21		+000007	06 CUSTOMER-NAME PIC X(25).		CUSREC	
		+000008*	CUSTOMER NAME		CUSREC	
22		+000009	06 ADDRESS-DDS PIC X(20).		CUSREC	
		+000010*	CUSTOMER ADDRESS		CUSREC	
23		+000011	06 CITY PIC X(20).		CUSREC	
		+000012*	CUSTOMER CITY		CUSREC	
24		+000013	06 STATE PIC X(2).		CUSREC	
		+000014*	STATE		CUSREC	
25		+000015	06 ZIP PIC S9(5) COMP-3.		CUSREC	
		+000016*	ZIP CODE		CUSREC	
26		+000017	06 SEARCH-CODE PIC X(6).		CUSREC	
		+000018*	CUSTOMER NUMBER SEARCH CODE		CUSREC	
27		+000019	06 CUSTOMER-TYPE PIC S9(1) COMP-3.		CUSREC	
		+000020*	CUSTOMER TYPE 1=GOV 2=SCH 3=BUS 4=PVT 5=OT		CUSREC	
28		+000021	06 ACCT-REC-BALANCE PIC S9(6)V9(2) COMP-3.		CUSREC	
		+000022*	ACCOUNTS REC. BALANCE		CUSREC	
29		+000023	06 ORDBAL PIC S9(6)V9(2) COMP-3.		CUSREC	
		+000024*	A/R AMT. IN ORDER FILE		CUSREC	
30		+000025	06 LSTAMT PIC S9(6)V9(2) COMP-3.		CUSREC	
		+000026*	LAST AMT. PAID IN A/R		CUSREC	
31		+000027	06 LSTDAT PIC S9(6) COMP-3.		CUSREC	
		+000028*	LAST DATE PAID IN A/R		CUSREC	
32		+000029	06 CRDLMT PIC S9(6)V9(2) COMP-3.		CUSREC	
		+000030*	CUSTOMER CREDIT LIMIT		CUSREC	
33		+000031	06 SLSYR PIC S9(8)V9(2) COMP-3.		CUSREC	
		+000032*	CUSTOMER SALES THIS YEAR		CUSREC	
34		+000033	06 SLSLYR PIC S9(8)V9(2) COMP-3.		CUSREC	
		+000034*	CUSTOMER SALES LAST YEAR		CUSREC	

Figure 97. Example of the Results of the Format 2 COPY Statement (DD) with the ALIAS Keyword

In addition to placing the external description of the file in the program through the use of the Format 2 COPY statement, you can also use standard record definition and redefinition to describe external files or to provide a group definition for a series of fields. It is the programmer's responsibility to ensure that program-described definitions are compatible with the external definitions of the file.

Figure 98 on page 405 shows how ILE COBOL programs can relate to files on the
i5/OS, making use of external file descriptions from DDS.

#



#

Figure 98. Example Showing How ILE COBOL Can Relate to i5/OS Files

- 1** The ILE COBOL program uses the field level description of a file that is defined to the operating system. You code a Format 2 COPY statement for the record description. At compilation time, the compiler copies in the external field-level description and translates it into a syntactically correct ILE COBOL record description. The file must exist at compilation time.
- 2** An externally described file is used as a program-described file in the ILE COBOL program. The entire record description for the file is coded in the ILE COBOL program. This file does not have to exist at compilation time.
- 3** A file is described to the operating system as far as the record level only. The entire record description must be coded in the ILE COBOL program. This file does not have to exist at compilation time.
- 4** A file name can be specified at compilation time, and a different file name can be specified at run time. An ILE COBOL Format 2 COPY statement generates the record description for the file at compilation time. At run time, a different library list or a file override command can be used so that a different file is accessed by the program. The file description copied in at compilation time is used to describe the input records used at run time.

Note: For externally described files, the two file formats must be the same. Otherwise, a level check error will occur.

Specifying Nonkeyed and Keyed Record Retrieval

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 (nonkeyed) access path or on a keyed sequence access path. For a complete description of the access paths for an externally described database file, see the *DB2 Universal Database for AS/400* section of the *Database and File Systems* category in the **i5/OS Information Center** at this Web site - <http://www.ibm.com/systems/i/infocenter/>.

The **arrival sequence access path** is based on the order in which the records are stored in the file. Records are added only to the end of the file.

For the **keyed sequence access path**, the sequence in which records are retrieved from the file is based on the contents of the key fields defined in the DDS for the file. For example, in the DDS shown in Figure 94 on page 401, 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 keyed sequence

access path, one or more fields can be defined in the DDS to be used as the key fields for a record format. Not all record formats in a file 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.

If you do not specify a format on the I/O operation then the key for a file is determined by the valid keys for the record formats in that file. The file's key is determined in the following manner:

- If all record formats in a file have the same number of key fields defined in DDS that are identical in attributes, the *key for the file* consists of all fields in the key for the record formats. (The corresponding fields do not have to have the same name.) For example, if the file has three record formats and the key for each record format 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 formats in the file do not have the same key fields, the key for the file consists of the key fields *common* to all record formats. For example, a file has three record formats 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.Then the file's key is field A, the key field common to all record formats.
- If no key field is common to all record formats, any keyed reference to the file will always return the first record in the file.

In ILE COBOL, you must specify a RECORD KEY for an indexed file to identify the record you want to process. ILE COBOL compares the key value with the key of the file or record, and processes the specified operation on the record whose key matches the RECORD KEY value.

When RECORD KEY IS EXTERNALLY-DESCRIBED-KEY is specified:

- If the FORMAT phrase is specified, the compiler builds the search argument from the key fields in the record area for the specified format
- If the FORMAT phrase is not specified, the compiler builds the search argument from the key fields in the record area for the first record format defined in the program for that file.

Note: For a file containing multiple key fields to be processed in ILE COBOL, the key fields must be contiguous in the record format used by the ILE COBOL program, except when RECORD KEY IS EXTERNALLY-DESCRIBED-KEY is specified.

Level Checking the Externally Described Files

When an ILE COBOL program uses an externally described file, the operating system provides a level check function (LVLCHK). This function ensures that the formats of the file have not changed since compilation time.

The compiler always provides the information required by level checking when an externally described file is used (that is, when a record description was defined for the file by using the Format 2 COPY statement). Only those formats that were copied by the Format 2 COPY statement under the FD for a file are level checked. The level check function will be initiated at run time based on the selection made on the create, change, or override file commands. The default on the create file command is to request level checking. If level checking was requested, level

checking occurs on a record format basis when the file is opened. If a level check error occurs, ILE COBOL sets a file status of 39.

When LVLCHK(*NO) is specified on the CRTxxxF, CHGxxxF, or OVRxxxF CL commands, and the file is re-created using an existing format, existing ILE COBOL programs that use that format may not work without recompilation, depending on the changes to the format.

You should use extreme caution when using files in ILE COBOL programs without level checking. You risk program failure and data corruption if you use ILE COBOL programs without level checking or recompiling.

Note: The ILE COBOL compiler does not provide level checking for program-described files.

For more information on level checking, refer to the *DB2 Universal Database for AS/400* section of the *Database and File Systems* category in the **i5/OS Information Center** at this Web site - <http://www.ibm.com/systems/i/infocenter/>.

Chapter 18. Processing Files

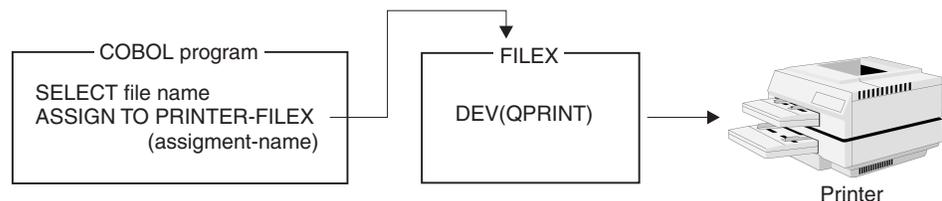
There are many ways in which files are used and processed by COBOL on the
i5/OS. This chapter describes how to:

• Associate files with input-output devices
• Change file attributes
• Redirect file input and output
• Lock and release files
• Unblock input records and block output records
• Share an open data path to access a file
• Use file status and feedback areas
• Use commitment control
• Sort and merge files
• Declare data items using CVTOPT data types.

Associating Files with Input-Output Devices

Files serve as the connecting link between a program and the device used for input and output. The actual device association is made at the time the file is opened. In some instances, this type of I/O control allows the user to change the attribute of the file (and, in some cases, change the device) used in a program without changing the program.

In the ILE COBOL language, the file name specified in the ASSIGNMENT-NAME entry of the ASSIGN clause of the file control entry is used to point to the file. This file name points to the system file description:

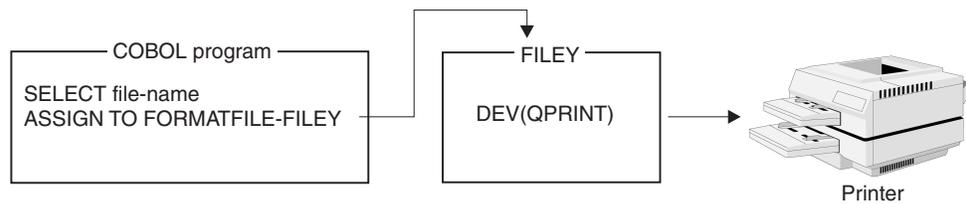


The ILE COBOL device name in the ASSIGN clause defines the ILE COBOL functions that can be processed on the selected file. At compilation time, certain ILE COBOL functions are valid only for a specific ILE COBOL device type; in this respect, ILE COBOL is device dependent. The following are examples of device dependency:

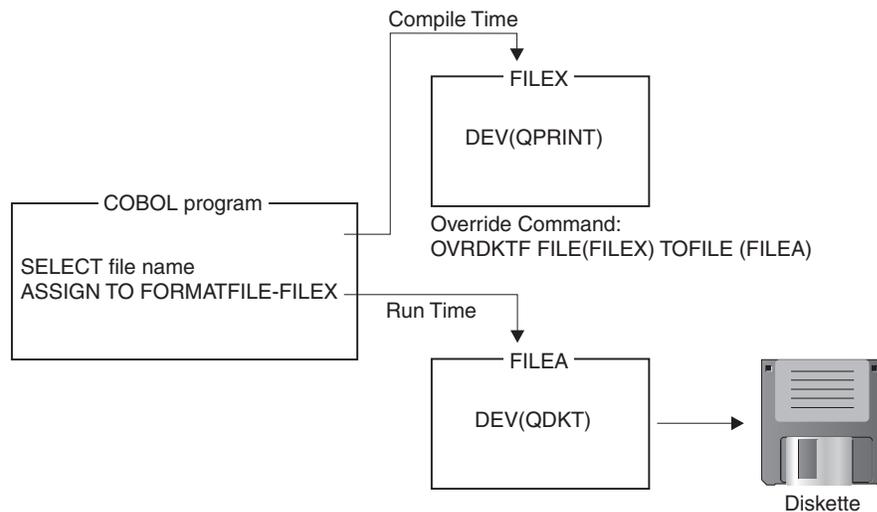
- SUBFILE operations are valid only for a WORKSTATION device.
- Indicators are valid only for WORKSTATION or FORMATFILE devices.
- LINAGE is valid only for a PRINTER device.
- OPEN INPUT WITH NO REWIND is valid only for a TAPEFILE device.

For example, assume that the file name FILEY is associated in the ILE COBOL program with the FORMATFILE device. The device FORMATFILE is an independent device type. Therefore, no line or page control specifications are valid so the ADVANCING phrase cannot be specified in the WRITE statement for a FORMATFILE file. When the program is run, the actual I/O device is specified in the description of FILEY. For example, the device might be a printer; only the

default line and page control or those defined in the DDS would be used:



CL 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 the user to specify one file at compilation time and another file at run time:



In the preceding example, the Override to Diskette File (OVRDKTF) command allows the program to run with an entirely different device file than was specified at compilation time.

Note: FORMATFILE devices cannot be used for input. Overriding input/output from a device that allows input, such as a DISKETTE device, to a FORMATFILE device may result in unexpected results if an input operation is attempted.

Not all file overrides are valid. At run time, checking occurs to ensure that the specifications within the ILE COBOL program are valid for the file being processed. If the specifications passed by the ILE COBOL program in the file control block and the I/O request are incorrect, the I/O operation will fail. The operating system allows some file redirections even if device specifics are contained in the program. For example, if the ILE COBOL device name is PRINTER and the actual file the program uses is not a printer, the operating system ignores the ILE COBOL print spacing and skipping specifications.

There are other file redirections that the operating system does not allow and that may cause the file to become unusable. For example, if the ILE COBOL device name is DATABASE or DISK and a keyed READ operation is specified in the program, the file becomes unusable if the actual file the program uses is not a disk or database file.

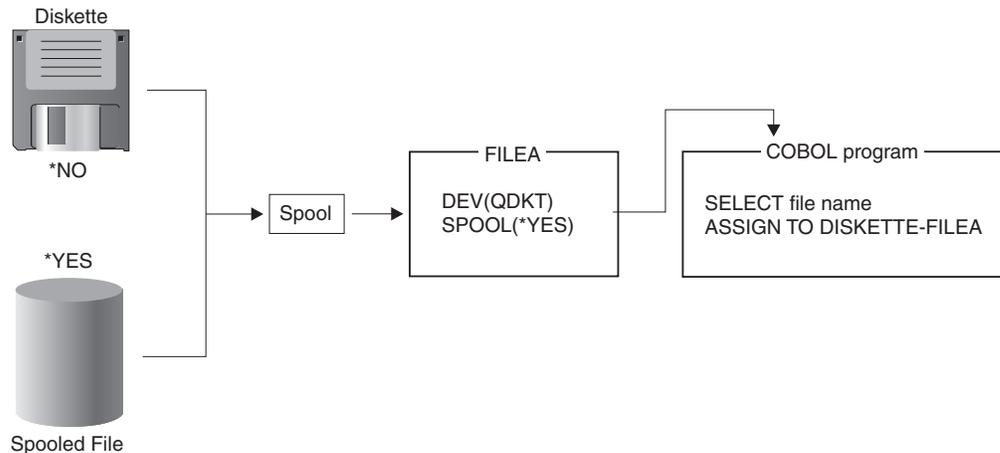
The i5/OS system provides for the use of input and output spooling functions.
 # Each i5/OS system file description contains a spool attribute that determines
 # whether spooling is used for the file at run time. The ILE COBOL program is not
 # aware that spooling is being used. The actual physical device from which a file is
 # read or to which a file is written is determined by the spool reader or the spool
 # writer. For more detailed information on spooling, refer to the *DB2 Universal
 # Database for AS/400* section of the *Database and File Systems* category in the **i5/OS
 # Information Center** at this Web site - [http://www.ibm.com/systems/i/
 # infocenter/](http://www.ibm.com/systems/i/infocenter/).

Specifying Input and Output Spooling

Input Spooling

Input spooling is valid only for inline data files in batch jobs. If the input data read by ILE COBOL comes from a spooled file, ILE COBOL is not aware of which device the data was spooled in from.

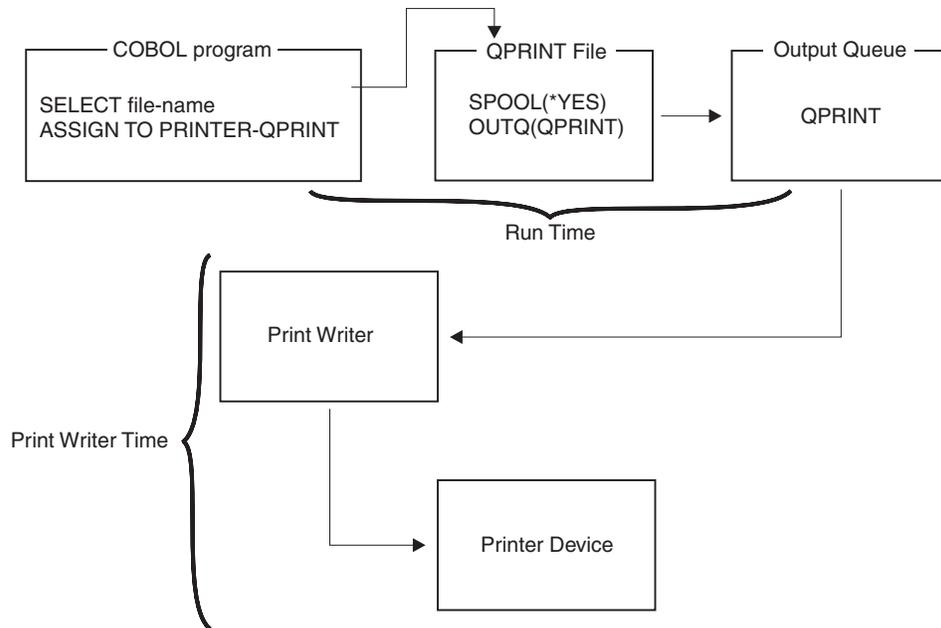
The data is read from a spooled inline file:



For more information on inline data files, refer to the *Database and File Systems*
 # category in the **i5/OS Information Center** at this Web site -[http://www.ibm.com/
 # systems/i/infocenter/](http://www.ibm.com/systems/i/infocenter/).

Output Spooling

Output spooling is valid for batch and interactive jobs. The description of the file that is specified in ILE COBOL by the system-name contains the specification for spooling as shown in the following example:



File override commands can be used at run time to override the spooling options
 # that are specified in the file description, such as the number of copies to be
 # printed. In addition, i5/OS spooling support allows you to redirect a file after the
 # program has run. For example, you can direct printer output to a different device,
 # such as a diskette.

Overriding File Attributes

You must specify any overrides before the file is opened by the ILE COBOL program. The system uses the file override command to determine the file to open and the attributes of the file. File overrides are scoped to the call level, the activation group level, or the job level.

For **call level** scoping, an override issued at a particular call level is effective for any invocations after the call level regardless of which activation group the invocations are in, and its effect ends when control is returned for the call level at which the override is issued.

For **activation group level** scoping, the override applies to all program objects running in that activation group and the override remains in effect until the activation group ends or the override is explicitly deleted.

Note: In the Default Activation Group (*DFTACTGRP), when activation group level scoping is specified, the override is actually scoped at the call level.

For **job level** scoping, the override applies to all program objects within the job, and it remains active until the job ends or the override is explicitly deleted.

Use the OVRSCOPE parameter of any override CL command to specify the scope of the override. If you do not explicitly specify the scope, the default scope of the override depends on where the override is issued. If the override is issued from the default activation group, it is scoped at the call level. If the override is issued from any other activation group, it is scoped to the activation group.

The simplest form of overriding a file is to override some attributes of the file. For example, FILE(OUTPUT) with COPIES(2) is specified when a printer file is created. Then, before the ILE COBOL program is run, the number of printed copies of output can be changed to 3. The override command is as follows:

```
OVRPRTF FILE(OUTPUT) COPIES(3)
```

Redirecting File Input and Output

Another form of file overriding is to redirect the ILE COBOL program to access a different file. When the override redirects the program to a file of the *same* type (such as a printer file to another printer file), the file is processed in the same manner as the original file.

When the override redirects the program to a file of a *different* type, the overriding file is processed in the same manner as the original file would have been processed. Device-dependent specifications in the ILE COBOL program that do not apply to the overriding device are ignored by the system.

Not all file redirections are valid. For example, an indexed file for an ILE COBOL program can only be overridden to another indexed file with a keyed access path.

Multiple member processing can be accomplished for a database file by overriding a database file to process all members. Note the following exceptions:

- A database source file used in the compilation of an ILE COBOL program cannot be overridden to process all members. Specifying OVRDBF MBR(*ALL) will result in the termination of the compilation.
- A database file used for a COPY statement cannot be overridden to process all members. Specifying OVRDBF MBR(*ALL) will cause the COPY statement to be ignored.

You must ensure that file overrides are applied properly. For more information on
valid file redirections, the device dependent characteristics ignored, and the
defaults assumed, refer to the *Programming* category in the **i5/OS Information**
Center at <http://www.ibm.com/systems/i/infocenter/>.

Locking and Releasing Files

The operating 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 a job step. You can place the file in a lock state with the Allocate Object (ALCOBJ) command.

By default, the operating system places the following lock states on database files when the files are opened by ILE COBOL programs:

OPEN Type	Lock State
INPUT	Shared-for-read
I-O	Shared-for-update
EXTEND	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 a shared-for-read or shared-for-update lock state.

The operating system places the shared-for-read lock on the device file and an exclusive-allow-read lock state on the device. Another user can open the file but cannot use the same device.

Note: When an ILE COBOL program opens a physical file for OUTPUT, that file will be subject to an exclusive lock for the period of time necessary to clear the member.

For more information on allocating resources and the lock states, refer to the
Database and File Systems category in the **i5/OS Information Center** at this Web site
-<http://www.ibm.com/systems/i/infocenter/>.

Locking and Releasing Records

When a READ is performed by an ILE COBOL program on a database file and the file is opened for I-O, a lock is placed on that record so that another program cannot update it. That is, the record can be read by another program if it opens a file for input, but not if it opens the file for I-O. Similarly, after a successful START operation for a file opened in I-O mode, a lock will be placed on the record at which the file is positioned.

For information about the duration of record lock with and without commitment control, refer to Figure 99 on page 420.

To prevent the READ or START statements from locking records on files opened in I-O (update) mode, you can use the NO LOCK phrase. The READ WITH NO LOCK statement unlocks records locked by a previous READ statement or START statement. Also, the record read by the READ WITH NO LOCK statement is not locked. The START WITH NO LOCK statement unlocks records locked by a previous START statement or READ statement. For more information about this phrase, refer to the section on the READ and START statements in the *IBM Rational Development Studio for i: ILE COBOL Reference*.

For a logical file based on one physical file, the lock is placed on the record in the physical file. If a logical file is based on more than one physical file, a lock is placed on one record in each physical file.

This lock applies not only to other programs, but also to the original program if it attempts to update the same underlying physical record through a second file.

Note: When a file with indexed or relative organization is opened for I-O, using random or dynamic access, a failed I/O operation on any of the I/O verbs except WRITE also unlocks the record. A WRITE operation is not considered an update operation; therefore, the record lock is not released.

For more information about releasing database records read for update, refer to the
Database and File Systems category in the **i5/OS Information Center** at this Web site
-<http://www.ibm.com/systems/i/infocenter/>.

Sharing an Open Data Path to Access a File

If you have already opened a file through another program in your routing step, your ILE COBOL program can use the same Open Data Path (ODP) to access the file.

#

Note: A job usually contains only one routing step. Routing steps are described in the *DB2 Universal Database for AS/400* section of the *Database and File Systems* category in the **i5/OS Information Center** at this Web site - <http://www.ibm.com/systems/i/infocenter/>.

The following rules apply to shared ODPs:

1. You must specify SHARE(*YES) in the command that creates the file (CRTxxxF), in a change command (CHGxxxF), or in an override (OVRxxxF) command for the file.
2. Once a file with a shared ODP has been opened for the first time by a program and remains open, subsequent shared OPEN operations within the same routing step run faster than standard OPEN operations. The speed of other I/O operations is not affected.
3. Your use of the file within your different programs should be consistent. Other programs using the same shared file will affect the current file position when they perform I/O operations on the file.

Unblocking Input Records and Blocking Output Records

A **block** contains more than one record. Unblocking of input records and blocking of output records occurs under the following conditions:

1. *NOBLK is specified on the OPTION parameter of the CRTCBMOD or CRTBNDCBL commands (with or without a BLOCK CONTAINS clause) and **all** of the following conditions are met:
 - a. ACCESS IS SEQUENTIAL is specified for the file.
 - b. The file is opened only for INPUT or OUTPUT in that program.
 - c. The file is assigned to DISK, DATABASE, DISKETTE, or TAPEFILE.
 - d. No START statements are specified for the file.For RELATIVE organization, blocking is not performed for OPEN OUTPUT. If you specify BLOCK CONTAINS, it is ignored. The system determines the number of records to be blocked.
2. *BLK is specified on the OPTION parameter of the CRTCBMOD or CRTBNDCBL commands (with or without a BLOCK CONTAINS clause) and **all** of the following conditions are met:
 - a. ACCESS IS SEQUENTIAL or ACCESS IS DYNAMIC is specified for the file.
 - b. The file is opened only for INPUT or OUTPUT in that program.
 - c. The file is assigned to DISK, DATABASE, DISKETTE, or TAPEFILE.

For RELATIVE organization, blocking is not performed for OPEN OUTPUT. The BLOCK CONTAINS clause controls the number of records to be blocked. In the case of DISKETTE files, the system always determines the number of records to be blocked.

Even when all of the above conditions are met, certain operating system restrictions can cause blocking and unblocking to not take effect. In these cases, performance improvements will not be realized.

If you are using dynamically accessed indexed files, you can use READ PRIOR and READ NEXT to perform blocking. When using READ PRIOR and READ NEXT to perform blocking, you cannot change direction while there are records remaining in the block. To clear the records from a block, specify a random operation, such as a random READ or a random START, or use a sequential READ FIRST or READ LAST.

If an illegal change of direction takes place, file status 9U results. No further I/O is possible until the file is closed and reopened.

You can override blocking at run time by specifying SEQONLY(*NO) for the OVRDBF command.

For disk and database files, when you use BLOCK CONTAINS, and if the blocking factor of zero is specified or calculated, the system determines the blocking factor.

There are certain instances in which the blocking factor you specify may be changed.

Where a block of records is written or read, the I-O feedback area contains the number of records in that block. The I-O-FEEDBACK area is not updated after each read or write for files where multiple records are blocked and unblocked by ILE COBOL. It is updated when the next block is read or written.

For database files with blocking in effect, you may not see all changes as they occur, if the changes are made in different programs.

Blocking is implicitly disabled if the file has alternate record keys.

For a description of the effect of blocking on changes to database files and changing the blocking factor, see the *DB2 Universal Database for AS/400* section of the *Database and File Systems* category in the **i5/OS Information Center** at this Web site - <http://www.ibm.com/systems/i/infocenter/>.

Using File Status and Feedback Areas

To transfer data in the OPEN-FEEDBACK and I-O-FEEDBACK areas associated with an open file to an identifier, use the Format 3 ACCEPT statement. See the "ACCEPT Statement" section of the *IBM Rational Development Studio for i: ILE COBOL Reference* for more information on specifying this statement.

FILE STATUS

When the FILE STATUS clause is specified, the system moves a value into the status key data item after each input/output request that explicitly or implicitly refers to this file. This 2-character value indicates the run status of the statement. When input records are unblocked and output records are blocked, file status values that are caused by IBM i exceptions are set only when a block is processed. For more information about blocking records, refer to "Unlocking Input Records and Blocking Output Records" on page 415.

OPEN-FEEDBACK Area

The OPEN-FEEDBACK area is the part of the open data path (ODP) that contains information about the OPEN operation. This information is set during OPEN processing and is available as long as the file is open.

This area provides information about the file that the program is using. It contains:

- Information about the file that is currently open, such as file name and file type
- Information that depends on the type of file that is opened, such as printer size, screen size, diskette labels, or tape labels.

Note: OPTIONAL INPUT files that are successfully opened will not have any OPEN-FEEDBACK area information.

I-O-FEEDBACK Area

The system updates the I-O-FEEDBACK area each time a block transfers between the operating system and the program. A block can contain one or more records.

The I-O-FEEDBACK area is not updated after each read or write operation for files in which multiple records are blocked and unblocked by COBOL. If the I-O-FEEDBACK information is needed after each read or write operation in the program, the user can do either of the following:

- Prevent the compiler from generating blocking and unblocking code by not satisfying one of the conditions listed under “Unblocking Input Records and Blocking Output Records” on page 415.
- Specify SEQONLY(*NO) on the Override with database file (OVRDBF) CL command.

Preventing the compiler from generating blocking and unblocking code is more efficient than specifying SEQONLY(*NO).

Even when the compiler generates blocking and unblocking code, certain IBM i restrictions can cause blocking and unblocking to not be processed. In these cases, a performance improvement will not be realized. However, the I-O-FEEDBACK area will be updated after each read or write operation.

The I-O-FEEDBACK area contains information about the last successful I-O operation, such as: device name, device type, AID character, and error information for some devices. This area consists of a common area and a device-dependent area. The device-dependent area varies in length and content depending on the device type to which the file is associated. This area follows the I-O-FEEDBACK common area and can be obtained by specifying the receiving identifier large enough to include the common area and the appropriate device-dependent area.

For a layout and description of the data areas contained in the OPEN-FEEDBACK
and I-O-FEEDBACK areas, refer to the *DB2 Universal Database for AS/400* section of
the *Database and File Systems* category in the **i5/OS Information Center** at this Web
site - <http://www.ibm.com/systems/i/infocenter/>.

Using Commitment Control

Commitment control is a function that allows:

- Synchronization of changes to database files within the same job
- Cancellation of changes that should not be permanently entered into the database
- Locking of records being changed until changes are complete
- Techniques for recovering from job or system failure.

In some applications, it is desirable to synchronize changes to database records. If the program determines the changes are valid, the changes are then permanently made to the database (a COMMIT statement is processed). If the changes are not valid, or if a problem occurs during processing, the changes can be canceled (a ROLLBACK statement is processed). (When a file is cleared after being opened for OUTPUT, processing of a ROLLBACK does not restore cleared records to the file.) Changes made to records in a file that is *not* under commitment control are always permanent. Such changes are never affected by subsequent COMMIT or ROLLBACK statements.

Each point where a COMMIT or ROLLBACK is successfully processed is a commitment boundary. (If no COMMIT or ROLLBACK has yet been issued in a program, a commitment boundary is created by the first open of any file under commitment control.) The committing or rolling back of changes only affects changes made since the previous commitment boundary.

The synchronizing of changes at commitment boundaries makes restart or recovery procedures after a failure easier. For more information, see “Recovery After a Failure” on page 387.

When commitment control is used for database files, records in those files are subject to one of the following lock levels:

- **high lock level**

A high lock level is specified by the LCKLVL(*ALL) parameter of the Start Commitment Control (STRCMTCTL) CL command. With a high lock level (*ALL), *all* records accessed for files under commitment control, whether for input or output, are locked until a COMMIT or ROLLBACK is successfully processed.

- **cursor stability lock level**

A cursor stability lock level is specified by the LCKLVL(*CS) parameter of the Start Commitment Control (STRCMTCTL) CL command. With a cursor stability lock level (*CS), every record accessed for files opened under commitment control is locked. A record that is read, but not changed or deleted, is unlocked when a different record is read. Records that are changed, added, or deleted are locked until a COMMIT or ROLLBACK statement is successfully processed.

- **low lock level**

A low lock level is specified by the LCKLVL(*CHG) parameter of the Start Commitment Control (STRCMTCTL) CL command. With a low lock level (*CHG), every record read for update (for a file opened under commitment control) is locked. If a record is changed, added, or deleted, that record remains locked until a COMMIT or ROLLBACK statement is successfully processed. Records that are accessed for update operations but are released without being changed are unlocked.

A locked record can only be modified within the same job and through the same physical or logical file.

The lock level also governs whether locked records can be read. With a high lock level (*ALL), you cannot read locked records in a database file. With a low lock level (*CHG), you can read locked records in a database file, provided the file is opened as INPUT in your job, or opened as I-O and READ WITH NO LOCK is used.

Other jobs, where files are *not* under commitment control, can always read locked records, regardless of the lock level used, provided the files are opened as INPUT. Because it is possible in some cases for other jobs to read locked records, data can be accessed *before it is permanently committed to a database*. If a ROLLBACK statement is processed *after* another job has read locked records, the data accessed will not reflect the contents of the database.

Figure 99 on page 420 shows record locking considerations for files with and without commitment control.

			DURATION OF RECORD LOCK	
VERB	OPEN MODE	LOCK LEVEL	Next Update Operation	COMMIT or ROLLBACK
DELETE	I-O	Without Commitment Control	DELETE	
		With Commitment Control	*CHG	→
			*CS	→
*ALL	→			
READ	INPUT	Without Commitment Control	READ	
		With Commitment Control	*CHG	→
			*CS	→
*ALL	→			
READ WITH NO LOCK	I-O	Without Commitment Control	READ	
		With Commitment Control	*CHG	→
			*CS	→
*ALL	→			
READ	I-O	Without Commitment Control	READ	
		With Commitment Control	*CHG	→
			*CS	→
*ALL	→			
REWRITE	I-O	Without Commitment Control	REWRITE	
		With Commitment Control	*CHG	→
			*CS	→
*ALL	→			
START	INPUT	Without Commitment Control	START	
		With Commitment Control	*CHG	→
			*CS	→
*ALL	→			
START WITH NO LOCK	I-O	Without Commitment Control	START	
		With Commitment Control	*CHG	→
			*CS	→
*ALL	→			
START	I-O	Without Commitment Control	START	
		With Commitment Control	*CHG	→
			*CS	→
*ALL	→			
WRITE	I-O	Without Commitment Control	WRITE	
		With Commitment Control	*CHG	→
			*CS	→
*ALL	→			
WRITE	OUTPUT	Without Commitment Control	WRITE	
		With Commitment Control	*CHG	→
			*CS	→
*ALL	→			

Note: Update operations include a START, READ, REWRITE, or DELETE operation for the same file (regardless of whether it is successful or unsuccessful), and closing the file. A WRITE operation is not considered an update operation; therefore, no lock will be set or released as the result of a WRITE operation.

Figure 99. Record Locking Considerations with and without Commitment Control

A file under commitment control can be closed or opened without affecting the status of changes made since the last commitment boundary. A COMMIT must still be issued to make the changes permanent, or a ROLLBACK issued to cancel the changes. A COMMIT statement, when processed, leaves files in the same open or closed state as before processing.

If you have Version 2 Release 3 Modification 0 or earlier of the i5/OS licensed
program, all files opened under the same commitment definition within a job must
be journaled to the same journal. If you have Version 3 Release 1 or later, this
restriction no longer applies in most situations. For more information about journal
management and its related functions, and for more information about
commitment control, refer to the *Recovering your system* manual.

Commitment control must also be specified outside ILE COBOL through the i5/OS
control language (CL). The Start Commitment Control (STRCMTCTL) command
establishes the capability for commitment control and sets the level of record
locking at the high level (*ALL), the cursor stability level (*CS), or the low level
(*CHG).

When commitment control is started by using the STRCMTCTL command, the system creates a **commitment definition**. Each commitment definition is known only to the job or the activation group within the job that issued the STRCMTCTL command, depending on the commitment control scoping. The commitment definition contains information pertaining to the resources being changed under commitment control within that job or activation group within the job. The commitment control information in the commitment definition is maintained by the system as the commitment resources change.

The STRCMTCTL command does not automatically initiate commitment control for
a file. That file must also be specified in the COMMITMENT CONTROL clause of
the I-O-CONTROL paragraph within the ILE COBOL program. The commitment
control environment is normally ended by using the End Commitment Control
(ENDCMTCTL) command. This causes any uncommitted changes for database files
under commitment control to be canceled. (An implicit ROLLBACK is processed.)
For more information on the STRCMTCTL and ENDCMTCTL commands, see the
CL and APIs section of the *Programming* category in the **i5/OS Information Center**
at this Web site -<http://www.ibm.com/systems/i/infocenter/>.

For more information about commitment control, see the *Recovering your system* manual.

Note: The ability to prevent reading of uncommitted data that has been changed is a function of commitment control and is only available if you are running under commitment control. Normal (noncommitted) database support is not changed by the commitment control extension, and allows reading of locked records when a file that is opened only for input is read. Try to use files consistently. Typically, files should always be run under commitment control or never be run under commitment control.

Note: Commitment control will only be in effect when blocking is not being
performed for records in the file. You can prevent blocking at runtime by
specifying SEQONLY(*NO) on the OVRDBF command. For more
information on blocking, see "Unblocking Input Records and Blocking
Output Records" on page 415.

Commitment Control Scoping

Multiple commitment definitions can be started and used by program objects running within a job. Each commitment definition for a job identifies a separate transaction that has resources associated with it. These resources can be committed or rolled back independently of all other commitment definitions started for the job.

The **scope** for a commitment definition indicates which programs, that run within the job, use that commitment definition. Commitment definitions can be scoped in two ways:

- At the activation group level
- At the job level.

You specify the scope for a commitment definition on the CMTSCOPE parameter of the STRCMTCTL command.

The default scope for a commitment definition is to the activation group of the program issuing the STRCMTCTL command. Only program objects that run within that activation group will use that commitment definition. The commitment definition started at the activation group level for the OPM default activation group is known as the default activation group (*DFACTGRP) commitment definition. Each activation group may have its own commitment definition.

A commitment definition can also be scoped to the job. Any program object running in an activation group that does not have a commitment definition started at the activation group level uses the job level commitment definition. This occurs if the job level commitment definition has already been started by another program object for the job. Only a single job level commitment definition can be started for a job.

For a given activation group, only a single commitment definition can be used by the program objects that run within that activation group. Program objects that run within an activation group can use the commitment definition at either the job level or the activation group level. However, they cannot use both commitment definitions at the same time.

When an ILE COBOL program performs a commitment control operation, it does not directly indicate which commitment definition to use for the request. Instead, the system determines which commitment definition to use based on which activation group the requesting program object is running in.

Files associated with a commitment definition scoped to an ILE activation group will be closed and implicitly committed when the activation group ends normally. When an activation group ends abnormally, files associated with a commitment definition scoped to the activation group will be rolled back and closed.

Refer to the *ILE Concepts* book for further information about commitment control scoping.

Example of Using Commitment Control

Figure 102 on page 424 illustrates a possible usage of commitment control in a banking environment. The program processes transactions for transferring funds from one account to another. If no problems occur during the transaction, the changes are committed to the database file. If the transfer cannot take place because of improper account number or insufficient funds, a ROLLBACK is issued to cancel the changes.

```

.....+.....1.....+.....2.....+.....3.....+.....4.....+.....5.....+.....6.....+.....7.....+.....8
A* ACCOUNT MASTER PHYSICAL FILE -- ACCTMST
A
A
A          R ACCNTREC                UNIQUE
A          ACCNTKEY                5S
A          NAME                    20
A          ADDR                    20
A          CITY                    20
A          STATE                    2
A          ZIP                      5S
A          BALANCE                  10S 2
A          K ACCNTKEY

```

Figure 100. Example of Use of Commitment Control -- Account Master File DDS

```

.....+.....1.....+.....2.....+.....3.....+.....4.....+.....5.....+.....6.....+.....7.....+.....8
A* PROMPT SCREEN FILE NAME 'ACCTFMST'
A*
A          R ACCTPMT                1 INDARA
A
A          TEXT('CUSTOMER ACCOUNT PROMPT')
A
A          CA01(15 'END OF PROGRAM')
A          PUTRETAIN OVERLAY
A          1 3 'ACCOUNT MASTER UPDATE'
A          3 3 'FROM ACCOUTN NUMBER'
A          ACCTFROM                5Y 0I 3 23CHECK(ME)
A 99          ERRMSG('INVALID FROM ACCOUNT +
A          NUMBER' 99)
A 98          ERRMSG('INSUFFICIENT FUNDS IN FROM +
A          ACCOUNT' 98)
A          4 3 'TO ACCOUNT NUMBER'
A          ACCTTO                5Y 0I 4 23CHECK(ME)
A 97          ERRMSG('INVALID TO ACCOUNT +
A          NUMBER' 97)
A          5 3 'AMOUNT TRANSFERRED'
A          TRANSAMT                10Y02I 5 23
A          R ERRFMT
A 96          6 5 'INVALID FILE STATUS'
A 95          7 5 'INVALID KEY IN REWRITE'
A 94          8 5 'EOF CONDITION IN READ'

```

Figure 101. Example of Use of Commitment Control -- Prompt Screen DDS

```

Source
STMT PL SEQNBR -A 1 B.+....2...+....3...+....4...+....5...+....6...+....7..IDENTFCN S COPYNAME CHG DATE
1 000100 IDENTIFICATION DIVISION.
2 000200 PROGRAM-ID. ACCOUNT.
3 000300 ENVIRONMENT DIVISION.
4 000400 CONFIGURATION SECTION.
5 000500 SOURCE-COMPUTER. IBM-ISERIES.
6 000600 OBJECT-COMPUTER. IBM-ISERIES.
7 000700 INPUT-OUTPUT SECTION.
8 000800 FILE-CONTROL.
9 000900 SELECT ACCOUNT-FILE ASSIGN TO DATABASE-ACCTMST
11 001000 ORGANIZATION IS INDEXED
12 001100 ACCESS IS DYNAMIC
13 001200 RECORD IS EXTERNALLY-DESCRIBED-KEY
14 001300 FILE STATUS IS ACCOUNT-FILE-STATUS.
15 001400 SELECT DISPLAY-FILE ASSIGN TO WORKSTATION-ACCTFMTS-SI 1
17 001500 ORGANIZATION IS TRANSACTION.
001600*****
18 001700 I-O-CONTROL.
19 001800 COMMITMENT CONTROL FOR ACCOUNT-FILE. 2
001900*****
20 002000 DATA DIVISION.
21 002100 FILE SECTION.
22 002200 FD ACCOUNT-FILE.
23 002300 01 ACCOUNT-RECORD.
002400 COPY DDS-ALL-FORMATS OF ACCTMST.
24 +000001 05 ACCTMST-RECORD PIC X(82). <-ALL-FMTS
+000002* I-O FORMAT:ACCNTREC FROM FILE ACCTMST OF LIBRARY CBLGUIDE <-ALL-FMTS
+000003* <-ALL-FMTS
+000004*THE KEY DEFINITIONS FOR RECORD FORMAT ACCTMST <-ALL-FMTS
+000005* NUMBER NAME RETRIEVAL ALTSEQ <-ALL-FMTS
+000006* 0001 ACCNTKEY ASCENDING NO <-ALL-FMTS
25 +000007 05 ACCNTREC REDEFINES ACCTMST-RECORD. <-ALL-FMTS
26 +000008 06 ACCNTKEY PIC S9(5). <-ALL-FMTS
27 +000009 06 NAME PIC X(20). <-ALL-FMTS
28 +000010 06 ADDR PIC X(20). <-ALL-FMTS
29 +000011 06 CITY PIC X(20). <-ALL-FMTS
30 +000012 06 STATE PIC X(2). <-ALL-FMTS
31 +000013 06 ZIP PIC S9(5). <-ALL-FMTS
32 +000014 06 BALANCE PIC S9(8)V9(2). <-ALL-FMTS
002500
33 002600 FD DISPLAY-FILE.
34 002700 01 DISPLAY-REC.
002800 COPY DDS-ALL-FORMATS OF ACCTFMTS.
35 +000001 05 ACCTFMTS-RECORD PIC X(20). <-ALL-FMTS
+000002* INPUT FORMAT:ACCTPMT FROM FILE ACCTFMTS OF LIBRARY CBLGUIDE <-ALL-FMTS
+000003* CUSTOMER ACCOUNT PROMPT <-ALL-FMTS
36 +000004 05 ACCTPMT-I REDEFINES ACCTFMTS-RECORD. <-ALL-FMTS
37 +000005 06 ACCTFROM PIC S9(5). <-ALL-FMTS
38 +000006 06 ACCTTO PIC S9(5). <-ALL-FMTS
39 +000007 06 TRANSAMT PIC S9(8)V9(2). <-ALL-FMTS
+000008* OUTPUT FORMAT:ACCTPMT FROM FILE ACCTFMTS OF LIBRARY CBLGUIDE <-ALL-FMTS
+000009* CUSTOMER ACCOUNT PROMPT <-ALL-FMTS
+000010* 05 ACCTPMT-0 REDEFINES ACCTFMTS-RECORD. <-ALL-FMTS
+000011* INPUT FORMAT:ERRFMT FROM FILE ACCTFMTS OF LIBRARY CBLGUIDE <-ALL-FMTS

```

Figure 102. Example of Use of Commitment Control (Part 1 of 4)

```

5722WDS V5R4M0 060210 LN IBM ILE COBOL CBLGUIDE/ACCOUNT ISERIES1 06/02/15 13:53:23 Page 3
STMT PL SEQNBR -A 1 B.+...2...+...3...+...4...+...5...+...6...+...7..IDENTFCN S COPYNAME CHG DATE
+000012* <-ALL-FMTS
+000013* 05 ERRFMT-I REDEFINES ACCTFMTS-RECORD. <-ALL-FMTS
+000014* OUTPUT FORMAT:ERRFMT FROM FILE ACCTFMTS OF LIBRARY CBLGUIDE <-ALL-FMTS
+000015* <-ALL-FMTS
+000016* 05 ERRFMT-0 REDEFINES ACCTFMTS-RECORD. <-ALL-FMTS
002900
40 003000 WORKING-STORAGE SECTION.
41 003100 77 ACCOUNT-FILE-STATUS PIC X(2).
42 003200 77 IND-ON PIC 1 VALUE B"1".
43 003300 77 IND-OFF PIC 1 VALUE B"0".
44 003400 01 DISPFILE-INDICS.
003500 COPY DDS-ALL-FORMATS-INDIC OF ACCTFMTS. 3
45 +000001 05 ACCTFMTS-RECORD. <-ALL-FMTS
+000002* INPUT FORMAT:ACCTPMT FROM FILE ACCTFMTS OF LIBRARY CBLGUIDE <-ALL-FMTS
+000003* CUSTOMER ACCOUNT PROMPT <-ALL-FMTS
46 +000004 06 ACCTPMT-I-INDIC. <-ALL-FMTS
47 +000005 07 IN15 PIC 1 INDIC 15. <-ALL-FMTS
+000006* END OF PROGRAM <-ALL-FMTS
48 +000007 07 IN97 PIC 1 INDIC 97. <-ALL-FMTS
+000008* INVALID TO ACCOUNT NUMBER <-ALL-FMTS
49 +000009 07 IN98 PIC 1 INDIC 98. <-ALL-FMTS
+000010* INSUFFICIENT FUNDS IN FROM ACCOUNT <-ALL-FMTS
50 +000011 07 IN99 PIC 1 INDIC 99. <-ALL-FMTS
+000012* INVALID FROM ACCOUNT NUMBER <-ALL-FMTS
+000013* OUTPUT FORMAT:ACCTPMT FROM FILE ACCTFMTS OF LIBRARY CBLGUIDE <-ALL-FMTS
+000014* CUSTOMER ACCOUNT PROMPT <-ALL-FMTS
51 +000015 06 ACCTPMT-O-INDIC. <-ALL-FMTS
52 +000016 07 IN97 PIC 1 INDIC 97. <-ALL-FMTS
+000017* INVALID TO ACCOUNT NUMBER <-ALL-FMTS
53 +000018 07 IN98 PIC 1 INDIC 98. <-ALL-FMTS
+000019* INSUFFICIENT FUNDS IN FROM ACCOUNT <-ALL-FMTS
54 +000020 07 IN99 PIC 1 INDIC 99. <-ALL-FMTS
+000021* INVALID FROM ACCOUNT NUMBER <-ALL-FMTS
+000022* INPUT FORMAT:ERRFMT FROM FILE ACCTFMTS OF LIBRARY CBLGUIDE <-ALL-FMTS
+000023* <-ALL-FMTS
+000024* 06 ERRFMT-I-INDIC. <-ALL-FMTS
+000025* OUTPUT FORMAT:ERRFMT FROM FILE ACCTFMTS OF LIBRARY CBLGUIDE <-ALL-FMTS
+000026* <-ALL-FMTS
55 +000027 06 ERRFMT-O-INDIC. <-ALL-FMTS
56 +000028 07 IN94 PIC 1 INDIC 94. <-ALL-FMTS
57 +000029 07 IN95 PIC 1 INDIC 95. <-ALL-FMTS
58 +000030 07 IN96 PIC 1 INDIC 96. <-ALL-FMTS
003600
59 003700 PROCEDURE DIVISION.
60 003800 DECLARATIVES.
003900 ACCOUNT-ERR-SECTION SECTION.
004000 USE AFTER STANDARD EXCEPTION PROCEDURE ON ACCOUNT-FILE.
004100 ACCOUNT-ERR-PARAGRAPH.
61 004200 IF ACCOUNT-FILE-STATUS IS NOT EQUAL "23" THEN
62 004300 MOVE IND-ON TO IN96 OF ERRFMT-O-INDIC 4
004400 ELSE
63 004500 MOVE IND-ON TO IN95 OF ERRFMT-O-INDIC 5
004600 END-IF
64 004700 WRITE DISPLAY-REC FORMAT IS "ERRFMT"
004800 INDICATORS ARE ERRFMT-O-INDIC

```

Figure 102. Example of Use of Commitment Control (Part 2 of 4)

```

STMT PL SEQNBR -A 1 B.+...2....+...3....+...4....+...5....+...6....+...7..IDENTFCN S COPYNAME CHG DATE
004900 END-WRITE
65 005000 CLOSE DISPLAY-FILE
005100 ACCOUNT-FILE.
66 005200 STOP RUN.
005300
005400 DISPLAY-ERR-SECTION SECTION.
005500 USE AFTER STANDARD EXCEPTION PROCEDURE ON DISPLAY-FILE.
005600 DISPLAY-ERR-PARAGRAPH.
67 005700 MOVE IND-ON TO IN94 OF ERRFMT-0-INDIC
68 005800 WRITE DISPLAY-REC FORMAT IS "ERRFMT"
005900 INDICATORS ARE ERRFMT-0-INDIC
006000 END-WRITE
69 006100 CLOSE DISPLAY-FILE
006200 ACCOUNT-FILE.
70 006300 STOP RUN.
006400 END DECLARATIVES.
006500
006600 MAIN-PROGRAM SECTION.
006700 MAINLINE.
71 006800 OPEN I-O DISPLAY-FILE
006900 I-O ACCOUNT-FILE.
72 007000 MOVE ZEROS TO ACCTPMT-I-INDIC
007100 ACCTPMT-0-INDIC.
73 007200 PERFORM WRITE-READ-DISPLAY.
74 007300 PERFORM VERIFY-ACCOUNT-NO UNTIL IN15 EQUAL IND-ON.
75 007400 CLOSE DISPLAY-FILE
007500 ACCOUNT-FILE.
76 007600 STOP RUN.
007700
007800 VERIFY-ACCOUNT-NO.
77 007900 PERFORM VERIFY-TO-ACCOUNT.
78 008000 IF IN97 OF ACCTPMT-0-INDIC EQUAL IND-OFF THEN
79 008100 PERFORM VERIFY-FROM-ACCOUNT.
80 008200 PERFORM WRITE-READ-DISPLAY.
008300
008400 VERIFY-FROM-ACCOUNT.
81 008500 MOVE ACCTFROM TO ACCNTKEY.
82 008600 READ ACCOUNT-FILE
83 008700 INVALID KEY MOVE IND-ON TO IN99 OF ACCTPMT-0-INDIC
008800 END-READ
84 008900 IF IN99 OF ACCTPMT-0-INDIC EQUAL IND-ON THEN 6
009000*
85 009100 ROLLBACK
009200*
009300 ELSE
86 009400 PERFORM UPDATE-FROM-ACCOUNT
009500 END-IF.
009600
009700 VERIFY-TO-ACCOUNT.
87 009800 MOVE ACCTTO TO ACCNTKEY.
88 009900 READ ACCOUNT-FILE
89 010000 INVALID KEY MOVE IND-ON TO IN97 OF ACCTPMT-0-INDIC 7
010100 END-READ
90 010200 IF IN97 OF ACCTPMT-0-INDIC EQUAL IND-ON THEN
010300*

```

Figure 102. Example of Use of Commitment Control (Part 3 of 4)

```

5722WDS V5R4M0 060210 LN  IBM ILE COBOL          CBLGUIDE/ACCOUNT      ISERIES1  06/02/15 13:53:23      Page    5
STMT PL SEQNBR -A 1 B.+...2....+...3....+...4....+...5....+...6....+...7..IDENTFCN  S COPYNAME  CHG DATE
 91   010400      ROLLBACK  8
      010500*
      010600      ELSE
 92   010700      PERFORM UPDATE-TO-ACCOUNT
      010800      END-IF.
      010900
      011000      UPDATE-TO-ACCOUNT.
 93   011100      ADD TRANSAMT TO BALANCE.
 94   011200      REWRITE ACCOUNT-RECORD.
      011300
      011400      UPDATE-FROM-ACCOUNT.
 95   011500      SUBTRACT TRANSAMT FROM BALANCE.
 96   011600      REWRITE ACCOUNT-RECORD.
 97   011700      IF BALANCE IS LESS THAN 0 THEN
 98   011800          MOVE IND-ON TO IN98 OF ACCTPMT-0-INDIC
      011900*
 99   012000      ROLLBACK  9
      012100*
      012200      ELSE
100  012300*
      012400      COMMIT  10
      012500*
      012600      END-IF.
      012700
      012800      WRITE-READ-DISPLAY.
101  012900      WRITE DISPLAY-REC FORMAT IS "ACCTPMT"
      013000          INDICATORS ARE ACCTPMT-0-INDIC  11
      013100      END-WRITE
102  013200      MOVE ZEROS TO ACCTPMT-I-INDIC
      013300          ACCTPMT-0-INDIC.
103  013400      READ DISPLAY-FILE RECORD
      013500          INDICATORS ARE ACCTPMT-I-INDIC
      013600      END-READ.
      013700
      013800
          * * * * *  E N D   O F   S O U R C E  * * * * *

```

Figure 102. Example of Use of Commitment Control (Part 4 of 4)

- 1** A separate indicator area is provided for the program.
- 2** The COMMITMENT CONTROL clause specifies files to be placed under commitment control. Any files named in this clause are affected by the COMMIT and ROLLBACK verbs.
- 3** The Format 2 COPY statement with the indicator attribute INDIC, defines data description entries in WORKING-STORAGE for the indicators to be used in the program.
- 4** IN96 is set if there is an invalid file status.
- 5** IN95 is set if there is an INVALID KEY condition on the REWRITE operation.
- 6** IN99 is set if the entered account number is invalid for the account from which money is being transferred.
- 7** IN97 is set if the entered account number is invalid for the account to which money is being transferred.
- 8** If an INVALID KEY condition occurs on the READ, a ROLLBACK is used and the record lock placed on the record after the first READ is released.
- 9** If the transfer of funds is not allowed (an indicator has been set), the ROLLBACK statement is processed. All changes made to database files under commitment control are canceled.
- 10** If the transfer of funds was valid (no indicators have been set), the COMMIT statement is processed, and all changes made to database files under commitment control become permanent.

- 11** The INDICATORS phrase is required for options on the work station display that are controlled by indicators.

Sorting and Merging Files

Arranging records in a particular sequence is a common requirement in data processing. Such record sequencing can be accomplished using sort or merge operations.

- The **sort** operation accepts unsequenced input and produces output in a specified sequence.
- The **merge** operation compares two or more sequenced files and combines them in sequential order.

To sort or merge files, you need to do the following:

1. Describe the input and output files, if any, for sorting or merging.
 - This is accomplished by selecting the files in the FILE-CONTROL paragraph of the INPUT-OUTPUT SECTION and by describing the file using FD (File Description) entries in the FILE SECTION of the DATA DIVISION.
2. Describe the sort files and merge files.
 - This is accomplished by selecting the sort or merge files in the FILE-CONTROL paragraph of the INPUT-OUTPUT SECTION and by describing the file using SD (Sort Description) entries in the FILE SECTION of the DATA DIVISION.
3. Specify the sort or merge operation.
 - This is accomplished by performing the SORT or MERGE statements in the PROCEDURE DIVISION.

Describing the Files

Sort files and merge files must be described with SELECT statements in the Environment Division and SD (Sort Description) entries in the Data Division. For example, see Figure 103 on page 429. The sort file or merge file described in an SD entry is the working file used during the sort or merge operation. You cannot execute any input/output statements for this file.

To describe files used for input to or output from a sort or merge operation, specify FD (File Description) entries in the Data Division. You can also sort or merge records that are defined only in the Working-Storage Section or Linkage Section. If you are only sorting or merging data items from the Working-Storage Section or Linkage Section and are not using files as input to or output from a sort or merge operation, you still need SD and FILE-CONTROL entries for the sort file or merge file.

Every SD entry must contain a record description, for example:

```
SD SORT-WORK-1.  
01 SORT-WORK-1-AREA.  
   05 SORT-KEY-1      PIC X(10).  
   05 SORT-KEY-2      PIC X(10).  
   05 FILLER          PIC X(80).
```

```

5722WDS V5R4M0 060210 LN  IBM ILE COBOL          CBLGUIDE/SMPLSORT      ISERIES1  06/02/15 13:54:42      Page    2
                               S o u r c e
STMT PL SEQNBR -A 1 B.+....2....+....3....+....4....+....5....+....6....+....7..IDENTFCN  S COPYNAME  CHG DATE
 1 000100 IDENTIFICATION DIVISION.
 2 000200 PROGRAM-ID. SMPLSORT.
   000300
 3 000400 ENVIRONMENT DIVISION.
 4 000500 CONFIGURATION SECTION.
 5 000600 SOURCE-COMPUTER. IBM-ISERIES
 6 000700 OBJECT-COMPUTER. IBM-ISERIES
 7 000800 INPUT-OUTPUT SECTION.
 8 000900 FILE-CONTROL.
   001000*
   001100* Assign name for a sort file is treated as documentation.
   001200*
 9 001300 SELECT SORT-WORK-1
10 001400 ASSIGN TO DISK-SORTFILE1.
11 001500 SELECT SORT-WORK-2
12 001600 ASSIGN TO DISK-SORTFILE1.
13 001700 SELECT INPUT-FILE
14 001800 ASSIGN TO DISK-INFILE.
   001900
15 002000 DATA DIVISION.
16 002100 FILE SECTION.
17 002200 SD SORT-WORK-1.
18 002300 01 SORT-WORK-1-AREA.
19 002400 05 SORT-KEY-1 PIC X(10).
20 002500 05 SORT-KEY-2 PIC X(10).
21 002600 05 FILLER PIC X(80).
   002700
22 002800 SD SORT-WORK-2.
23 002900 01 SORT-WORK-2-AREA.
24 003000 05 SORT-KEY PIC X(5).
25 003100 05 FILLER PIC X(25).
   003200
26 003300 FD INPUT-FILE.
27 003400 01 INPUT-RECORD PIC X(100).
   003500
   003600* .
   003700* .
   003800* .
   003900
28 004000 WORKING-STORAGE SECTION.
29 004100 01 EOS-SW PIC X.
30 004200 01 FILLER.
31 004300 05 TABLE-ENTRY OCCURS 100 TIMES
   004400 INDEXED BY X1 PIC X(30).
   004500* .
   004600* .
   004700* .
   004800
                               * * * * * E N D O F S O U R C E * * * * *

```

Figure 103. Environment and Data Division Entries for a Sort Program

The sort and merge files are processed with SORT or MERGE statements in the Procedure Division. The statement specifies the key field(s) within the record upon which the sort or merge is to be sequenced. You can specify a key or keys as ascending or descending, or when you specify more than one key, as a mixture of the two.

You can mix SORT and MERGE statements in the same ILE COBOL program. An ILE COBOL program can contain any number of sort or merge operations, each with its own independent input or output procedure.

You can perform more than one sort or merge operation in your ILE COBOL program, including:

- Multiple invocations of the same sort or merge operation
- Multiple different sort or merge operations.

However, one operation must be completed before another can begin.

Sorting Files

The **sort** operation accepts unsequenced input and produces output in a specified sequence.

You can specify **input** procedures to be performed on the sort records **before** they are sorted using the SORT...INPUT PROCEDURE statement.

You can specify **output** procedures to be performed on the sort records **after** they are sorted using the SORT...OUTPUT PROCEDURE statement.

You use input or output procedures to add, delete, alter, edit, or otherwise modify the records.

You can use the SORT statement to:

- Sort data items (including tables) in the Working-Storage Section or Linkage Section
- Read records directly into the new file without any preliminary processing using the SORT...USING statement
- Transfer sorted records directly to a file without further processing using the SORT...GIVING statement.

An ILE COBOL program containing a sort operation is usually organized so that one or more input files are read and operated on by an input procedure. Within the input procedure, a RELEASE statement places a record into the sort file. If you don't want to modify or process the records before the sorting operation begins, the SORT statement USING phrase releases the unmodified records from the specified input files to the new file.

After completion of the sorting operation, sorted records can be made available, one at a time, through a RETURN statement, for modification in an output procedure. If you don't want to modify or process the sorted records, the SORT statement GIVING option names the output file and writes the sorted records to an output file.

Refer to the *IBM Rational Development Studio for i: ILE COBOL Reference* for further information on the SORT, RELEASE, and RETURN statements.

Merging Files

The **merge** operation compares two or more sequenced files and combines them in sequential order.

You have access to output procedures, used after merging, that can modify the output records using the MERGE...OUTPUT PROCEDURE statement.

Unlike the SORT statement, you cannot specify an input procedure in the MERGE statement; you must use the MERGE...USING statement.

It is not necessary to sequence input files prior to a merge operation. The merge operation sequences and combines them into one sequenced file.

When the MERGE statement is encountered in the Procedure Division, it begins the merge processing. This merge operation compares keys within the records of the input files, and passes the sequenced records, one at a time, to the RETURN statement of an output procedure or to the file named in the GIVING phrase.

If you want to process the merged records, they can be made available to your ILE COBOL program, one at a time, through a RETURN statement in an output procedure. If you don't want to modify or process the merged records, the MERGE statement GIVING phrase names the merged output file into which the merged records will be written.

Specifying the Sort Criteria

In the SORT statement, you specify the key on which the records will be sorted. The key must be defined in the record description of the records to be sorted. In the following example, notice that SORT-GRID-LOCATION and SORT-SHIFT are defined in the Data Division before they are used in the SORT statement.

```

DATA DIVISION.
:
SD SORT-FILE.
01 SORT-RECORD.
   05 SORT-KEY.
      10 SORT-SHIFT           PIC X(1).
      10 SORT-GRID-LOCATION    PIC X(2).
      10 SORT-REPORT         PIC X(3).
   05 SORT-EXT-RECORD.
      10 SORT-EXT-EMPLOYEE-NUM PIC X(6).
      10 SORT-EXT-NAME       PIC X(30).
      10 FILLER              PIC X(73).
PROCEDURE DIVISION.
:
      SORT SORT-FILE
        ON ASCENDING KEY SORT-GRID-LOCATION SORT-SHIFT
        INPUT PROCEDURE 600-SORT3-INPUT
        OUTPUT PROCEDURE 700-SORT3-OUTPUT.
:

```

To sort on more than one key, as shown in the example above, list the keys in descending order of importance. The example also shows the use of an input procedure and an output procedure. Use an input procedure if you want to process the records before you sort them, and use an output procedure if you want to further process the records after you sort them.

Restrictions on Sort Key Length

There is no maximum number of keys, as long as the total length of the keys does not exceed 2000 bytes.

Floating-Point Considerations

Key data items may be floating-point for both the SORT (and MERGE) operations. If the key is an external floating-point item, it is treated as character data, which means that the sequence in which the records are sorted depends on the collating sequence used. If the key is an internal floating-point item, the sequence will be in numeric order.

Date-Time Data Type Considerations

```
#
```

Key data items may be of class date-time for both the SORT (and MERGE) operations. In general items of class date-time are only sorted as if they were date, time, and timestamp items for those items that match i5/OS DDS date, time and timestamp formats; in all other cases they are treated as character data. Items of class date-time that are treated as character data ignore the collating sequence in

```
#
#
#
#
#
```

effect during the SORT or MERGE. For more information about using date-time
data types in ILE COBOL programs, refer to “Working with Date-Time Data
Types” on page 189.

The following is the list of DDS data types that are treated as date-time items for the purpose of sorting:

- DATE format *MDY
- DATE format *DMY
- DATE format *EUR
- DATE format *USA
- TIME format *USA.

Null-Value Considerations

Key data items may have null-values for both SORT (and MERGE) operations. In a database file, the null value occupies the highest value in the collating sequence. To be able to SORT (and MERGE) null-capable files containing null values, however, you need to first define the file as null-capable by specifying the ALWNULL keyword in the ASSIGN clause.

Alternate Collating Sequences

You can sort records on EBCDIC, ASCII, or another collating sequence. The default collating sequence is EBCDIC or the PROGRAM COLLATING SEQUENCE you specified in the Configuration Section. You can override the collating sequence named in the PROGRAM COLLATING SEQUENCE by using the COLLATING SEQUENCE phrase of the SORT statement. Consequently, you can use different collating sequences for multiple sorts in your program.

You can also specify the collating sequence that a program will use when it is run, at the time that you compile the ILE COBOL source program. You can specify the collating sequence to be used, through the SRTSEQ and LANGID parameters of the CRTCBMOD and CRTBNDCBL commands. Refer to “Specifying National Language Sort Sequence in CRTCBMOD” on page 49 for a description of how to specify the collating sequence at compile time. You can override the collating sequence specified at compile time by specifying the PROGRAM COLLATING SEQUENCE clause in the OBJECT-COMPUTER paragraph or by using the COLLATING SEQUENCE phrase of the SORT statement.

When you sort an ASCII file, you have to request the ASCII collating sequence. To do this, use the COLLATING SEQUENCE *alphabet-name* phrase of the SORT statement, where *alphabet-name* has been defined in the SPECIAL-NAMES paragraph as STANDARD-1. You can also specify this in the PROGRAM COLLATING SEQUENCE clause of the OBJECT-COMPUTER paragraph if no COLLATING SEQUENCE phrase is specified on the SORT or MERGE statement that overrides it.

Writing the Input Procedure

Use SORT...USING if you don't need to process the records in an input file(s) before they are released to the sort program. With SORT...USING *file-name*, the ILE COBOL compiler generates an input procedure to open the file, read the records, release the records to the sort program, and close the file.

The input file must not be open when the SORT statement is performed. If you want to process the records in the input file before they are released to the sort program, use the INPUT PROCEDURE option of the SORT statement.

Each input procedure must be represented as either a paragraph or a section. For example, to release records from a table in Working-Storage to the sort file, use the following:

```
PROCEDURE DIVISION.  
  
:  
    SORT SORT-FILE  
        ON ASCENDING KEY SORT-KEY  
        INPUT PROCEDURE 600-SORT3-INPUT-PROC  
  
:  
600-SORT3-INPUT-PROC SECTION.  
    PERFORM WITH TEST AFTER  
        VARYING X1 FROM 1 BY 1 UNTIL X1 = 100  
        RELEASE SORT-RECORD FROM TABLE-ENTRY(X1)  
    END-PERFORM.
```

An input procedure contains code for processing records and releasing them to the sort operation. You might want to use an input procedure to:

- Release data items to the sort file from Working-Storage
- Release records that have already been read in elsewhere in the program
- Read records from an input file, select or process them, and release them to the sort file.

To transfer records to the sort file, all input procedures must contain at least one `RELEASE` or `RELEASE FROM` statement.

Writing the Output Procedure

Use `SORT...GIVING` if you want to transfer the sorted records directly from the sort file into another file without any further processing. With `SORT...GIVING file-name`, the ILE COBOL compiler generates an output procedure to open the file, return the records, write the records, and close the file. At the time the `SORT` statement is performed, the file named with the `GIVING` phrase must not be open.

If you want to select, edit, or otherwise modify the sorted records before writing them from the sort work file into another file, use the `OUTPUT PROCEDURE` phrase of the `SORT` statement.

In the output procedure, you must use the `RETURN` statement to make each sorted record available to the output procedure. Your output procedure may then contain any statements necessary to process the records that are made available, one at a time, by the `RETURN` statement.

You can use `RETURN INTO`, instead of `RETURN`, to return and process records into Working-Storage or to an output area. You may also use the `AT END` phrase with the `RETURN` statement. The imperative statements on the `AT END` phrase are performed after all the records have been returned from the sort file.

Each output procedure must include at least one `RETURN` or `RETURN INTO` statement. Also, each output procedure must be represented as either a section or a paragraph.

Restrictions on the Input Procedures and Output Procedures

The following restrictions apply to the statements within input procedures and output procedures:

- The input procedures and output procedures must not contain any SORT or MERGE statements.
- The input procedures and output procedures must not contain any STOP RUN, EXIT PROGRAM, or GOBACK statements.
- A CALL statement to another program is permitted. The called program cannot perform a SORT or MERGE statement.
- You can use ALTER, GO TO, and PERFORM statements in the input procedures and output procedures to refer to procedure names outside the input procedure or output procedure; however, you must return to the input procedure or output procedure after a GO TO or PERFORM statement. Any COBOL procedure performed as a result of the GO TO statement or PERFORM statement must not contain any SORT or MERGE statements.
- The remainder of the Procedure Division must not contain any transfers of control to points inside the input procedure or output procedure (with the exception of the return of control from a Declarative Section).
- During a sort or merge operation, the SD data item is used. You should not use it in the output procedure before a RETURN statement is performed.

Determining Whether the Sort or Merge Was Successful

After a sort or merge operation is completed, a return code or completion code is stored in the SORT-RETURN special register. The SORT-RETURN special register contains a return code of 0 if the sort or merge operation was successful, or it contains 16 if the sort or merge operation was unsuccessful.

The contents of the SORT-RETURN special register changes after each SORT or MERGE statement is performed. You should test for successful completion after each SORT or MERGE statement. For example:

```

PROCEDURE DIVISION.
:
:   SORT SORT-WORK-2
:     ON ASCENDING KEY SORT-KEY
:     INPUT PROCEDURE 600-SORT3-INPUT-PROC
:     OUTPUT PROCEDURE 700-SORT3-OUTPUT-PROC.
:   IF SORT-RETURN NOT EQUAL TO 0
:     DISPLAY "SORT ENDED ABNORMALLY. SORT-RETURN = " SORT-RETURN
:
:   600-SORT3-INPUT-PROC SECTION.
:
:   700-SORT3-OUTPUT-PROC SECTION.
:

```

Premature Ending of a Sort or Merge Operation

You can use the SORT-RETURN special register to end a sort or merge operation before it has completed. You set the SORT-RETURN special register to 16 in an error declarative or input/output procedure to end the sort or merge operation before all of the records have been processed. The sort or merge operation ends before a record is returned or released. Control then returns to the statement following the SORT or MERGE statement.

Sorting Variable Length Records

Files with variable length records have a minimum record length and a maximum record length, rather than a single record length.

If variable length records are being sorted or merged, all of the data items referenced by key data-names must be contained within the first n character positions of the record, where n is equal to the minimum record size specified for the file.

When processing the SORT statement, the ILE COBOL compiler will issue an error message if any KEY specified in the SORT statement falls in the record length beyond the minimum record size.

Sort records will be truncated when:

- The maximum record length of the input file record is greater than the maximum record length of the sort file record
- The maximum record length of the sort file record is greater than the maximum record length of the output file record.

A compile time error message is issued when truncation will occur; a diagnostic message is issued at run time.

Sort records will be padded with blanks when:

- The minimum record length of the input file record is less than the minimum record length of the sort file record
- The minimum record length of the sort file record is less than the minimum record length of the output file record.

A compile time informational message is issued when records will be padded with blanks; no message is issued at run time.

Example of Sorting and Merging Files

Figure 104 on page 436 illustrates the creation of sorted files of current sales and year-to-date sales.

First, the SORT statement for current sales is executed. The input procedure for this sorting operation is SCREEN-DEPT. The records are sorted in ascending order of department, and within each department, in descending order of net sales. The output for this sort is then printed.

After the sorting operation is completed, the current sales records are merged with the year-to-date sales records. The records in this file are merged in ascending order of department number and, within each department, in ascending order of employee numbers, and, for each employee, in ascending order of months to create an updated year-to-date master file.

When the merging process finishes, the updated year-to-date master file is printed.

```

Source
STMT PL SEQNBR -A 1 B.+....2...+....3...+....4...+....5...+....6...+....7..IDENTFCN S COPYNAME CHG DATE
1 000100 IDENTIFICATION DIVISION.
2 000200 PROGRAM-ID. SORTMERGE.
000300*****
000400* THIS IS A SORT/MERGE EXAMPLE USING AN INPUT PROCEDURE *
000500*****
3 000600 ENVIRONMENT DIVISION.
4 000700 CONFIGURATION SECTION.
5 000800 SOURCE-COMPUTER. IBM-ISERIES
6 000900 OBJECT-COMPUTER. IBM-ISERIES
7 001000 INPUT-OUTPUT SECTION.
8 001100 FILE-CONTROL.
9 001200 SELECT WORK-FILE
10 001300 ASSIGN TO DISK-WRK.
11 001400 SELECT CURRENT-SALES-FILE-IN
12 001500 ASSIGN TO DISK-CURRIN.
13 001600 SELECT CURRENT-SALES-FILE-OUT
14 001700 ASSIGN TO DISK-CURROUT.
15 001800 SELECT YTD-SALES-FILE-IN
16 001900 ASSIGN TO DISK-YTDIN.
17 002000 SELECT YTD-SALES-FILE-OUT
18 002100 ASSIGN TO DISK-YTDOUT.
19 002200 SELECT PRINTER-OUT
20 002300 ASSIGN TO PRINTER-PRTSUMM.
002400
21 002500 DATA DIVISION.
22 002600 FILE SECTION.
23 002700 SD WORK-FILE.
24 002800 01 SALES-RECORD.
25 002900 05 EMPL-NO PIC 9(6).
26 003000 05 DEPT PIC 9(2).
27 003100 05 SALES PIC 9(7)V99.
28 003200 05 NAME-ADDR PIC X(61).
29 003300 05 MONTH PIC X(2).
30 003400 FD CURRENT-SALES-FILE-IN.
31 003500 01 CURRENT-SALES-IN.
32 003600 05 EMPL-NO PIC 9(6).
33 003700 05 DEPT PIC 9(2).
34 003800 88 ON-SITE-EMPLOYEE VALUES 0 THRU 6, 8.
35 003900 05 SALES PIC 9(7)V99.
36 004000 05 NAME-ADDR PIC X(61).
37 004100 05 MONTH PIC X(2).
38 004200 FD CURRENT-SALES-FILE-OUT.
39 004300 01 CURRENT-SALES-OUT.
40 004400 05 EMPL-NO PIC 9(6).
41 004500 05 DEPT PIC 9(2).
42 004600 05 SALES PIC 9(7)V99.
43 004700 05 NAME-ADDR PIC X(61).
44 004800 05 MONTH PIC X(2).
45 004900 FD YTD-SALES-FILE-IN.
46 005000 01 YTD-SALES-IN.
47 005100 05 EMPL-NO PIC 9(6).
48 005200 05 DEPT PIC 9(2).
49 005300 05 SALES PIC 9(7)V99.

```

Figure 104. Example of Use of SORT/MERGE (Part 1 of 3)

```

5722WDS V5R4M0 060210 LN IBM ILE COBOL CBLGUIDE/SORTMERG ISERIES1 06/02/15 13:56:03 Page 3
STMT PL SEQNBR -A 1 B.+...2...+...3...+...4...+...5...+...6...+...7..IDENTFCN S COPYNAME CHG DATE
50 005400 05 NAME-ADDR PIC X(61).
51 005500 05 MONTH PIC X(2).
52 005600 FD YTD-SALES-FILE-OUT.
53 005700 01 YTD-SALES-OUT.
54 005800 05 EMPL-NO PIC 9(6).
55 005900 05 DEPT PIC 9(2).
56 006000 05 SALES PIC 9(7)V99.
57 006100 05 NAME-ADDR PIC X(61).
58 006200 05 MONTH PIC X(2).
59 006300 FD PRINTER-OUT.
60 006400 01 PRINT-LINE.
61 006500 05 RECORD-LABEL PIC X(25).
62 006600 05 DISK-RECORD-DISPLAY PIC X(80).
006700
63 006800 WORKING-STORAGE SECTION.
64 006900 01 SALES-FILE-IN-EOF-STATUS PIC X VALUE "F".
65 007000 88 SALES-FILE-IN-END-OF-FILE VALUE "T".
66 007100 01 SALES-FILE-OUT-EOF-STATUS PIC X VALUE "F".
67 007200 88 SALES-FILE-OUT-END-OF-FILE VALUE "T".
68 007300 01 YTD-SALES-OUT-EOF-STATUS PIC X VALUE "F".
69 007400 88 YTD-SALES-OUT-END-OF-FILE VALUE "T".
007500
70 007600 PROCEDURE DIVISION.
007700 MAIN-PROGRAM SECTION.
007800 MAINLINE.
007900
71 008000 OPEN INPUT CURRENT-SALES-FILE-IN
008100 CURRENT-SALES-FILE-OUT
008200 YTD-SALES-FILE-OUT
008300 OUTPUT PRINTER-OUT.
008400*
008500* Sort current sales
008600*
72 008700 SORT WORK-FILE
008800 ON ASCENDING KEY DEPT OF SALES-RECORD
008900 ON DESCENDING KEY SALES OF SALES-RECORD
009000 INPUT PROCEDURE SCREEN-DEPT
009100 GIVING CURRENT-SALES-FILE-OUT.
73 009200 READ CURRENT-SALES-FILE-OUT
74 009300 AT END SET SALES-FILE-OUT-END-OF-FILE TO TRUE
009400 END-READ.
75 009500 PERFORM UNTIL SALES-FILE-OUT-END-OF-FILE
76 009600 MOVE "SORTED CURRENT SALES "
009700 TO RECORD-LABEL OF PRINT-LINE
77 009800 MOVE CURRENT-SALES-OUT TO DISK-RECORD-DISPLAY
78 009900 WRITE PRINT-LINE
79 010000 READ CURRENT-SALES-FILE-OUT
80 010100 AT END SET SALES-FILE-OUT-END-OF-FILE TO TRUE
010200 END-READ
010300 END-PERFORM.
010400*
010500* Update yearly report
010600*
81 010700 MERGE WORK-FILE
010800 ON ASCENDING KEY DEPT OF SALES-RECORD

```

Figure 104. Example of Use of SORT/MERGE (Part 2 of 3)

```

5722WDS V5R4M0 060210 LN  IBM ILE COBOL          CBLGUIDE/SORTMERG      ISERIES1  06/02/15 13:56:03      Page    4
STMT PL SEQNBR -A 1 B..+...2...+...3...+...4...+...5...+...6...+...7..IDENTFCN  S COPYNAME  CHG DATE
      010900      ON ASCENDING KEY EMPL-NO OF SALES-RECORD
      011000      ON ASCENDING KEY MONTH OF SALES-RECORD
      011100      USING YTD-SALES-FILE-IN
      011200      CURRENT-SALES-FILE-IN
      011300      GIVING YTD-SALES-FILE-OUT.
      011400*
      011500*      Print yearly report
      011600*
82      011700      READ YTD-SALES-FILE-OUT
83      011800      AT END SET YTD-SALES-OUT-END-OF-FILE TO TRUE
      011900      END-READ.
84      012000      PERFORM UNTIL YTD-SALES-OUT-END-OF-FILE
85      012100      MOVE "MERGED YTD SALES ",
      012200      TO RECORD-LABEL OF PRINT-LINE
86      012300      MOVE YTD-SALES-OUT TO DISK-RECORD-DISPLAY
87      012400      WRITE PRINT-LINE
88      012500      READ YTD-SALES-FILE-OUT
89      012600      AT END SET YTD-SALES-OUT-END-OF-FILE TO TRUE
      012700      END-READ
      012800      END-PERFORM.
      012900
90      013000      CLOSE CURRENT-SALES-FILE-IN
      013100      CURRENT-SALES-FILE-OUT
      013200      YTD-SALES-FILE-OUT
      013300      PRINTER-OUT.
91      013400      STOP RUN.
      013500
      013600      SCREEN-DEPT SECTION.
      013700      SCREEN-DEPT-PROCEDURE.
      013800
92      013900      READ CURRENT-SALES-FILE-IN
93      014000      AT END SET SALES-FILE-IN-END-OF-FILE TO TRUE
      014100      END-READ.
94      014200      PERFORM UNTIL SALES-FILE-IN-END-OF-FILE
95      014300      MOVE "UNSORTED CURRENT SALES ",
      014400      TO RECORD-LABEL OF PRINT-LINE
96      014500      MOVE CURRENT-SALES-IN TO DISK-RECORD-DISPLAY
97      014600      WRITE PRINT-LINE
98      014700      IF ON-SITE-EMPLOYEE
99      014800      MOVE CURRENT-SALES-IN TO SALES-RECORD
100     014900      RELEASE SALES-RECORD
      015000      END-IF
101     015100      READ CURRENT-SALES-FILE-IN
102     015200      AT END SET SALES-FILE-IN-END-OF-FILE TO TRUE
      015300      END-READ
      015400      END-PERFORM.
      015500
          * * * * *  E N D   O F   S O U R C E   * * * * *

```

Figure 104. Example of Use of SORT/MERGE (Part 3 of 3)

Declaring Data Items Using SAA Data Types

The ILE COBOL compiler allows you to convert variable-length fields from externally described files and SAA database data types to standard COBOL data items. The SAA data types you can convert are variable-length fields, date, time, timestamp fields, and DBCS-graphic and floating-point fields. ILE COBOL provides limited support for these variable-length fields.

Variable-length Fields

You can bring a variable-length field into your program if you specify *VARCHAR on the CVTOPT parameter of the CRTCBMOD or CRTBNDCBL commands, or the VARCHAR option of the PROCESS statement. When *VARCHAR is specified, your ILE COBOL program will convert a variable-length field from an externally described file into an ILE COBOL group item.

An example of such a group item is:

```

06 ITEM1.
   49 ITEM1-LENGTH    PIC S9(4) COMP-4.
   49 ITEM1-DATA      PIC X(n).

```

where n represents the maximum length of the variable-length field. Within the program, the PIC S9(4) COMP-4 is treated like any other declaration of this type, and the PIC X(n) is treated as standard alphanumeric.

When *VARCHAR is not specified, variable-length fields are ignored and declared as FILLER fields in ILE COBOL programs. If *NOVARCHAR is specified, the item is declared as follows:

```

06 FILLER    PIC x(n+2).

```

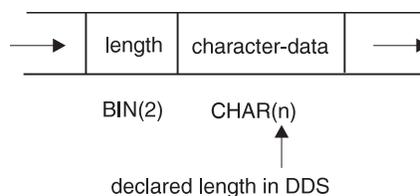
For syntax information, see the CVTOPT parameter under “CVTOPT Parameter” on page 35.

Your program can perform any valid character operations on the generated data portion; however, because of the structure of the field, the length portion must be valid binary data. This data is not valid if it is negative, or greater than the maximum field length.

If the first two bytes of the field do not contain a valid binary number, an error will occur if you try to WRITE or REWRITE a record containing the field, and file status 90 is returned.

The following conditions apply when you specify variable-length fields:

- If a variable-length field is encountered when a Format 2 COPY statement is used in the Data Division, it is declared in an ILE COBOL program as a fixed-length character field.
- For single-byte character fields, the length of the declared ILE COBOL field is the number of single-byte characters in the DDS field plus 2 bytes.
- For DBCS-graphic data fields, the length of the declared ILE COBOL field is two times the number of DBCS-graphic characters in the DDS field plus 2 bytes. For more information on graphic data types, see “DBCS-Graphic Fields” on page 451. The two extra bytes in the ILE COBOL field contain a binary number that represents the current length of the variable-length field. Figure 105 shows the ILE COBOL field length of variable-length fields.



For single-byte character fields: $2 + n = \text{ILE COBOL field length}$

For DBCS-graphic data type fields: $2 + 2(n) = \text{ILE COBOL field length}$

Figure 105. ILE COBOL Field Length of a Variable-Length Field

- Your ILE COBOL program can perform any valid character manipulation operations on the declared fixed-length field. However, because of the structure of the field, the first two bytes of the field must contain valid binary data (invalid current field-length data is less than 0, or greater than the DDS field

length). An error occurs for an input or output operation if the first two bytes of the field contain invalid field-length data; file status 90 is returned.

- If you do not specify *VARCHAR, you can encounter problems performing WRITE operations on variable-length fields, because you cannot assign a value to FILLER. The two-byte field may have a value (for example X'4040') which gives a length beyond the range allowed for the field. This causes an I/O error.
- Variable length fields can not be used in a SORT/MERGE key as a variable length field. If the variable length field is used in a SORT/MERGE key, then the entire structure is compared as an alphanumeric data item.

To see an example of a program using variable-length fields, refer to “Examples of Using Variable-length DBCS-graphic Fields” on page 452.

Date, Time, and Timestamp Fields

In ILE COBOL programs, you can use DDS date, time, and timestamp fields in two ways:

- As date, time, or timestamp data items of class date-time
- As alphanumeric fields.

Class Date-Time

A DDS date, time, and timestamp field can be declared as a FILLER item in ILE COBOL or with its DDS name depending on the *DATETIME option of the CVTOPT parameter of CRTCBMOD or CRTBNDCBL. If *NODATETIME is specified DDS date, time, and timestamp fields are declared as FILLER items in ILE COBOL. When *DATETIME is specified DDS date, time, and timestamp items are declared with their DDS names in ILE COBOL.

By default, DDS date, time, and timestamp fields create COBOL alphanumeric data items. That is, COPY DDS generates a PIC X(n) for each DDS date, time, or timestamp field. In order to generate a FORMAT clause, and thus create COBOL class date-time items, you must specify the CVTOPT values:

- *DATE for DDS date fields
- *TIME for DDS time fields
- *TIMESTAMP for DDS timestamp fields.

The equivalent PROCESS statement options for the above CVTOPT parameter values are DATE, TIME, and TIMESTAMP, respectively.

See “Working with Date-Time Data Types” on page 189 for more information of working with items of class date-time.

DDS zoned, packed, and character fields can have a DATFMT keyword. Normally, such fields will generate a PICTURE clause when a COPY DDS occurs. The resulting COBOL item will be a numeric zoned, a numeric packed, or an alphanumeric data type. However, you can use COPY DDS to generate a FORMAT clause for these items (in which case a COBOL date data item of class date-time is created). If you specify the *CVTTODATE value of the CVTOPT parameter, the DDS zoned, packed, and character fields with the DATFMT keyword will result in a date data item. The *NOCVTTODATE value of the CVTOPT parameter generates a numeric zoned, numeric packed, or alphanumeric field, respectively. These two values also exist on the PROCESS statement as CVTTODATE and NOCVTTODATE options.

Table 25 on page 441 and Table 26 on page 441 list the DATFMT parameters allowed for zoned, packed, and character DDS fields, and their equivalent ILE

COBOL format that is generated from COPY DDS when the CVTOPT(*CVTTODATE) conversion parameter is specified.

Table 25 is for character and zoned fields; USAGE DISPLAY is assumed.

Table 25. DATFMT Parameters Allowed for Character and Zoned Fields

IBM i Format	COBOL-Generated Format	Description	Format	Length
*MDY	%m%d%y	MonthDayYear	mmddy	6
*DMY	%d%m%y	DayMonthYear	ddmmy	6
*YMD	%y%m%d	YearMonthDay	yymmdd	6
*JUL	%y%j	Julian	yyddd	5
*ISO	@Y%m%d	International Standards Organization	yyyymmdd	8
*USA	%m%d@Y	IBM USA Standard	mmddy	8
*EUR	%d@m@Y	IBM European Standard	ddmmy	8
*JIS	@Y%m%d	Japanese Industrial Standard Christian Era	yyyymmdd	8
*CMDY	@C%m%d%y	CenturyMonthDayYear	cmmddy	7
*CDMY	@C%d%m%y	CenturyDayMonthYear	cddmmy	7
*CYMD	@C%y%m%d	CenturyYearMonthDay	cyymmdd	7
*MDYY	%m%d@Y	MonthDayYear	mmddy	8
*DMYY	%d@m@Y	DayMonthYear	ddmmy	8
*YYMD	@Y%m%d	YearMonthDay	yyyymmdd	8
*YM	%y%m	YearMonth	yymm	4
*MY	%m%y	MonthYear	mmy	4
*YYM	@Y%m	YearMonth	yyyymm	6
*MYY	%m@Y	MonthYear	mmy	6
*LONGJUL	@Y%j	Julian	yyddd	7

Table 26 is for packed fields; USAGE PACKED-DECIMAL is generated.

Table 26. DATFMT Parameters Allowed for Packed Fields

IBM i Format	COBOL-Generated Format	Description	Format	Length
*MDY	%m%d%y	MonthDayYear	mmddy	4
*DMY	%d%m%y	DayMonthYear	ddmmy	4
*YMD	%y%m%d	YearMonthDay	yymmdd	4
*JUL	%y%j	Julian	yyddd	3
*ISO	@Y%m%d	International Standards Organization	yyyymmdd	5
*USA	%m%d@Y	IBM USA Standard	mmddy	5
*EUR	%d@m@Y	IBM European Standard	ddmmy	5
*JIS	@Y%m%d	Japanese Industrial Standard Christian Era	yyyymmdd	5
*CMDY	@C%m%d%y	CenturyMonthDayYear	cmmddy	4
*CDMY	@C%d%m%y	CenturyDayMonthYear	cddmmy	4

Table 26. DATFMT Parameters Allowed for Packed Fields (continued)

IBM i Format	COBOL-Generated Format	Description	Format	Length
*CYMD	@C%y%m%d	CenturyYearMonthDay	cyymmdd	4
*MDYY	%m%d@Y	MonthDayYear	mmddyyy	5
*DMYY	%d%m@Y	DayMonthYear	ddmmyyy	5
*YYMD	@Y%m%d	YearMonthDay	yyyymmdd	5
*YM	%y%m	YearMonth	yymm	3
*MY	%m%y	MonthYear	mmyy	3
*YYM	@Y%m	YearMonth	yyyymm	4
*MYY	%m@Y	MonthYear	mmyyy	4
*LONGJUL	@Y%j	Julian	yyyddd	4

Class Alphanumeric

This section describes how to use date, time, and timestamp data items as alphanumeric fields in ILE COBOL programs. Contrast this with using date, time, or timestamp data items of class date-time as described in “Class Date-Time” on page 440.

By default, DDS date, time or timestamp fields are brought into an ILE COBOL program as fixed-length character fields. Your ILE COBOL program can perform any valid character operations on the fixed-length fields. These operations will follow the standard COBOL rules for alphanumeric data items. The *NODATE, *NOTIME, and *NOTIMESTAMP CVTOPT parameter values of the CRTCBMOD and CRTBNDCBL commands will cause COPY DDS to generate alphanumeric COBOL data items. These CVTOPT parameter values also exist on the PROCESS statement as: NODATE, NOTIME, and NOTIMESTAMP respectively.

Date, time, and timestamp fields are brought into your program only if you specify the *DATETIME option of the CVTOPT parameter of CRTCBMOD or CRTBNDCBL command, or the DATETIME option of the PROCESS statement. For a description and the syntax of the CVTOPT parameter, see “CVTOPT Parameter” on page 35. If *DATETIME is not specified, date, time, and timestamp fields are ignored and are declared as FILLER fields in your ILE COBOL program.

The date, time, and timestamp data types each have their own format.

```
# If a field containing date, time, or timestamp information is updated by your
# program, and the updated information is to be passed back to your database, the
# format of the field must be exactly the same as it was when the field was retrieved
# from the database. If you do not use the same format, an error will occur. For
# information on valid formats for each data type, see the Database and File Systems
# category in the i5/OS Information Center at this Web site -http://www.ibm.com/systems/i/infocenter/.
```

```
# To obtain information on how to enter source statements using the CL commands,
# refer to the CL and APIs section of the Programming category in the i5/OS
# Information Center at this Web site -http://www.ibm.com/systems/i/infocenter/.
```

If you try to WRITE a record before moving an appropriate value to a date, time, or timestamp field, the WRITE operation will fail, and file status 90 will be

returned. An error will also occur for a READ or START operation that tries to use a key field that is a date, time, or timestamp field, and that does not have an appropriate value.

If you declare date, time or timestamp items in your program as FILLER, do not attempt to WRITE records containing these fields, since you will not be able to set them to values that will be accepted by the system.

DDS date, time, and timestamp fields which are generated as alphanumeric data types in ILE COBOL can be specified as a SORT/MERGE key; however, they will be compared as alphanumeric data items, not as date, time, and timestamp data items.

Examples of How the *DATETIME Compiler Option Works with *DATE

Figure 106 defines the DDS date item DATEITEM. This section only describes how DDS date items are affected.

.....	1.....	2.....	3.....	4.....	5.....	6.....	7.....	8
A	R DATETIME							
A*								
A	VARITEM	100	VARLEN					
A*								
A	TIMEITEM	T	TIMFMT(*HMS)					
A	DATEITEM	L	DATFMT(*YMD)					
A	TIMESTAMP	Z						

Figure 106. DDS File Defining Date and Time Fields

The following examples show you how the combinations in which the *DATETIME option of the CVTOPT parameter can be specified with the *DATE option of the CVTOPT parameter, and how these combinations affect the way in which DATEITEM is brought into the program.

Example 1: If *NODATETIME is specified with *NODATE, DATEITEM is brought into the program as follows:

```
05 FILLER PIC X(8).
```

Example 2: If *DATETIME is specified with *NODATE, DATEITEM is brought into the program as follows:

```
05 DATEITEM PIC X(8).
```

Example 3: If *DATETIME is specified with *DATE, DATEITEM is brought into the program as follows:

```
05 DATEITEM FORMAT DATE '%y/%m/%d'.
```

Example 4: If *NODATETIME is specified with *DATE, DATEITEM is brought into the program as follows:

```
05 FILLER FORMAT DATE '%y/%m/%d'.
```

Null-Capable Fields

Null-capable fields are fields that can hold null values. The **null value** is a special value that is distinct from all non-null values, indicating the absence of any information. For example, a null value is not the same as a value of zero, all blanks, or hex zeroes. It is not equal to any value, not even to other null values.

For each field in a database record, there is a one-byte value that indicates whether or not the field is null. If the field *is null*, it contains the value 1; if the field *is not null*, it contains the value 0. This string of values is called the **null map**, and there is one null map for each record in a null-capable database file. Each record format in a null-capable database file has its own null map.

If a file is also keyed, then it contains a **null key map**. A null key map is a separate string of similarly defined values: one for each field in the key. There is one null key map for each record in a keyed null-capable database file. Each record format in a keyed null-capable database file has its own null key map.

The values in a null map can be boolean or alphanumeric, depending on how you define the null map in the WORKING-STORAGE section. If you are using an externally described file, and you specify a COPY-DDS statement WITH NULL-MAP, then one or more null maps with boolean values will be set up for you. If you specify a COPY-DDS statement WITH NULL-MAP-ALPHANUM, then one or more null maps with alphanumeric values will be set up for you. A COPY-DDS statement WITH NULL-KEY-MAP will generate one or more null key maps with boolean values. If you are using a program-described file, you can define the null map as either boolean or alphanumeric in the WORKING-STORAGE section.

NULL-MAP-ALPHANUM extends the range of values that can be received into or
sent from the null map to include values other than 0 or 1. Only a value of 1 in a
null map field indicates that the field is null. For more information on values other
than 0 or 1 that can be sent or received in the null map, refer to the *DB2 Universal*
Database for AS/400 section of the *Database and File Systems* category in the **i5/OS**
Information Center at this Web site - <http://www.ibm.com/systems/i/infocenter/>.
#

When a database record containing null-capable fields is accessed by an ILE COBOL program, the record's null key map, if one exists, and the record's null map are copied to or from the program's copy of the null map (null key map) by specifying a NULL-MAP (NULL-KEY-MAP) phrase on an I/O statement. For more information about using the NULL-MAP and NULL-KEY-MAP phrases on an I/O statement, refer to *IBM Rational Development Studio for i: ILE COBOL Reference*.

Null-capable file I/O, positioning to a record, and deleting a record in a null-capable keyed file are discussed in the following sections:

- “Using Null Maps and Null Key Maps in Input and Output Operations”
- “Positioning to a Null-Capable Record in a Database File” on page 445
- “Deleting a Null-Capable Record in a Database File” on page 445.

For more information about handling error conditions for null-capable fields, refer to “Handling Errors in Operations Using Null-Capable Fields” on page 393. For more information about defining null-capable fields, and using null-capable fields with the COPY DDS statement, refer to *IBM Rational Development Studio for i: ILE COBOL Reference*.

Using Null Maps and Null Key Maps in Input and Output Operations

Input and output operations can be done on null-capable fields using the NULL-MAP IS or NULL-KEY MAP IS phrases in these I/O statements:

- READ (Formats 1, 2 and 3)

- WRITE (Formats 1 and 2)
- REWRITE (Format 1).

These phrases work with the system's data management settings of the null map and null key maps that define the record and its key. The settings specified in these phrases can be subscripted or reference modified.

If the ALWNULL attribute has been specified on the ASSIGN clause, and on a WRITE or REWRITE statement you do not specify a NULL-MAP IS phrase, then a string of B'0's are passed. All of the fields in the record are assumed to *not* be null. If the file is an indexed file and you have specified a NULL-MAP IS phrase, then you must also specify a NULL-KEY-MAP IS phrase. You must ensure that for key fields, the values in the null key map are the same as the corresponding values in the null map.

If the ALWNULL attribute has been specified on the ASSIGN clause, and on a READ statement you do not specify a NULL-MAP IS phrase, then the null map will contain the same values that it contained before the READ. The same happens for null-capable keys, if you have not specified the NULL-KEY-MAP IS phrase. If the file is an indexed file and you have specified a NULL-MAP IS phrase, then you must also specify a NULL-KEY-MAP IS phrase.

For more information about the I/O statements that allow you to work with null-capable fields, refer to the *IBM Rational Development Studio for i: ILE COBOL Reference*.

Positioning to a Null-Capable Record in a Database File

To position to a null-capable record in a database file, use the NULL-KEY-MAP IS phrase in the START statement. The object of this phrase can be subscripted or reference modified. If one of the key fields referenced in the START statement is null-capable and the NULL-KEY-MAP IS phrase is not used, a null map with all zeroes is used instead.

For more information about using the NULL-KEY-MAP IS phrase to position to a null-capable record in a database, refer to the *IBM Rational Development Studio for i: ILE COBOL Reference*.

Deleting a Null-Capable Record in a Database File

To delete a null-capable record in a database file, use the NULL-KEY-MAP IS phrase in the DELETE statement. The object of this phrase can be subscripted or reference modified. If one of the key fields referenced in the DELETE statement is null-capable and the NULL-KEY-MAP IS phrase is not used, a null map with all zeros is used, instead.

For more information about using the NULL-KEY-MAP IS phrase to delete a null-capable record in a database, refer to the *IBM Rational Development Studio for i: ILE COBOL Reference*.

Example of Using Null Maps and Null Key Maps

```
*...+...1...+...2...+...3...+...4...+...5...+...6...+...7...+...8
A* THIS IS THE STUDENT INFORMATION FILE - NULLSTDT
A
A          R PERSON
A          FNAME          20
A          LNAME          30
A          MARK           3P          ALWNULL
```

Figure 107. Example of Use of Null Map and Null Key Map—Student Information File DDS

```
*...+...1...+...2...+...3...+...4...+...5...+...6...+...7...+...8
A* THIS IS THE CAR INFORMATION FILE - NULLCAR
A
A
A          R CARS
A          CARMODEL       25A        ALWNULL
A          YEAR           4P
A          OPTIONS        2P
A          PRICE          7P 2
A          K CARMODEL
```

Figure 108. Example of Use of Null Map and Null Key Map—Car Information File DDS

```

Source
STMT PL SEQNBR -A 1 B.+....2....+....3....+....4....+....5....+....6....+....7..IDENTFCN S COPYNAME  CHG DATE
1 000100 IDENTIFICATION DIVISION.
2 000200 PROGRAM-ID. NULLMAP.
3 000300 ENVIRONMENT DIVISION.
4 000400 CONFIGURATION SECTION.
5 000500 SOURCE-COMPUTER. IBM-ISERIES
6 000600 OBJECT-COMPUTER. IBM-ISERIES
7 000700 INPUT-OUTPUT SECTION.
8 000800 FILE-CONTROL.
9 000900 SELECT NULLSTDT
10 001000 ASSIGN TO DATABASE=NULLSTDT-ALWNULL 1
11 001100 ORGANIZATION IS SEQUENTIAL
12 001200 ACCESS IS SEQUENTIAL
13 001300 FILE STATUS IS NULLSTDT-STATUS.
14 001400 SELECT NULLCAR
15 001500 ASSIGN TO DATABASE=NULLCAR-ALWNULL
16 001600 ORGANIZATION IS INDEXED
17 001700 ACCESS IS DYNAMIC
18 001800 RECORD KEY IS EXTERNALLY-DESCRIBED-KEY
19 001900 FILE STATUS IS NULLCAR-STATUS.
20 002000 DATA DIVISION.
21 002100 FILE SECTION.
22 002200 FD NULLSTDT.
23 002300 01 NULLSTDT-REC.
002400 COPY DDS-ALL-FORMATS OF NULLSTDT.
24 +000001 05 NULLSTDT-RECORD PIC X(52). <-ALL-FMTS
+000002* I-O FORMAT:PERSON FROM FILE NULLSTDT OF LIBRARY CBLGUIDE <-ALL-FMTS
+000003* <-ALL-FMTS
25 +000004 05 PERSON REDEFINES NULLSTDT-RECORD. <-ALL-FMTS
26 +000005 06 FNAME PIC X(20). <-ALL-FMTS
27 +000006 06 LNAME PIC X(30). <-ALL-FMTS
28 +000007 06 MARK PIC S9(3) COMP-3. 2 <-ALL-FMTS
+000008* (Null-capable field) <-ALL-FMTS
29 002500 FD NULLCAR.
30 002600 01 NULLCAR-REC.
002700 COPY DDS-ALL-FORMATS OF NULLCAR.
31 +000001 05 NULLCAR-RECORD PIC X(34). <-ALL-FMTS
+000002* I-O FORMAT:CARS FROM FILE NULLCAR OF LIBRARY CBLGUIDE <-ALL-FMTS
+000003* <-ALL-FMTS
+000004*THE KEY DEFINITIONS FOR RECORD FORMAT CARS <-ALL-FMTS
+000005* NUMBER NAME RETRIEVAL ALTSEQ <-ALL-FMTS
+000006* 0001 CARMODEL ASCENDING NO <-ALL-FMTS
32 +000007 05 CARS REDEFINES NULLCAR-RECORD. <-ALL-FMTS
33 +000008 06 CARMODEL PIC X(25). <-ALL-FMTS
+000009* (Null-capable field) <-ALL-FMTS
34 +000010 06 YEAR PIC S9(4) COMP-3. <-ALL-FMTS
35 +000011 06 OPTIONS PIC S9(2) COMP-3. <-ALL-FMTS
36 +000012 06 PRICE PIC S9(5)V9(2) COMP-3. <-ALL-FMTS
002800
37 002900 WORKING-STORAGE SECTION.
38 003000 01 NULLSTDT-STATUS PIC XX VALUE " ".
39 003100 01 NULLCAR-STATUS PIC XX VALUE " ".
40 003200 01 NULLSTDT-NM.
003300 COPY DDS-ALL-FORMATS OF NULLSTDT

```

Figure 109. Example of Use of Null Map and Null Key Map (Part 1 of 4)

```

STMT PL SEQNBR -A 1 B...2...3...4...5...6...7..IDENTFCN S COPYNAME CHG DATE
003400 WITH NULL-MAP. 3
+000001* NULL MAP: PERSON FROM FILE NULLSTDT OF LIBRARY CBLGUIDE <-ALL-FMTS
+000002* <-ALL-FMTS
41 +000003 05 PERSON-NM. 4 <-ALL-FMTS
42 +000004 06 FILLER PIC X(2) VALUE ZEROS. <-ALL-FMTS
43 +000005 06 MARK-NF PIC 1 VALUE B"0". 5 <-ALL-FMTS
44 003500 01 NULLCAR-NKM.
003600 COPY DDS-ALL-FORMATS OF NULLCAR
003700 WITH NULL-KEY-MAP
003800 WITH NULL-MAP.
+000001* NULL MAP: CARS FROM FILE NULLCAR OF LIBRARY CBLGUIDE <-ALL-FMTS
+000002* <-ALL-FMTS
+000003* NULL KEY MAP: 6 <-ALL-FMTS
45 +000004 05 CARS-NKM. <-ALL-FMTS
46 +000005 06 CARMODEL-NF PIC 1 VALUE B"0". <-ALL-FMTS
47 +000006 05 CARS-NM. <-ALL-FMTS
48 +000007 06 CARMODEL-NF PIC 1 VALUE B"0". <-ALL-FMTS
49 +000008 06 FILLER PIC X(3) VALUE ZEROS. <-ALL-FMTS
003900
50 004000 PROCEDURE DIVISION.
004100 MAINLINE.
51 004200 OPEN OUTPUT NULLSTDT.
52 004300 MOVE "JOHN" TO FNAME OF PERSON.
53 004400 MOVE "SMITH" TO LNAME OF PERSON.
54 004500 MOVE B"1" TO MARK-NF OF PERSON-NM. 7
55 004600 WRITE NULLSTDT-REC
004700 NULL-MAP IS PERSON-NM.
56 004800 CLOSE NULLSTDT.
004900
57 005000 OPEN INPUT NULLSTDT.
58 005100 MOVE " " TO FNAME OF PERSON.
59 005200 MOVE " " TO LNAME OF PERSON.
60 005300 MOVE B"0" TO MARK-NF OF PERSON-NM.
61 005400 READ NULLSTDT NULL-MAP IS PERSON-NM.
62 005500 IF FNAME OF PERSON = "JOHN" AND
005600 LNAME OF PERSON = "SMITH" AND
005700 MARK-NF OF PERSON-NM = B"1" AND 8
005800 NULLSTDT-STATUS = "00"
63 005900 DISPLAY "NAME IS CORRECT"
006000 ELSE
64 006100 DISPLAY "NAME IS NOT CORRECT"
006200 END-IF.
65 006300 CLOSE NULLSTDT.
006400
66 006500 OPEN EXTEND NULLSTDT.
67 006600 MOVE "TOM" TO FNAME OF PERSON.
68 006700 MOVE "JONES" TO LNAME OF PERSON.
69 006800 MOVE B"1" TO MARK-NF OF PERSON-NM.
70 006900 WRITE NULLSTDT-REC NULL-MAP IS PERSON-NM.
71 007000 CLOSE NULLSTDT.
007100
72 007200 OPEN INPUT NULLSTDT.
73 007300 MOVE " " TO FNAME OF PERSON.
74 007400 MOVE " " TO LNAME OF PERSON.
75 007500 MOVE B"0" TO MARK-NF OF PERSON-NM.
    
```

Figure 109. Example of Use of Null Map and Null Key Map (Part 2 of 4)

```

STMT PL SEQNBR -A 1 B.+...2...+...3...+...4...+...5...+...6...+...7..IDENTFCN S COPYNAME CHG DATE
007600
76 007700 READ NULLSTDT
007800 NULL-MAP IS PERSON-NM.
77 007900 READ NULLSTDT
008000 NULL-MAP IS PERSON-NM.
78 008100 IF FNAME OF PERSON = "TOM" AND
008200 LNAME OF PERSON = "JONES" AND
008300 MARK-NF OF PERSON-NM = B"1" AND
008400 NULLSTDT-STATUS = "00"
79 008500 DISPLAY "NAME IS CORRECT"
008600 ELSE
80 008700 DISPLAY "NAME IS NOT CORRECT"
81 008800 DISPLAY "NAME IS: " FNAME " " LNAME
008900 END-IF.
82 009000 CLOSE NULLSTDT.
009100
83 009200 OPEN EXTEND NULLSTDT.
84 009300 MOVE "PETER" TO FNAME OF PERSON.
85 009400 MOVE "STONE" TO LNAME OF PERSON.
86 009500 MOVE B"1" TO MARK-NF OF PERSON-NM.
87 009600 WRITE NULLSTDT-REC
009700 NULL-MAP IS PERSON-NM.
88 009800 CLOSE NULLSTDT.
009900
89 010000 OPEN I-O NULLSTDT.
90 010100 MOVE " " TO FNAME OF PERSON.
91 010200 MOVE " " TO LNAME OF PERSON.
92 010300 MOVE B"1" TO MARK-NF OF PERSON-NM.
93 010400 READ NULLSTDT
010500 NULL-MAP IS PERSON-NM.
94 010600 READ NULLSTDT
010700 NULL-MAP IS PERSON-NM.
95 010800 READ NULLSTDT
010900 NULL-MAP IS PERSON-NM.
96 011000 MOVE "BRICK" TO LNAME OF PERSON.
97 011100 MOVE B"0" TO MARK-NF OF PERSON-NM.
98 011200 REWRITE NULLSTDT-REC NULL-MAP IS PERSON-NM.
99 011300 CLOSE NULLSTDT.
011400
100 011500 OPEN I-O NULLSTDT.
101 011600 MOVE " " TO FNAME OF PERSON.
102 011700 MOVE " " TO LNAME OF PERSON.
103 011800 MOVE B"1" TO MARK-NF OF PERSON-NM.
104 011900 READ NULLSTDT
012000 NULL-MAP IS PERSON-NM.
105 012100 READ NULLSTDT
012200 NULL-MAP IS PERSON-NM.
106 012300 READ NULLSTDT
012400 NULL-MAP IS PERSON-NM.
107 012500 IF FNAME OF PERSON = "PETER" AND
012600 LNAME OF PERSON = "BRICK" AND
012700 MARK-NF OF PERSON-NM = B"0" AND
012800 NULLSTDT-STATUS = "00"
108 012900 DISPLAY "NAME IS CORRECT"
013000 ELSE

```

Figure 109. Example of Use of Null Map and Null Key Map (Part 3 of 4)

```

5722WDS V5R4M0 060210 LN  IBM ILE COBOL                CBLGUIDE/NULLMAP          ISERIES1  06/02/15 14:20:55      Page 5
STMT PL SEQNBR -A 1 B...2...3...4...5...6...7..IDENTFCN  S COPYNAME  CHG DATE
109 013100      DISPLAY "NAME IS NOT CORRECT"
110 013200      DISPLAY "NAME IS: " FNAME " " LNAME
      013300      END-IF.
111 013400      CLOSE NULLSTDT.
      013500*-----*
      013600* WRITE records to indexed NULLCAR.          *
      013700*-----*
112 013800      OPEN OUTPUT NULLCAR.
113 013900      MOVE B"0"          TO CARMODEL-NF OF CARS-NKM.
114 014000      MOVE B"0"          TO CARMODEL-NF OF CARS-NM.
115 014100      MOVE "SUPERCAR"    TO CARMODEL    OF CARS.
116 014200      MOVE 1995          TO YEAR        OF CARS.
117 014300      MOVE 2             TO OPTIONS     OF CARS.
118 014400      MOVE 14799         TO PRICE       OF CARS.
119 014500      WRITE NULLCAR-REC NULL-KEY-MAP IS CARS-NKM
      014600          NULL-MAP IS CARS-NM.
120 014700      MOVE "FASTCAR"     TO CARMODEL    OF CARS.
121 014800      MOVE 1997          TO YEAR        OF CARS.
122 014900      MOVE 5             TO OPTIONS     OF CARS.
123 015000      MOVE 18799         TO PRICE       OF CARS.
124 015100      WRITE NULLCAR-REC NULL-KEY-MAP IS CARS-NKM
      015200          NULL-MAP IS CARS-NM. 9
125 015300      MOVE B"1"          TO CARMODEL-NF OF CARS-NKM.
126 015400      MOVE B"1"          TO CARMODEL-NF OF CARS-NM.
127 015500      MOVE 1996          TO YEAR        OF CARS.
128 015600      MOVE 5             TO OPTIONS     OF CARS.
129 015700      MOVE 16199         TO PRICE       OF CARS.
130 015800      WRITE NULLCAR-REC NULL-KEY-MAP IS CARS-NKM
      015900          NULL-MAP IS CARS-NM.
131 016000      CLOSE NULLCAR.
      016100
132 016200      OPEN I-O NULLCAR.
133 016300      MOVE B"0"          TO CARMODEL-NF OF CARS-NKM.
134 016400      MOVE B"0"          TO CARMODEL-NF OF CARS-NM.
135 016500      MOVE "SUPERCAR"    TO CARMODEL    OF CARS.
136 016600      MOVE 0             TO YEAR        OF CARS.
137 016700      MOVE 0             TO OPTIONS     OF CARS.
138 016800      MOVE 0             TO PRICE       OF CARS.
139 016900      READ NULLCAR
      017000          NULL-KEY-MAP IS CARS-NKM
      017100          NULL-MAP    IS CARS-NM.
140 017200      READ NULLCAR NEXT
      017300          NULL-KEY-MAP IS CARS-NKM
      017400          NULL-MAP    IS CARS-NM.
141 017500      IF CARMODEL-NF OF CARS-NKM    = B"1"      AND
      017600          YEAR        OF CARS      = 1996      AND
      017700          OPTIONS     OF CARS      = 5          AND
      017800          PRICE       OF CARS      = 16199     AND
      017900          NULLCAR-STATUS = "00"
142 018000      DISPLAY "CAR IS CORRECT"
      018100      ELSE
143 018200          DISPLAY "CAR IS NOT CORRECT"
144 018300          DISPLAY "CAR IS: " CARMODEL " " YEAR " " OPTIONS " " PRICE
145 018400          DISPLAY "NULLCAR-STATUS " NULLCAR-STATUS
      018500      END-IF.
146 018600      CLOSE NULLCAR.
      018700
147 018800      STOP RUN.
      * * * * * E N D   O F   S O U R C E   * * * * *

```

Figure 109. Example of Use of Null Map and Null Key Map (Part 4 of 4)

The sample program shown in Figure 109 on page 447 is an example of how to use null key maps and null maps in database files to track valid students and car models.

- 1** Defines the database file NULLSTDT as null-capable.
- 2** Defines data item MARK. The message (Null-capable field) appears below, since the field was defined as null-capable with the ALWNULL keyword in the DDS.
- 3** The null-capable DDS file NULLSTDT is brought into the program using the COPY DDS statement and the WITH NULL-MAP phrase.
- 4** The null map PERSON-NM is defined.

- 5 The data item MARK-NF is initialized to *not null* with a value of B"0". A value of B"1" in a null-capable field makes it null.
- 6 The null key map CARS-NKM is defined.
- 7 The record NULLSTDT-REC is written with the NULL-MAP IS PERSON-NM phrase, showing how the null map is used during a write operation. The NULL MAP IS phrase is also used in all of the other I/O operations.
- 8 The MARK-NF OF PERSON-NM data item is checked for a null value (B"1").
- 9 The NULLCAR-REC record is written, and both the null key map and null map need to be referenced using the NULL-KEY-MAP IS and NULL-MAP IS phrases.

DBCS-Graphic Fields

The DBCS-graphic data type is a character string in which each character is represented by 2 bytes. The DBCS-graphic data type does not contain shift-out (SO) or shift-in (SI) characters. The difference between single-byte and DBCS-graphic data is shown in the following figure:

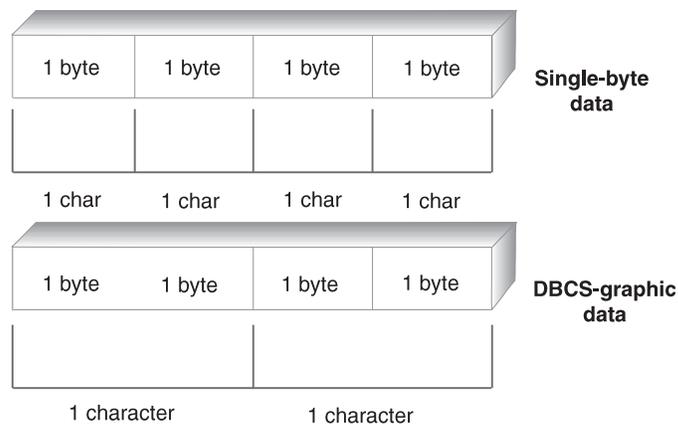


Figure 110. Comparing Single-byte and Graphic Data

DBCS-graphic data is brought into your ILE COBOL program only if you specify the *PICXGRAPHIC or *PICGGRAPHIC value on the CVTOPT parameter of the CRTCBMOD or CRTBNDCBL commands, or the CVTPICXGRAPHIC or CVTPICGGRAPHIC option of the PROCESS statement. If you do not do this, graphic data is ignored and declared as FILLER fields in your ILE COBOL program. For a description and the syntax of the CVTOPT parameter, see "Parameters of the CRTCBMOD Command" on page 28.

The following conditions apply when DBCS-graphic data is specified:

- DBCS-graphic data is copied into an ILE COBOL program as a fixed-length alphanumeric or DBCS field.
- Every DBCS-graphic data *character* has a length of 2 bytes.
- When *PICXGRAPHIC is specified, every fixed-length DBCS-graphic data *field* (for example, a field defined with PIC G(2) DISPLAY-1, as shown in Figure 110) has a length of the number of bytes in the field (a length of four bytes for the given example). When *PICGGRAPHIC is specified, every fixed-length

DBCS-graphic data field has a length of the number of double-byte characters (a length of two characters for the example).

Variable-length DBCS-graphic Fields

You can use variable-length fields in combination with DBCS-graphic data types, to specify variable-length DBCS-graphic data. To specify variable-length DBCS-graphic data, specify *VARCHAR and *PICXGRAPHIC for the CVTOPT parameter of the CRTCBMOD or CRTBNDCBL commands, or the VARCHAR and CVTPICXGRAPHIC options for the PROCESS statement.

If you specify any of the following: CVTOPT(*NOVARCHAR *NOPICGGRAPHIC), CVTOPT(*NOVARCHAR *PICGGRAPHIC), CVTOPT(*NOVARCHAR *NOPICXGRAPHIC) or CVTOPT(*NOVARCHAR *PICXGRAPHIC) and the ILE COBOL compiler encounters a variable-length DBCS-graphic data item, the resulting program contains the following:

```
          06 FILLER          PIC X(2n+2).
*              (Variable-length field)
```

where n is the number of characters in the DDS field.

If you specify CVTOPT(*VARCHAR *NOPICGGRAPHIC) or CVTOPT(*VARCHAR *NOPICXGRAPHIC), and the ILE COBOL compiler encounters a variable-length DBCS-graphic data item, the resulting program contains the following:

```
          06 NAME
*              (Variable-length field)
          49 NAME-LENGTH     PIC S9(4) COMP-4.
*              (Number of 2-byte characters)
          49 FILLER          PIC X(2n).
```

where n is the number of DBCS characters in the DDS field.

If you specify CVTOPT(*VARCHAR *PICXGRAPHIC), and the ILE COBOL compiler encounters a variable-length DBCS-graphic data item, the resulting program contains the following:

```
          06 NAME
*              (Variable-length field)
          49 NAME-LENGTH     PIC S9(4) COMP-4.
*              (Number of 2-byte characters)
          49 NAME-DATA       PIC X(2n).
```

where n is the number of DBCS characters in the DDS field.

If you specify CVTOPT(*VARCHAR *PICGGRAPHIC), and the ILE COBOL compiler encounters a variable-length DBCS-graphic data item, the resulting program contains the following:

```
          06 NAME
*              (Variable-length field)
          49 NAME-LENGTH     PIC S9(4) COMP-4.
*              (Number of 2-byte characters)
          49 NAME-DATA       PIC G(n) DISPLAY-1.
```

where n is the number of DBCS characters in the DDS field.

Examples of Using Variable-length DBCS-graphic Fields

Figure 111 on page 453 shows an example of a DDS file that defines a variable-length DBCS-graphic data item. Figure 112 on page 453 shows the ILE COBOL program using a Format 2 COPY statement with *PICXGRAPHIC and the

resulting listing when the program is compiled. Figure 113 on page 454 shows the ILE COBOL program using variable length DBCS-graphic data items with *PICGGRAPHIC.

```

.....1.....2.....3.....4.....5.....6.....7.....8
A          R SAMPLEFILE
A*
A          VARITEM      100          VARLEN
A*
A          TIMEITEM     T          TIMFMT(*HMS)
A          DATEITEM     L          DATFMT(*YMD)
A          TIMESTAMP    Z
A*
A          GRAPHITEM    100G
A          VGRAPHITEM   100G          VARLEN

```

Figure 111. DDS File Defining a Variable-Length Graphic Data Field

```

5722WDS V5R4M0 060210 LN  IBM ILE COBOL          CBLGUIDE/PGM1          ISERIES1  06/02/15 14:31:24          Page    2
                          S o u r c e
STMT PL SEQNBR -A 1 B..+....2....+....3....+....4....+....5....+....6....+....7..IDENTFCN  S COPYNAME  CHG DATE
   000100 process varchar datetime cvtpicxgraphic
   000200 Identification division.
   000300 Program-id. pgml.
   000400
   000500 Environment division.
   000600 Configuration section.
   000700 Source-computer. ibm-iSeries
   000800 Object-computer. ibm-iSeries
   000900 Input-output section.
   001000 File-control.
   001100 Select file1
   001200 assign to database-samplefi          00/08/15
   001300 organization is sequential
   001400 access is sequential
   001500 file status is fs1.
   001600
   001700 Data division.
   001800 File section.
   001900 fd file1.
   002000 01 record1.
   002100 copy dds-all-formats of samplefi.          00/08/15
+000001 05 SAMPLEFI-RECORD PIC X(546).          <-ALL-FMTS
+000002* I-O FORMAT:SAMPLEFILE FROM FILE SAMPLEFI OF LIBRARY CBLGUIDE          <-ALL-FMTS
+000003*          <-ALL-FMTS
+000004 05 SAMPLEFILE REDEFINES SAMPLEFI-RECORD.          <-ALL-FMTS
+000005 06 VARITEM.          <-ALL-FMTS
+000006* (Variable length field)          <-ALL-FMTS
+000007 49 VARITEM-LENGTH PIC S9(4) COMP-4.          <-ALL-FMTS
+000008 49 VARITEM-DATA PIC X(100).          <-ALL-FMTS
+000009 06 TIMEITEM PIC X(8).          <-ALL-FMTS
+000010* (Time field)          <-ALL-FMTS
+000011 06 DATEITEM PIC X(8).          <-ALL-FMTS
+000012* (Date field)          <-ALL-FMTS
+000013 06 TIMESTAMP PIC X(26).          <-ALL-FMTS
+000014* (Timestamp field)          <-ALL-FMTS
+000015 06 GRAPHITEM PIC X(200).          <-ALL-FMTS
+000016* (Graphic field)          <-ALL-FMTS
+000017 06 VGRAPHITEM.          <-ALL-FMTS
+000018* (Variable length field)          <-ALL-FMTS
+000019 49 VGRAPHITEM-LENGTH          <-ALL-FMTS
+000020 PIC S9(4) COMP-4.          <-ALL-FMTS
+000021* (Number of 2 byte Characters)          <-ALL-FMTS
+000022 49 VGRAPHITEM-DATA PIC X(200).          <-ALL-FMTS
+000023* (Graphic field)          <-ALL-FMTS
   002200 Working-Storage section.
   002300 77 fs1 pic x(2).
   002400
   002500 Procedure division.
   002600 Mainline.
   002700 stop run.
          * * * * * E N D   O F   S O U R C E   * * * * *

```

Figure 112. ILE COBOL Program Using Variable-Length DBCS-Graphic Data Items and *PICXGRAPHIC

```

5722WDS V5R4M0 060210 LN  IBM ILE COBOL          CBLGUIDE/DBCSPICG          ISERIES1  06/02/15 14:48:02          Page    3
                               S o u r c e
STMT PL SEQNBR -A 1 B.+....2....+....3....+....4....+....5....+....6....+....7..IDENTFCN  S COPYNAME  CHG DATE
1      000100 process varchar datetime cvtpicgraphic
2      000200 Identification division.
3      000300 Program-id. dbcspicg.
4      000400
5      000500 Environment division.
6      000600 Configuration section.
7      000700 Source-computer. ibm-iSeries
8      000800 Object-computer. ibm-iSeries
9      000900 Input-output section.
10     001000 File-control.
11     001100 Select file
12     001200 assign to database-samplefi
13     001300 organization is sequential
14     001400 access is sequential
15     001500 file status is fs1.
16     001600
17     001700 Data division.
18     001800 File section.
19     001900 fd file1.
20     002000 01 record1.
21     002100 copy dds-all-formats of samplefi.
22     +000001 05 SAMPLEFI-RECORD PIC X(546).
23     +000002* I-O FORMAT:SAMPLEFILE FROM FILE SAMPLEFI OF LIBRARY CBLGUIDE
24     +000003*
25     +000004 05 SAMPLEFILE REDEFINES SAMPLEFI-RECORD.
26     +000005 06 VARITEM.
27     +000006* (Variable length field)
28     +000007 49 VARITEM-LENGTH PIC S9(4) COMP-4.
29     +000008 49 VARITEM-DATA PIC X(100).
30     +000009 06 TIMEITEM PIC X(8).
31     +000010* (Time field)
32     +000011 06 DATEITEM PIC X(8).
33     +000012* (Date field)
34     +000013 06 TIMESTAMP PIC X(26).
35     +000014* (Timestamp field)
36     +000015 06 GRAPHITEM PIC G(100) DISPLAY-1.
37     +000016* (Graphic field)
38     +000017 06 VGRAPHITEM.
39     +000018* (Variable length field)
40     +000019 49 VGRAPHITEM-LENGTH
41     +000020 PIC S9(4) COMP-4.
42     +000021* (Number of 2 byte Characters)
43     +000022 49 VGRAPHITEM-DATA PIC G(100) DISPLAY-1.
44     +000023* (Graphic field)
45     002200 Working-Storage section.
46     002300 77 fs1 pic x(2).
47     002400
48     002500 Procedure division.
49     002600 Mainline.
50     002700 stop run.
                               * * * * * E N D O F S O U R C E * * * * *

```

Figure 113. ILE COBOL Program Using Variable-Length DBCS-Graphic Data Items and *PICGGRAPHIC

Floating-point Fields

You can bring internal floating-point fields into your program if you specify *FLOAT on the CVTOPT parameter of the CRTCBMOD or CRTBNDCBL commands, or the FLOAT option on the PROCESS statement.

When *FLOAT is specified, floating-point data types are brought into the program with their DDS names and a USAGE of COMP-1 (single-precision) or COMP-2 (double-precision). If you do not specify *FLOAT, floating-point data types are declared as FILLER fields with a USAGE of binary.

For example, if you specify *FLOAT for a single-precision floating-point field with the following DDS:

```
COMP1      9F      FLTPCN(*SINGLE)
```

the data item brought into the program is:

```
06 COMP1          COMP-1.
```

If you do not specify *FLOAT (or you specify *NOFLOAT) for the DDS specified above, the DDS field will be generated as follows:

```
06 FILLER        PIC 9(5)      COMP-4.
```

In general, floating-point data items can be used anywhere numeric decimal are used.

Chapter 19. Accessing Externally Attached Devices

This chapter describes how ILE COBOL interacts with externally attached devices. These devices are externally attached hardware such as printers, tape units, diskette units, display stations, and other systems.

You can access externally attached devices from ILE COBOL by using device files. **Device Files** are files that provide access to externally attached devices such as displays, printers, tapes, diskettes, and other systems that are attached by a communications line.

Types of Device Files

Before your ILE COBOL 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 ILE COBOL program uses the device in the way that it is described to the system. The device file formats output data from your ILE COBOL program for presentation to the device, and formats input data from the device for presentation to your ILE COBOL program.

You use the device files listed in Table 27 to access the associated externally attached devices:

Table 27. Device files and their associated externally attached devices

Device File	Associated Externally Attached Device	CL commands	ILE COBOL Device Name	ILE COBOL Default File Name
Printer Files	Provide access to printer devices and describe the format of printed output.	CRTPRTF CHGPRTF OVRPRTF	PRINTER FORMATFILE	QPRINT
Tape Files	Provide access to data files which are stored on tape devices.	CRTTAPF CHGTAPF OVRTAPF	TAPEFILE	QTAPE
Diskette Files	Provide access to data files which are stored on diskette devices.	CRTDKTF CHGDKTF OVRDKTF	DISKETTE	QDKT
Display Files	Provide access to display devices.	CRTDSPF CHGDSPF OVRDSPF	WORKSTATION	
ICF Files	Allow a program on one system to communicate with a program on another system.	CRTICFF CHGICFF OVRICFF	WORKSTATION	

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

Accessing Printer Devices

```
# You can create printed output on a printer device from an ILE COBOL program by
# issuing a WRITE statement to one or more printer files. You can use one of the
# IBM-supplied printer files, such as QPRINT, or you can create your own printer
# files using the Create Print File (CRTPRTF) command. For further information on
```

the CRTPRTF command, see the *CL and APIs* section of the *Programming* category
in the **i5/OS Information Center** at this Web site -[http://www.ibm.com/systems/](http://www.ibm.com/systems/i/infocenter/)
[i/infocenter/](http://www.ibm.com/systems/i/infocenter/).

To use a printer file in an ILE COBOL program, you must:

- Name the printer file through a file control entry in the FILE-CONTROL paragraph of the Environment Division
- Describe the printer file through a file description entry in the Data Division.

The file operations that are valid for a printer file are WRITE, OPEN, and CLOSE.

Naming Printer Files

To use a printer file in an ILE COBOL program, you must name the printer file through a file control entry in the FILE-CONTROL paragraph of the Environment Division. See the *IBM Rational Development Studio for i: ILE COBOL Reference* for a full description of the FILE-CONTROL paragraph. You can use more than one printer file in an ILE COBOL program but each printer file must have a unique name.

Printer files can be program-described files or externally-described files.

You name a program-described printer file in the FILE-CONTROL paragraph as follows:

```
FILE-CONTROL.  
    SELECT printer-file-name  
        ASSIGN TO PRINTER-printer_device_name  
        ORGANIZATION IS SEQUENTIAL.
```

You name an externally described printer file in the FILE-CONTROL paragraph as follows:

```
FILE-CONTROL.  
    SELECT printer-file-name  
        ASSIGN TO FORMATFILE-printer_device_name  
        ORGANIZATION IS SEQUENTIAL.
```

You use the SELECT clause to choose a file. This file must be identified by a FD entry in the Data Division.

You use the ASSIGN clause to associate the printer file with a printer device. You must specify a device type of PRINTER in the ASSIGN clause to use a program-described printer file. To use an externally-described printer file, you must specify a device type of FORMATFILE in the ASSIGN clause.

Use ORGANIZATION IS SEQUENTIAL in the file control entry when you name a printer file.

Describing Printer Files

Once you have named the printer file in the Environment Division, you must then describe the printer file through a file description entry in the Data Division. See the *IBM Rational Development Studio for i: ILE COBOL Reference* for a full description of the File Description Entry. Use the Format 4 File Description Entry to describe a printer file.

Printer files can be program-described or externally described. Program-described printer files are assigned to a device of PRINTER. Externally described printer files

are assigned to a device of FORMATFILE. Using FORMATFILE allows you to exploit the AS/400 function to its maximum, and using PRINTER allows for greater program portability.

The use of externally described printer files has the following advantages over program-described printer files:

- Multiple lines can be printed by one WRITE statement. When multiple lines are written by one WRITE statement and the END-OF-PAGE condition is reached, the END-OF-PAGE imperative statement is processed after all of the lines are printed. It is possible to print lines in the overflow area, and onto the next page before the END-OF-PAGE imperative statement is processed.

Figure 116 on page 463 shows an example of an occurrence of the END-OF-PAGE condition through FORMATFILE.

- Optional printing of fields based on indicator values is possible.
- Editing of field values is easily defined.
- Maintenance of print formats, especially those used by multiple programs, is easier.

Use of the ADVANCING phrase for FORMATFILE files causes a compilation error to be issued. Advancing of lines is controlled in a FORMATFILE file through DDS keywords, such as SKIPA and SKIPB, and through the use of line numbers.

Describing Program-Described Printer Files

Program described printer files must be assigned to a device of PRINTER. A simple file description entry in the Data Division that describes a program described printer file looks as follows:

```
FD print-file.  
01 print-record          PIC X(132).
```

Using the LINAGE Clause to Handle Spacing and Paging Controls: You can request all spacing and paging controls be handled internally by compiler generated code by specifying the LINAGE clause in the file description entry of a program described printer file.

```
FD print-file  
  LINAGE IS integer-1 LINES  
          WITH FOOTING AT integer-2  
          LINES AT TOP integer-3  
          LINES AT BOTTOM integer-4.  
01 print-record          PIC X(132).
```

Paper positioning is done only when the first WRITE statement is run. The paper in the printer is positioned to a new physical page, and the LINAGE-COUNTER is set to 1. When the printer file is shared and other programs have written records to the file, the ILE COBOL WRITE statement is still considered to be the first WRITE statement. Paper positioning is handled by the ILE COBOL compiler even though it is not the first WRITE statement for that file.

All spacing and paging for WRITE statements is controlled internally. The physical size of the page is ignored when paper positioning is not properly defined for the ILE COBOL compiler. For a file that has a LINAGE clause and is assigned to PRINTER, paging consists of spacing to the end of the logical page (page body) and then spacing past the bottom and top margins.

Use of the LINAGE clause degrades performance. The LINAGE clause should be used only as necessary. If the physical paging is acceptable, the LINAGE clause is not necessary.

Describing Externally Described Printer Files (FORMATFILE)

Externally described printer files must be assigned to a device of FORMATFILE. The term FORMATFILE is used because the FORMAT phrase is valid in WRITE statements for the file, and the data formatting is specified in the DDS for the file. A simple file description entry in the Data Division that describes an externally described printer file looks as follows:

```
FD print-file.  
01 print-record.  
   COPY DDS-ALL-FORMATS-0 OF print-file-dds.
```

Create a DDS for the FORMATFILE file you want to use. For information on
creating a DDS, refer to the *Database and File Systems* category in the **i5/OS**
Information Center at this Web site -<http://www.ibm.com/systems/i/infocenter/>.

Once you have created the DDS for the FORMATFILE file, use the Format 2 COPY statement to describe the layout of the printer file data record. When you compile your ILE COBOL program, the Format 2 COPY will create the Data Division statements to describe the printer file. Use the DDS-ALL-FORMATS-O option of the Format 2 COPY statement to generate one storage area for all formats.

When you have specified a device of FORMATFILE, you can obtain formatting of printed output in two ways:

1. Choose the formats to print and their order by using appropriate values in the FORMAT phrases specified for WRITE statements. For example, use one format once per page to produce a heading, and use another format to produce the detail lines on the page.
2. Choose the appropriate options to be taken when each format is printed by setting indicator values and passing these indicators through the INDICATOR phrase for the WRITE statement. For example, fields may be underlined, blank lines may be produced before or after the format is printed, or the printing of certain fields may be skipped.

The LINAGE clause should not be used for files assigned to FORMATFILE. If it is, then a compile time error message is issued indicating that the LINAGE clause has been ignored.

Writing to Printer Files

Before you can write to a printer file, you must first open the file. You use the Format 1 OPEN statement to open a printer file. A printer file must be opened for OUTPUT.

```
OPEN OUTPUT printer-file-name.
```

You use the WRITE statement to send output to a printer file. Use the Format 1 WRITE statement when you are writing to a program described printer file. Use the Format 3 WRITE statement when you are writing to an externally described printer file.

When the mnemonic-name associated with the function-name CSP is specified in the ADVANCING phrase of a WRITE statement for a printer file, it has the same effect as specifying ADVANCING 0 LINES. When the mnemonic-name associated with the function-name C01 is specified in the ADVANCING phrase of a WRITE statement for a printer file, it has the same effect as specifying ADVANCING PAGE.

The ADVANCING phrase cannot be specified in WRITE statements for files with an ASSIGN to device type FORMATFILE.

When you have finished using a printer file, you must close it. Use the Format 1 CLOSE statement to close the printer file. Once you close the file, it cannot be processed again until it is opened again.

CLOSE printer-file-name.

Example of Using FORMATFILE Files in an ILE COBOL Program

This program prints detailed employee records for all male employees from a personnel file. The input records are arranged in ascending order of employee number. Both the input file and output file are externally described.

```
.....1.....2.....3.....4.....5.....6.....7.....8
A* PHYSICAL FILE DDS FOR PERSONNEL FILE IN FORMATFILE EXAMPLE
A
A                                     UNIQUE
A      R PERSREC
A      EMPLNO          6S
A      NAME            30
A      ADDRESS1        35
A      ADDRESS2        20
A      BIRTHDATE        6
A      MARSTAT          1
A      SPOUSENAME       30
A      NUMCHILD         2S
A      K EMPLNO
```

Figure 114. Example of using FORMATFILE files in an ILE COBOL program -- Physical file DDS

```

.....+.....1.....+.....2.....+.....3.....+.....4.....+.....5.....+.....6.....+.....7.....+.....8
A* PRINTER FILE DDS FOR FORMATFILE EXAMPLE
A*
A          R HEADING 2                1 INDARA REF(PERSFILE)
A                                     SKIPB(1) SPACE(3) 3
A                                     15'PERSONNEL LISTING'
A                                     UNDERLINE
A                                     33'- ORDERED BY'
A          ORDERTYPE      15          46
A                                     80DATE EDTCDE(Y)
A                                     93TIME 4
A                                     115'PAGE:'
A                                     +1PAGNBR EDTCDE(3)
A*
A          R DETAIL 5                  SPACEA(3) 6
A* LINE 1
A          NAME          R            1'NAME:'
A                                     11UNDERLINE
A          EMPLNO       R            55'EMPLOYEE NUMBER:'
A                                     73
A          BIRTHDATE   R            87'DATE OF BIRTH:'
A* LINE 2
A                                     103SPACEA(1) 7
A          ADDRESS1    R            1'ADDRESS:'
A                                     11
A          MARSTAT     R            55'MARITAL STATUS:'
A                                     73
A 01 8 SPOUSENAMER      87'SPOUSE' 'S NAME:'
A 01 8
A* LINE 3
A          ADDRESS2    R            11SPACEB(1)
A                                     55'CHILDREN:'
A          NUMCHILD    R            73EDTCDE(3) 9

```

Figure 115. Example of Using FORMATFILE Files in an ILE COBOL Program -- Printer File DDS

- 1** INDARA specifies that a separate indicator area is to be used for the file.
- 2** HEADING is the format name that provides headings for each page.
- 3** SKIPB(1) and SPACEA(3) are used to:
 1. Skip to line 1 of the next page before format HEADING is printed.
 2. Leave 3 blank lines after format HEADING is printed.
- 4** DATE, TIME, and PAGNBR are used to have the current date, time and page number printed automatically when format HEADING is printed.
- 5** DETAIL is the format name used to print the detail line for each employee in the personnel file.
- 6** SPACEA(3) causes three lines to be left blank after each employee detail line.
- 7** SPACEA(1) causes a blank line to be printed after the field BIRTHDATE is printed. As a result, subsequent fields in the same format are printed on a new line.
- 8** 01 means that these fields are printed only if the ILE COBOL program turns indicator 01 on and passes it when format DETAIL is printed.
- 9** EDTCDE(3) is used to remove leading zeros when printing this numeric field.

```

Source
STMT PL SEQNBR -A 1 B.+....2....+....3....+....4....+....5....+....6....+....7..IDENTFCN S COPYNAME CHG DATE
1 000100 IDENTIFICATION DIVISION.
2 000200 PROGRAM-ID. FRMTFILE.
   000300
3 000400 ENVIRONMENT DIVISION.
4 000500 CONFIGURATION SECTION.
5 000600 SOURCE-COMPUTER. IBM-ISERIES
6 000700 OBJECT-COMPUTER. IBM-ISERIES
7 000800 INPUT-OUTPUT SECTION.
8 000900 FILE-CONTROL.
9 001000 SELECT PERSREPT ASSIGN TO FORMATFILE-PERSREPT-SI 1
11 001100 ORGANIZATION IS SEQUENTIAL.
12 001200 SELECT PERSFILE ASSIGN TO DATABASE-PERSFILE
14 001300 ORGANIZATION IS INDEXED
15 001400 ACCESS MODE IS SEQUENTIAL
16 001500 RECORD IS EXTERNALLY-DESCRIBED-KEY.
   001600
17 001700 DATA DIVISION.
18 001800 FILE SECTION.
19 001900 FD PERSREPT.
20 002000 01 PERSREPT-REC.
   002100 COPY DDS-ALL-FORMATS-0 OF PERSREPT. 2
21 +000001 05 PERSREPT-RECORD PIC X(130). <-ALL-FMTS
+000002* OUTPUT FORMAT:HEADING FROM FILE PERSREPT OF LIBRARY CBLGUIDE <-ALL-FMTS
+000003* <-ALL-FMTS
22 +000004 05 HEADING-0 REDEFINES PERSREPT-RECORD. <-ALL-FMTS
23 +000005 06 ORDERTYPE PIC X(15). <-ALL-FMTS
+000006* OUTPUT FORMAT:DETAIL FROM FILE PERSREPT OF LIBRARY CBLGUIDE <-ALL-FMTS
+000007* <-ALL-FMTS
24 +000008 05 DETAIL-0 REDEFINES PERSREPT-RECORD. 3 <-ALL-FMTS
25 +000009 06 NAME PIC X(30). <-ALL-FMTS
26 +000010 06 EMPLNO PIC S9(6). <-ALL-FMTS
27 +000011 06 BIRTHDATE PIC X(6). <-ALL-FMTS
28 +000012 06 ADDRESS1 PIC X(35). <-ALL-FMTS
29 +000013 06 MARSTAT PIC X(1). <-ALL-FMTS
30 +000014 06 SPOUSENAME PIC X(30). <-ALL-FMTS
31 +000015 06 ADDRESS2 PIC X(20). <-ALL-FMTS
32 +000016 06 NUMCHILD PIC S9(2). <-ALL-FMTS
33 002200 FD PERSFILE.
34 002300 01 PERSFILE-REC.
   002400 COPY DDS-ALL-FORMATS-0 OF PERSFILE.
35 +000001 05 PERSFILE-RECORD PIC X(130). <-ALL-FMTS
+000002* I-O FORMAT:PERSREC FROM FILE PERSFILE OF LIBRARY CBLGUIDE <-ALL-FMTS
+000003* <-ALL-FMTS
+000004*THE KEY DEFINITIONS FOR RECORD FORMAT PERSREC <-ALL-FMTS
+000005* NUMBER NAME RETRIEVAL ALTSEQ <-ALL-FMTS
+000006* 0001 EMPLNO NAME ASCENDING NO <-ALL-FMTS
36 +000007 05 PERSREC REDEFINES PERSFILE-RECORD. <-ALL-FMTS
37 +000008 06 EMPLNO PIC S9(6). <-ALL-FMTS
38 +000009 06 NAME PIC X(30). <-ALL-FMTS
39 +000010 06 ADDRESS1 PIC X(35). <-ALL-FMTS
40 +000011 06 ADDRESS2 PIC X(20). <-ALL-FMTS
41 +000012 06 BIRTHDATE PIC X(6). <-ALL-FMTS
42 +000013 06 MARSTAT PIC X(1). <-ALL-FMTS

```

Figure 116. Example of Using FORMATFILE Files in an ILE COBOL Program (Part 1 of 2)

```

5722WDS V5R4M0 060210 LN IBM CBLGUIDE/FRMTFILE ISERIES1 06/02/15 14:35:57 Page 3
STMT PL SEQNBR -A 1 B...2...+...3...+...4...+...5...+...6...+...7..IDENTFCN S COPYNAME CHG DATE
43 +000014 06 SPOUSENAME PIC X(30). <-ALL-FMTS
44 +000015 06 NUMCHILD PIC S9(2). <-ALL-FMTS
002500
45 002600 WORKING-STORAGE SECTION.
46 002700 77 HEAD-ORDER PIC X(15)
002800 VALUE "EMPLOYEE NUMBER".
47 002900 01 PERSREPT-INDICS.
003000 COPY DDS-ALL-FORMATS-0-INDIC OF PERSREPT. 4
48 +000001 05 PERSREPT-RECORD. <-ALL-FMTS
+000002* OUTPUT FORMAT:HEADING FROM FILE PERSREPT OF LIBRARY CBLGUIDE <-ALL-FMTS
+000003* <-ALL-FMTS
+000004* 06 HEADING-0-INDIC. <-ALL-FMTS
+000005* OUTPUT FORMAT:DETAIL FROM FILE PERSREPT OF LIBRARY CBLGUIDE <-ALL-FMTS
+000006* <-ALL-FMTS
49 +000007 06 DETAIL-0-INDIC. <-ALL-FMTS
50 +000008 07 IN01 PIC 1 INDIC 01. <-ALL-FMTS
003100
51 003200 77 EOF-FLAG PIC X(1)
003300 VALUE "0".
52 003400 88 NOT-END-OF-FILE VALUE "0".
53 003500 88 END-OF-FILE VALUE "1".
54 003600 77 MARRIED PIC X(1)
003700 VALUE "M".
003800
55 003900 PROCEDURE DIVISION.
004000 MAIN-PROGRAM SECTION.
004100 MAINLINE.
56 004200 OPEN INPUT PERSFILE
004300 OUTPUT PERSREPT.
57 004400 PERFORM HEADING-LINE.
58 004500 PERFORM UNTIL END-OF-FILE
59 004600 READ PERSFILE
60 004700 AT END SET END-OF-FILE TO TRUE
61 004800 NOT AT END PERFORM PRINT-RECORD 5
004900 END-READ
005000 END-PERFORM
62 005100 CLOSE PERSFILE
005200 PERSREPT.
63 005300 STOP RUN.
005400
005500 PRINT-RECORD.
64 005600 MOVE CORR PERSREC TO DETAIL-0. 6
*** CORRESPONDING items for statement 64:
*** EEMPLNO
*** NAME
*** ADDRESS1
*** ADDRESS2
*** BIRTHDATE
*** MARSTAT
*** SPOUSENAME
*** NUMCHILD
*** End of CORRESPONDING items for statement 64
65 005700 IF MARSTAT IN PERSFILE-REC IS EQUAL MARRIED THEN 7
66 005800 MOVE B"1" TO IN01 IN DETAIL-0-INDIC
005900 ELSE
67 006000 MOVE B"0" TO IN01 IN DETAIL-0-INDIC
006100 END-IF
68 006200 WRITE PERSREPT-REC FORMAT IS "DETAIL" 8
006300 INDICATORS ARE DETAIL-0-INDIC
69 006400 AT EOP PERFORM HEADING-LINE 9
006500 END-WRITE.
006600
006700 HEADING-LINE.
70 006800 MOVE HEAD-ORDER TO ORDERTYPE
71 006900 WRITE PERSREPT-REC FORMAT IS "HEADING"
007000 END-WRITE.
007100
***** END OF SOURCE *****

```

Figure 116. Example of Using FORMATFILE Files in an ILE COBOL Program (Part 2 of 2)

- 1** The externally described printer file is assigned to device FORMATFILE. SI indicates that a separate indicator area has been specified in the DDS.
- 2** The Format 2 COPY statement is used to copy the fields for the printer file into the program.

- 3** Note that, although the fields in format DETAIL will be printed on three separate lines, they are defined in one record.
- 4** The Format 2 COPY statement is used to copy the indicators used in the printer file into the program.
- 5** Paragraph PROCESS-RECORD processes PRINT-RECORD for each employee record.
- 6** All fields in the employee record are moved to the record for format DETAIL.
- 7** If the employee is married, indicator 01 is turned on; if not, the indicator is turned off, preventing the spouse's name field in DETAIL from being printed.
- 8** Format DETAIL is printed with indicator 01 passed to control printing.
- 9** If the number of lines per page has been exceeded, END-OF-PAGE occurs. The format HEADING is printed on a new page.

Accessing Files Stored on Tape Devices

You use **tape files** to read and write records on a tape device. Files stored on tape devices can be divided into the following two categories:

- **Sequential Single Volume:** A sequential file contained entirely on one volume. More than one file may be contained on this volume.
- **Sequential Multivolume:** A sequential file contained on more than one volume.

#

You can create your own tape files using the Create Tape File (CRTTAPF) command. For further information on the CRTTAPF command, see the *CL and APIs* section of the *Programming* category in the **i5/OS Information Center** at this Web site -<http://www.ibm.com/eserver/series/infocenter>. Alternately, you can use the default IBM-supplied tape file QTAPE. The tape file identifies the tape device to be used.

To use a file that is stored on a tape device, in your ILE COBOL program, you must:

- Name the file through a file control entry in the FILE-CONTROL paragraph of the Environment Division
- Describe the file through a file description entry in the Data Division.

You can only store a sequential file on a tape device because tape devices can only be accessed sequentially. Files stored on a tape device can have fixed or variable length records.

The file operations that are valid for a tape device are OPEN, CLOSE, READ, and WRITE.

Naming Files Stored on Tape Devices

To use a sequential file that is stored on a tape device, in your ILE COBOL program, you must name the file through a file control entry in the FILE-CONTROL paragraph of the Environment Division. See the *IBM Rational Development Studio for i: ILE COBOL Reference* for a full description of the FILE-CONTROL paragraph.

You name the file in the FILE-CONTROL paragraph as follows:

```

FILE-CONTROL.
  SELECT sequential-file-name
    ASSIGN TO TAPEFILE-tape_device_name
    ORGANIZATION IS SEQUENTIAL.

```

You use the SELECT clause to choose a file. This file must be identified by a FD entry in the Data Division.

You use the ASSIGN clause to associate the file with a tape device. You must specify a device type of TAPEFILE in the ASSIGN clause to use a tape file.

Use ORGANIZATION IS SEQUENTIAL in the file control entry when you name a file that you will access through a tape file.

Describing Files Stored on Tape Devices

Once you have named the sequential file in the Environment Division, you must then describe the file through a file description entry in the Data Division. See the *IBM Rational Development Studio for i: ILE COBOL Reference* for a full description of the File Description Entry. Use the Format 3 File Description Entry to describe a sequential file that is accessed through a tape file.

Tape files have no data description specifications (DDS). A sequential file that is stored on a tape device must be a program-described file. Your ILE COBOL program must describe the fields in the record format so the program can arrange the data received from or sent to the tape device in the manner specified by the tape file description.

A simple file description entry in the Data Division that describes a sequential file that is accessed through a tape file looks as follows:

```

FD sequential-file-name.
01 sequential-file-record.
   05 record-element-1 PIC ... .
   05 record-element-2 PIC ... .
   05 record-element-3 PIC ... .
.
.
.

```

Describing Tape Files with Variable Length Records

You can store files that have variable length records on a tape device. You specify the Format 3 RECORD clause with the FD entry of the file to define the maximum and minimum record lengths for the file.

A simple file description entry in the Data Division that describes a sequential file with variable length records looks as follows:

```

FILE SECTION.
FD sequential-file-name
  RECORD IS VARYING IN SIZE
    FROM integer-6 TO integer-7
    DEPENDING ON data-name-1.
01 minimum-sized-record.
   05 minimum-sized-element PIC X(integer-6).
01 maximum-sized-record.
   05 maximum-sized-element PIC X(integer-7).
:
WORKING-STORAGE SECTION.

```

```
77 data-name-1          PIC 9(5).  
⋮
```

The minimum record size of any record in the file is defined by *integer-6*. The maximum record size of any record in the file is defined by *integer-7*. Do not create records descriptions for the file which contain a record length that is less than that specified by *integer-6* nor a record length that is greater than that specified by *integer-7*. If any record descriptions break this rule, then a compile time error message is issued by the ILE COBOL compiler. The ILE COBOL compiler will then use the limits implied by the record description. The ILE COBOL compiler also issues a compile time error message when none of the record descriptions imply a record length that is as long as *integer-7*.

When a READ or WRITE statement is performed on a variable length record, the size of that record is defined by the contents of *data-name-1*.

Refer to the Format 3 RECORD clause in the *IBM Rational Development Studio for i: ILE COBOL Reference* for a further description of how variable length records are handled.

Reading and Writing Files Stored on Tape Devices

Before you can read from or write to a file that is stored on a tape device, you must first open the file. You use the Format 1 OPEN statement to open the file. To read from a file stored on a tape device, you must open it in INPUT mode. To write to a file stored on a tape device, you must open it in OUTPUT or EXTEND mode. A file stored on a tape device **cannot** be opened in I-O mode. The following are examples of the OPEN statement.

```
OPEN INPUT sequential-file-name.  
OPEN OUTPUT sequential-file-name.  
OPEN EXTEND sequential-file-name.
```

You use the Format 1 READ statement to read a record from a sequential file stored on a tape device. The READ statement makes the next logical record from the file available to your ILE COBOL program. For a sequential multivolume file, if the end of volume is recognized during processing of the READ statement and the logical end of file has not been reached, the following actions are taken in the order listed:

1. The standard ending volume label procedure is processed.
2. A volume switch occurs.
3. The standard beginning volume label procedure is run.
4. The first data record of the next volume is made available.

Your ILE COBOL program will receive no indication that the above actions have occurred during the read operation.

You use the Format 1 WRITE statement to write a record to a sequential file stored on a tape device. For a sequential multivolume file, if the end of volume is recognized during processing of the WRITE statement, the following actions are taken in the order listed:

1. The standard ending volume label procedure is run.
2. A volume switch occurs.
3. The standard beginning volume label procedure is run.
4. The data record is written on the next volume.

No indication that an end of volume condition has occurred is returned to your ILE COBOL program.

When you have finished using a file stored on a tape device, you must close it. Use the Format 1 CLOSE statement to close the file. Once you close the file, it cannot be processed any longer until it is opened again.

CLOSE sequential-file-name.

The CLOSE statement also gives you the option of rewinding and unloading the volume.

Ordinarily, when the CLOSE statement is performed on a tape file, the volume is rewound. However, if you want the current volume to be left in its present position after the file is closed, specify the NO REWIND phrase on the CLOSE statement. When NO REWIND is specified, the reel is not rewound.

For sequential multivolume tape files, the REEL/UNIT FOR REMOVAL phrase causes the current volume to be rewound and unloaded. The system is then notified that the volume has been removed.

For further details on rewinding and unloading volumes, refer to the discussion on the Format 1 CLOSE statement in the *IBM Rational Development Studio for i: ILE COBOL Reference*.

Reading and Writing Tape Files with Variable Length Records

When reading or writing variable length records to a tape file, ensure that the maximum variable length record is less than or equal to the maximum record length for the tape. The maximum record length for the tape is determined at the time that it is opened for OUTPUT. If the maximum record length on the tape is less than any of the variable length records being written to it, then these records will be truncated to the maximum record length for the tape.

You use the Format 1 READ statement to read a record from a sequential file stored on a tape device. The READ statement makes the next logical record from the file available to your ILE COBOL program.

If the READ operation is successful then *data-name-1*, if specified, will hold the number of the character positions of the record just read. If the READ operation is unsuccessful then *data-name-1* will hold the value it had before the READ operation was attempted.

When you specify the INTO phrase in the READ statement, the number of character positions in the current record that participate as the sending item in the implicit MOVE statement is determined by

- The contents of *data-name-1* if *data-name-1* is specified, or
- The number of character positions in the record just read if *data-name-1* is not specified.

When the READ statement is performed, if the number of character positions in the record that is read is less than the minimum record length specified by the record description entries for the file, the portion of the record area that is to the right of the last valid character read is filled with blanks. If the number of characters positions in the record that is read is greater than the maximum record length specified by the record description entries for the file, the record is truncated on the right to the maximum record size specified in the record description entries. A file status of 04 is returned when a record is read whose length falls outside the minimum or maximum record lengths defined in the file description entries for the file.

You use the Format 1 WRITE statement to write a variable length record to a sequential file stored on a tape device. You specify the length of the record to write in *data-name-1*. If you do not specify *data-name-1*, the length of the record to write is determined as follows:

- If the record contains an OCCURS...DEPENDING ON item, by the sum of the fixed portion and that portion of the table described by the number of occurrences at the time the WRITE statement is performed
- If the record does not contain an OCCURS...DEPENDING ON item, by the number of character positions in the record definition.

Accessing Files Stored on Diskette Devices

You use **diskette files** to read and write records on diskettes that are in the diskette device and that have been initialized in the basic, H, or I exchange format. File stored on diskette devices can be divided into the following two categories:

- **Sequential Single Volume:** A sequential file contained entirely on one diskette. More than one file may be contained on this diskette.
- **Sequential Multivolume:** A sequential file contained on more than one diskette.

#

You can create your own diskette files using the Create Diskette File (CRTDKTF) command. For further information on the CRTDKTF command, see the *CL and APIs* section of the *Programming* category in the **i5/OS Information Center** at this Web site -<http://www.ibm.com/systems/i/infocenter/>. Alternately, you can use the default IBM-supplied diskette file QDKT. The diskette file identifies the diskette device to be used.

To use a file that is stored on a diskette device, in your ILE COBOL program, you must:

- Name the file through a file control entry in the FILE-CONTROL paragraph of the Environment Division
- Describe the file through a file description entry in the Data Division.

You can only store a sequential file on a diskette device because diskette devices can only be accessed sequentially. The file operations that are valid for a diskette device are OPEN, CLOSE, READ, and WRITE.

Naming Files Stored on Diskette Devices

To use a sequential file that is stored on a diskette device, in your ILE COBOL program, you must name the file through a file control entry in the FILE-CONTROL paragraph of the Environment Division. See *IBM Rational Development Studio for i: ILE COBOL Reference* for a full description of the FILE-CONTROL paragraph.

You name the file in the FILE-CONTROL paragraph as follows:

```
FILE-CONTROL.  
    SELECT sequential-file-name  
    ASSIGN TO DISKETTE-diskette_device_name  
    ORGANIZATION IS SEQUENTIAL.
```

You use the SELECT clause to choose a file. This file must be identified by a FD entry in the Data Division.

You use the ASSIGN clause to associate the file with a diskette device. You must specify a device type of DISKETTE in the ASSIGN clause to use a diskette file.

Use ORGANIZATION IS SEQUENTIAL in the file control entry when you name a file that you will access through a diskette file.

Describing Files Stored on Diskette Devices

Once you have named the sequential file in the Environment Division, you must then describe the file through a file description entry in the Data Division. See *IBM Rational Development Studio for i: ILE COBOL Reference* for a full description of the File Description Entry. Use the Format 2 File Description Entry to describe a sequential file that is accessed through a diskette file.

Diskette files have no data description specifications (DDS). A sequential file that is stored on a diskette device must be a program-described file. Your ILE COBOL program must describe the fields in the record format so the program can arrange the data received from or sent to the diskette device in the manner specified by the diskette file description.

A simple file description entry in the Data Division that describes a sequential file that is accessed through a diskette file looks as follows:

```
FD sequential-file-name.  
01 sequential-file-record.  
   05 record-element-1 PIC ... .  
   05 record-element-2 PIC ... .  
   05 record-element-3 PIC ... .  
   .  
   .  
   .
```

Reading and Writing Files Stored on Diskette Devices

Before you can read from or write to a file that is stored on a diskette device, you must first open the file. You use the Format 1 OPEN statement to open the file. To read from a file stored on a diskette device, you must open it in INPUT mode. To write to a file stored on a diskette device, you must open it in OUTPUT or EXTEND mode. A file stored on a diskette device **cannot** be opened in I-O mode. The following are examples of the OPEN statement.

```
OPEN INPUT sequential-file-name.  
OPEN OUTPUT sequential-file-name.  
OPEN EXTEND sequential-file-name.
```

You use the Format 1 READ statement to read a record from a sequential file stored on a diskette device. The READ statement makes the next logical record from the file available to your ILE COBOL program.

When reading records from the input file, the record length you specify in your COBOL program should be the same as the record length found on the data file label of the diskette. If the record length specified in your COBOL program is not equal to the length of the records in the data file, the records are padded or truncated to the length specified in the program.

For a sequential multivolume file, if the end of volume is recognized during processing of the READ statement and the logical end of file has not been reached, the following actions are taken in the order listed:

1. The standard ending volume label procedure is processed.
2. A volume switch occurs.
3. The standard beginning volume label procedure is run.
4. The first data record of the next volume is made available.

Your ILE COBOL program will receive no indication that the above actions have occurred during the read operation.

You use the Format 1 WRITE statement to write a record to a sequential file stored on a diskette device.

When writing records to the output file, you must specify the record length in your COBOL program. When the record length specified in the program exceeds that for which the diskette is formatted, a diagnostic message is sent to your program, and the records are truncated. The maximum record lengths supported for diskette devices, by exchange type, are as follows:

Exchange Type	Maximum record length supported
Basic exchange	128 bytes
H exchange	256 bytes
I exchange	4096 bytes

For a sequential multivolume file, if the end of volume is recognized during processing of the WRITE statement, the following actions are taken in the order listed:

1. The standard ending volume label procedure is run.
2. A volume switch occurs.
3. The standard beginning volume label procedure is run.
4. The data record is written on the next volume.

No indication that an end of volume condition has occurred is returned to your COBOL program.

When you have finished using a file stored on a diskette device, you must close it. Use the Format 1 CLOSE statement to close the file. Once you close the file, it cannot be processed again until it is opened again.

CLOSE sequential-file-name.

Accessing Display Devices and ICF Files

You use display files to exchange information between your ILE COBOL 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 how that information is to be processed by the system on its way to and from the display. ILE COBOL uses TRANSACTION files to communicate interactively with a display device.

You use Intersystem Communication Function (ICF) files to allow a program on one system to communicate with a program on the same system or a remote system. ILE COBOL uses TRANSACTION files for intersystem communication.

See Chapter 21, "Using Transaction Files," on page 521 for a discussion on how to use TRANSACTION files with display devices and ICF files.

Chapter 20. Using DISK and DATABASE Files

Database files, which are associated with the ILE COBOL devices of DATABASE and DISK, can be:

- Externally described files, whose fields are described to IBM i through DDS
- Program-described files, whose fields are described in the program that uses the file.

Database files are created using the Create Physical File (CRTPF) or Create Logical
File (CRTL) CL commands. For a description of these commands, see the *CL and*
APIs section of the *Programming* category in the **i5/OS Information Center** at this
Web site -<http://www.ibm.com/systems/i/infocenter/>.

This chapter describes:

- The differences between DISK and DATABASE files
- The ways in which DISK and DATABASE files are organized
- The various methods of processing DISK and DATABASE files.

Differences between DISK and DATABASE Files

You use the device type DISK to associate a file in your ILE COBOL program with any physical database file or single format logical database file. When you choose DISK as the device type, you cannot use any ILE COBOL database extensions. The device type DISK does support dynamic file creation (except for indexed files) and variable length records.

You use the device type DATABASE to associate a file in your ILE COBOL program with any database file or DDM file. Choosing DATABASE as the device type allows you to use any ILE COBOL database extensions. These database extensions include the following:

- Commitment control
- Duplicate record keys
- Record formats
- Externally described files
- Null-capable files.

However, device type DATABASE does not support dynamic file creation or variable length records.

File Organization and i5/OS File Access Paths

There are two types of access paths for accessing records in a file:

- Keyed sequence access path
- Arrival sequence access path.

A file with a **keyed sequence access path** can be processed in ILE COBOL as a file with SEQUENTIAL, RELATIVE, or INDEXED organization.

For a keyed sequence file to be processed as a relative file in ILE COBOL, it must be a physical file, or a logical file whose members are based on one physical file member. For a keyed sequence file to be processed as a sequential file in ILE

COBOL, it must be a physical file, or a logical file that is based on one physical file member and that does not contain select/omit logic.

A file with an **arrival sequence access path** can be processed in ILE COBOL as a file with RELATIVE or SEQUENTIAL organization. The file must be a physical file or a logical file where each member of the logical file is based on only one physical file member.

When sequential access is specified for a logical file, records in the file are accessed through the default access path for the file.

File Processing Methods for DISK and DATABASE Files

DISK and DATABASE files can have the following organization:

- SEQUENTIAL
- RELATIVE
- INDEXED.

Each type of file organization uses unique file processing methods.

Processing Sequential Files

An ILE COBOL sequential file is a file in which records are processed in the order in which they were placed in the file, that is, in arrival sequence. For example, the tenth record placed in the file occupies the tenth record position and is the tenth record to be processed. To process a file as a sequential file, you must specify ORGANIZATION IS SEQUENTIAL in the SELECT clause, or omit the ORGANIZATION clause. A sequential file can only be accessed sequentially.

To write Standard COBOL programs that access a sequential file, you must create the file with certain characteristics. Table 28 lists these characteristics and what controls them.

Table 28. Characteristics of Sequential Files that are Accessible to Standard COBOL Programs

Characteristic	Control
The file must be a physical file.	Create the file using the CRTPF CL command.
The file cannot be a shared file.	Specify SHARE(*NO) on the CRTPF CL command.
No key can be specified for the file.	Do not include any line with K in position 17 in the Data Description Specifications (DDS) of the file.
The file must have a file type of DATA.	Specify FILETYPE(*DATA) on the CRTPF CL command.
Field editing cannot be used.	Do not specify the EDTCDE and EDTWRD keywords in the file DDS.
Line and position information cannot be specified.	Leave blanks in positions 39 to 44 of all field descriptions in the file DDS.
Spacing and skipping keywords cannot be specified.	Do not specify the SPACEA, SPACEB, SKIPPA, or SKIPPB keywords in the file DDS.
Indicators cannot be used.	Leave blanks in positions 9 to 16 of all lines in the file DDS.

Table 28. Characteristics of Sequential Files that are Accessible to Standard COBOL Programs (continued)

Characteristic	Control
System-supplied functions such as date, time, and page number cannot be used.	Do not specify the DATE, TIME, or PAGNBR keywords in the file DDS.
Select/omit level keywords cannot be used for the file.	Do not include any line with S or O in position 17 in the file DDS. Do not specify the COMP, RANGE, VALUES, or ALL keywords.
Records in the file cannot be reused.	Specify REUSEDLT(*NO) on the CRTPF CL command.
Records in the file cannot contain NULL fields	Do not specify the ALWNULL keyword in the file DDS.

The OPEN, READ, WRITE, REWRITE, and CLOSE statements are used to access data that is stored in a sequential file. Refer to the *IBM Rational Development Studio for i: ILE COBOL Reference* for a description of each of these statements.

All physical database files with SEQUENTIAL organization, that are opened for OUTPUT are cleared.

To preserve the sequence of records in a file that you open in I-O (update) mode, do not create or change the file so that you can reuse the records in it. That is, do not use a Change Physical File (CHGPF) CL command bearing the REUSEDLT option.

Note: The ILE COBOL compiler does not check that the device associated with the external file is of the type specified in the device portion of assignment-name. The device specified in the assignment-name must match the actual device to which the file is assigned. See the "ASSIGN Clause" section of the *IBM Rational Development Studio for i: ILE COBOL Reference* for more information.

Processing Relative Files

An ILE COBOL relative file is a file to be processed by a relative record number. To process a file by relative record number, you must specify ORGANIZATION IS RELATIVE in the SELECT statement for the file. A relative file can be accessed sequentially, randomly by record number, or dynamically. An ILE COBOL relative file cannot have a keyed access path.

To write Standard COBOL programs that access a relative file, you must create the file with certain characteristics. Table 29 lists these characteristics and what controls them.

Table 29. Characteristics of Relative Files that are Accessible to Standard COBOL Programs

Characteristic	Control
The file must be a physical file. ¹	Create the file using the CRTPF CL command.
The file cannot be a shared file.	Specify SHARE(*NO) on the CRTPF CL command.

Table 29. Characteristics of Relative Files that are Accessible to Standard COBOL Programs (continued)

Characteristic	Control
No key can be specified for the file.	Do not include any line with K in position 17 in the Data Description Specifications (DDS) of the file.
A starting position for retrieving records cannot be specified.	Do not issue the OVRDBF CL command with the POSITION parameter.
Select/omit level keywords cannot be used for the file.	Do not include any line with S or O in position 17 in the file DDS. Do not specify the COMP, RANGE, VALUES, or ALL keywords.
Records in the file cannot be reused.	Specify REUSEDLT(*NO) on the CRTPF CL command.
Records in the file cannot contain NULL fields.	Do not specify the ALWNULL keyword in the file DDS.
<p>Note:</p> <p>¹ A logical file whose members are based on one physical file can be used as an ILE COBOL relative file.</p>	

The OPEN, READ, WRITE, START, REWRITE, DELETE, and CLOSE statements are used to access data that is stored in a relative file. Refer to the *IBM Rational Development Studio for i: ILE COBOL Reference* for a description of each of these statements. The START statement applies only to files that are opened for INPUT or I-O and are accessed sequentially or dynamically.

For relative files that are accessed sequentially, the SELECT clause KEY phrase is ignored except for the START statement. If the KEY phrase is not specified on the START statement, the RELATIVE KEY phrase in the SELECT clause is used and KEY IS EQUAL is assumed.

For relative files that are accessed randomly or dynamically, the SELECT clause RELATIVE KEY phrase is used.

The NEXT phrase can be specified only for the READ statement for a file with SEQUENTIAL or DYNAMIC access mode. If NEXT is specified, the SELECT clause KEY phrase is ignored. The RELATIVE KEY data item is updated with the relative record number for files with sequential access on READ operations.

All physical database files that are opened for OUTPUT are cleared. Database files with RELATIVE organization, and with dynamic or random access mode, are also initialized with deleted records. Lengthy delays in OPEN OUTPUT processing are normal for extremely large relative files (over 1 000 000 records) that are accessed in dynamic or random access mode because the files are being initialized with deleted records. The length of time it takes to open a file with initialization depends on the number of records in the file.

#

When the first OPEN statement for the file is not OPEN OUTPUT, relative files should be cleared and initialized with deleted records before they are used. The RECORDS parameter of the INZPFM command must specify *DLT. Overrides are applied when the clear and initialize operations are processed by ILE COBOL, but not when they are processed with CL commands. For more information, see the discussion of the CLRPFM and INZPFM commands in the *CL and APIs* section of

#

the *Programming* category in the **i5/OS Information Center** at this Web site
-<http://www.ibm.com/systems/i/infocenter/>.

New relative files opened for OUTPUT in sequential access mode are treated differently. Table 30 summarizes conditions affecting them.

Table 30. Initialization of Relative Output Files

File Access and CL Specifications	Conditions at Opening Time	Conditions at Closing Time	File Boundary
Sequential *INZDLT		Records not written are initialized. ¹	All increments.
Sequential *INZDLT *NOMAX size		CLOSE succeeds. ¹ File status is 0Q. ²	Up to boundary of records written.
Sequential *NOINZDLT			Up to boundary of records written.
Random or dynamic	Records are initialized. File is open.		All increments.
Random or dynamic *NOMAX size	OPEN fails. File status is 9Q. ³		File is empty.

Notes:

1. Lengthy delays are normal when there remains an extremely large number of records (over 1 000 000) to be initialized to deleted records when the CLOSE statement runs.
2. To extend a file boundary beyond the current number of records, but remaining within the file size, use the INZPFM command to add deleted records before processing the file. You need to do this if you receive a file status of 0Q, and you still want to add more records to the file. Any attempt to extend a relative file beyond its current size results in a boundary violation.
3. To recover from a file status of 9Q, use the CHGPF command as described in the associated run-time message text.

For an ILE COBOL file with an organization of RELATIVE, the Reorganize Physical File Member (RGZPFM) CL command can:

- Remove all deleted records from the file. Because ILE COBOL initializes all relative file records to deleted records, any record that has not been explicitly written will be removed from the file. The relative record numbers of all records after the first deleted record in the file will change.
- Change the relative record numbers if the file has a key and the arrival sequence is changed to match a key sequence (with the KEYFILE parameter).

In addition, a Change Physical File (CHGPF) CL command bearing the REUSEDLT option can change the order of retrieved or written records when the file is operated on sequentially, because it allows the reuse of deleted records.

Processing Indexed Files

An indexed file is a file whose default access path is built on key values. One way to create a keyed access path for an indexed file is by using DDS.

An indexed file is identified by the ORGANIZATION IS INDEXED clause of the SELECT statement.

The key fields identify the records in an indexed file. The user specifies the key field in the RECORD KEY clause of the SELECT statement. The RECORD KEY data item must be defined within a record description for the indexed file. If there are multiple record descriptions for the file, only one need contain the RECORD KEY data name. However, the same positions within the record description that contains the RECORD KEY data item are accessed in the other record descriptions as the KEY value for any references to the other record descriptions for that file.

Alternate keys can also be specified with the ALTERNATE RECORD KEY clause. Using alternate keys, you can access the indexed file to read records in a sequence other than the prime key sequence.

An indexed file can be accessed sequentially, randomly by key, or dynamically.

To write Standard COBOL programs that access an indexed file, you must create the file with certain characteristics. Table 31 lists these characteristics and what controls them.

Table 31. Characteristics of Indexed Files that are Accessible to Standard COBOL Programs

Characteristic	Control
The file must be a physical file.	Create the file using the CRTPF CL command.
The file cannot be a shared file.	Specify SHARE(*NO) on the CRTPF CL command.
A key must be defined for the file.	Define at least one key field in the Data Description Specifications (DDS) of the file, using a line with K in position 17.
Keys must be contiguous within the record.	Specify a single key field in the file DDS, or specify key fields that immediately follow each other in descending order of key significance.
Key fields must be alphanumeric. They cannot be numeric.	Specify A or H in position 35 when defining any field that is to be used as a DDS key field.
The value of the key used for sequencing must include all 8 bits of every byte.	Specify alphanumeric key fields.
The file cannot have records with duplicate key values.	Specify the UNIQUE keyword in the file DDS.
Keys must be in ascending sequence.	Do not specify the DESCEND keyword in the file DDS.
A starting position for retrieving records cannot be specified.	Do not issue the OVRDBF CL command with the POSITION parameter.
Select/omit level keywords cannot be used for the file.	Do not include any line with S or O in position 17 in the file DDS. Do not specify the COMP, RANGE, VALUES, or ALL keywords.
Records in the file cannot contain NULL fields.	Do not specify the ALWNULL keyword in the file DDS.

The OPEN, READ, WRITE, START, REWRITE, DELETE, and CLOSE statements are used to access data that is stored in an indexed file. Refer to the *IBM Rational Development Studio for i: ILE COBOL Reference* for a description of each of these statements. When accessing indexed files, the FORMAT phrase is optional for

DATABASE files, and not allowed for DISK files. If the FORMAT phrase is not specified, the default format name of the file is used. The default format name of the file is the first format name defined in the file. The special register, DB-FORMAT-NAME, can be used to retrieve the format name used on the last successful I/O operation.

When you read records sequentially from an indexed file, the records will be returned in arrival sequence or in keyed sequence depending on how the file is described in your ILE COBOL program. To retrieve the records in arrival sequence, use

```
ORGANIZATION IS SEQUENTIAL  
ACCESS IS SEQUENTIAL
```

with the SELECT statement for the indexed file. To retrieve the records in keyed sequence (typically in ascending order), use

```
ORGANIZATION IS INDEXED  
ACCESS IS SEQUENTIAL
```

with the SELECT statement for the indexed file.

For indexed files that are accessed sequentially, the SELECT clause KEY phrase is ignored except for the START statement. If the KEY phrase is not specified on the START statement, the RECORD KEY phrase in the SELECT clause is used and KEY IS EQUAL is assumed.

For indexed files that are accessed randomly or dynamically, the SELECT clause KEY phrase is used except for the START statement. If the KEY phrase is not specified on the START statement, the RECORD KEY phrase in the SELECT clause is used and KEY IS EQUAL is assumed.

NEXT, PRIOR, FIRST, or LAST can be specified for the READ statement for DATABASE files with DYNAMIC access. NEXT can also be specified for the READ statement for DATABASE files with SEQUENTIAL access. If NEXT, PRIOR, FIRST, or LAST is specified, the SELECT clause KEY phrase is ignored.

All physical database files with INDEXED organization that are opened for OUTPUT are cleared.

Valid RECORD KEYS

The DDS for the file specifies the fields to be used as the key field. If the file has multiple key fields, the key fields must be contiguous in each record unless RECORD KEY IS EXTERNALLY-DESCRIBED-KEY is specified.

When the DDS specifies only one key field for the file, the RECORD KEY must be a single field of the same length as the key field defined in the DDS.

If a Format 2 COPY statement is specified for the file, the RECORD KEY clause must specify one of the following:

- The name used in the DDS for the key field with -DDS added to the end, if the name is a COBOL reserved word.
- The data name defined in a program-described record description for the file, with the same length and in the same location as the key field defined in the DDS.

- **EXTERNALLY-DESCRIBED-KEY.** This keyword specifies that the keys defined in the DDS for each record format are to be used for accessing the file. These keys can be noncontiguous. They can be defined at different positions within one record format.

When the DDS specifies multiple contiguous key fields, the RECORD KEY data name must be a single field with its length equal to the sum of the lengths of the multiple key fields in the DDS. If a Format 2 COPY statement is specified for the file, there must also be a program-described record description for the file that defines the RECORD KEY data name with the proper length and at the proper position in the record.

Contiguous items are consecutive elementary or group items in the Data Division that are contained in a single data hierarchy.

Referring to a Partial Key

A START statement allows the use of a partial key. The KEY IS phrase is required.

Refer to the “START Statement” in the *IBM Rational Development Studio for i: ILE COBOL Reference* for information about the rules for specifying a search argument that refers to a partial key.

Figure 117 on page 481 shows an example of START statements using a program-described file.

```

Source
STMT PL SEQNBR -A 1 B.+....2....+....3....+....4....+....5....+....6....+....7..IDENTFCN S COPYNAME CHG DATE
1 000100 IDENTIFICATION DIVISION.
2 000200 PROGRAM-ID. STRTPGMD.
000300
3 000400 ENVIRONMENT DIVISION.
4 000500 CONFIGURATION SECTION.
5 000600 SOURCE-COMPUTER. IBM-ISERIES.
6 000700 OBJECT-COMPUTER. IBM-ISERIES.
7 000800 INPUT-OUTPUT SECTION.
8 000900 FILE-CONTROL.
9 001000 SELECT FILE-1 ASSIGN TO DISK-NAMES 00/08/15
11 001100 ACCESS IS DYNAMIC RECORD KEY IS FULL-NAME IN FILE-1
13 001200 ORGANIZATION IS INDEXED.
001300
14 001400 DATA DIVISION.
15 001500 FILE SECTION.
16 001600 FD FILE-1.
17 001700 01 RECORD-DESCRIPTION.
18 001800 03 FULL-NAME.
19 001900 05 LAST-AND-FIRST-NAMES.
20 002000 07 LAST-NAME PIC X(20).
21 002100 07 FIRST-NAME PIC X(20).
22 002200 05 MIDDLE-NAME PIC X(20).
23 002300 03 LAST-FIRST-MIDDLE-INITIAL-NAME REDEFINES FULL-NAME
002400 PIC X(41).
24 002500 03 REST-OF-RECORD PIC X(50).
002600
25 002700 PROCEDURE DIVISION.
002800 MAIN-PROGRAM SECTION.
002900 MAINLINE.
26 003000 OPEN INPUT FILE-1.
003100*
003200* POSITION THE FILE STARTING WITH RECORDS THAT HAVE A LAST NAME OF
003300* "SMITH"
27 003400 MOVE "SMITH" TO LAST-NAME.
28 003500 START FILE-1 KEY IS EQUAL TO LAST-NAME
29 003600 INVALID KEY DISPLAY "NO DATA IN SYSTEM FOR " LAST-NAME
30 003700 GO TO ERROR-ROUTINE
003800 END-START.
003900* .
004000* .
004100* .
004200*
004300* POSITION THE FILE STARTING WITH RECORDS THAT HAVE A LAST NAME OF
004400* "SMITH" AND A FIRST NAME OF "ROBERT"
31 004500 MOVE "SMITH" TO LAST-NAME.
32 004600 MOVE "ROBERT" TO FIRST-NAME.
33 004700 START FILE-1 KEY IS EQUAL TO LAST-AND-FIRST-NAMES
34 004800 INVALID KEY DISPLAY "NO DATA IN SYSTEM FOR "
004900 LAST-AND-FIRST-NAMES
35 005000 GO TO ERROR-ROUTINE
005100 END-START.
005200* .
005300* .

```

Figure 117. START Statements Using a Program-Described File (Part 1 of 2)

```

5722WDS V5R4M0 060210 LN  IBM ILE COBOL          CBLGUIDE/STRTPGMD          ISERIES1 06/02/15 14:41:49          Page    3
STMT PL SEQNBR -A 1 B.+...2....+...3....+...4....+...5....+...6....+...7..IDENTFCN S COPYNAME  CHG DATE
005400*
005500*
005600* POSITION THE FILE STARTING WITH RECORDS THAT HAVE A LAST NAME OF
005700* "SMITH", A FIRST NAME OF "ROBERT", AND A MIDDLE INITIAL OF "M"
005800
36 005900 MOVE "SMITH" TO LAST-NAME.
37 006000 MOVE "ROBERT" TO FIRST-NAME.
38 006100 MOVE "M" TO MIDDLE-NAME.
39 006200 START FILE-1 KEY IS EQUAL TO LAST-FIRST-MIDDLE-INITIAL-NAME
40 006300 INVALID KEY DISPLAY "NO DATA IN SYSTEM FOR "
006400 LAST-FIRST-MIDDLE-INITIAL-NAME
41 006500 GO TO ERROR-ROUTINE
006600 END-START.
006700
006800
006900 ERROR-ROUTINE.
42 007000 STOP RUN.
          * * * * *  E N D   O F   S O U R C E   * * * * *

```

Figure 117. START Statements Using a Program-Described File (Part 2 of 2)

Figure 118 and Figure 119 on page 483 show an example of START statements using an externally described file.

```

.....+.....1.....+.....2.....+.....3.....+.....4.....+.....5.....+.....6.....+.....7.....+.....8
A                                     UNIQUE
A          R RDE                       TEXT('RECORD DESCRIPTION')
A          FNAME          20          TEXT('FIRST NAME')
A          MINAME         1           TEXT('MIDDLE INITIAL NAME')
A          MNAME          19          TEXT('REST OF MIDDLE NAME')
A          LNAME          20          TEXT('LAST NAME')
A          PHONE          10  0       TEXT('PHONE NUMBER')
A          DATA          40          TEXT('REST OF DATA')
A          K LNAME
A          K FNAME
A          K MINAME
A          K MNAME

```

Figure 118. START Statements Using an Externally Described File -- DDS

```

Source
STMT PL SEQNBR -A 1 B.+....2...+....3...+....4...+....5...+....6...+....7..IDENTFCN S COPYNAME CHG DATE
1 000100 IDENTIFICATION DIVISION.
2 000200 PROGRAM-ID. STRTEXTD.
   000300
3 000400 ENVIRONMENT DIVISION.
4 000500 CONFIGURATION SECTION.
5 000600 SOURCE-COMPUTER. IBM-ISERIES
6 000700 OBJECT-COMPUTER. IBM-ISERIES
7 000800 INPUT-OUTPUT SECTION.
8 000900 FILE-CONTROL.
9 001000 SELECT FILE-1 ASSIGN TO DATABASE-NAMES
11 001100 ACCESS IS DYNAMIC RECORD KEY IS EXTERNALLY-DESCRIBED-KEY
13 001200 ORGANIZATION IS INDEXED.
   001300
14 001400 DATA DIVISION.
15 001500 FILE SECTION.
16 001600 FD FILE-1.
17 001700 01 RECORD-DESCRIPTION.
   001800 COPY DDS-RDE OF NAMES.
+000001* I-O FORMAT:RDE FROM FILE NAMES OF LIBRARY CBLGUIDE RDE
+000002* RECORD DESCRIPTION RDE
+000003*THE KEY DEFINITIONS FOR RECORD FORMAT RDE RDE
+000004* NUMBER NAME RETRIEVAL ALTSEQ RDE
+000005* 0001 LNAME ASCENDING NO RDE
+000006* 0002 FNAME ASCENDING NO RDE
+000007* 0003 MINAME ASCENDING NO RDE
+000008* 0004 MNAME ASCENDING NO RDE
18 +000009 05 RDE. RDE
19 +000010 06 FNAME PIC X(20). RDE
+000011* FIRST NAME RDE
20 +000012 06 MINAME PIC X(1). RDE
+000013* MIDDLE INITIAL NAME RDE
21 +000014 06 MNAME PIC X(19). RDE
+000015* REST OF MIDDLE NAME RDE
22 +000016 06 LNAME PIC X(20). RDE
+000017* LAST NAME RDE
23 +000018 06 PHONE PIC S9(10) COMP-3. RDE
+000019* PHONE NUMBER RDE
24 +000020 06 DATA-DDS PIC X(40). RDE
+000021* REST OF DATA RDE
25 001900 66 MIDDLE-NAME RENAMES MINAME THRU MNAME.
   002000
26 002100 PROCEDURE DIVISION.
   002200 MAIN-PROGRAM SECTION.
   002300 MAINLINE.
27 002400 OPEN INPUT FILE-1.
   002500*
   002600* POSITION THE FILE STARTING WITH RECORDS THAT HAVE A LAST NAME
   002700* OF "SMITH"
28 002800 MOVE "SMITH" TO LNAME.
29 002900 START FILE-1 KEY IS EQUAL TO LNAME
30 003000 INVALID KEY DISPLAY "NO DATA IN SYSTEM FOR " LNAME
31 003100 GO TO ERROR-ROUTINE
   003200 END-START.

```

Figure 119. START Statements Using an Externally Described File (Part 1 of 2)

```

5722WDS V5R4M0 060210 LN  IBM ILE COBOL          CBLGUIDE/STRTEXTD      ISERIES1  06/02/15 14:43:17      Page    3
STMT PL SEQNBR -A 1 B.+...2....+...3....+...4....+...5....+...6....+...7..IDENTFCN  S COPYNAME  CHG DATE
003300* .
003400* .
003500* .
003600* .
003700* POSITION THE FILE STARTING WITH RECORDS THAT HAVE A LAST NAME
003800* OF "SMITH" AND A FIRST NAME OF "ROBERT"
32 003900 MOVE "SMITH" TO LNAME.
33 004000 MOVE "ROBERT" TO FNAME.
34 004100 START FILE-1 KEY IS EQUAL TO LNAME, FNAME
35 004200 INVALID KEY DISPLAY "NO DATA IN SYSTEM FOR "
004300 LNAME " " FNAME
36 004400 GO TO ERROR-ROUTINE
004500 END-START.
004600* .
004700* .
004800* .
004900* .
005000* POSITION THE FILE STARTING WITH RECORDS THAT HAVE A LAST NAME OF
005100* "SMITH", A FIRST NAME OF "ROBERT", AND A MIDDLE INITIAL OF "M"
37 005200 MOVE "SMITH" TO LNAME.
38 005300 MOVE "ROBERT" TO FNAME.
39 005400 MOVE "M" TO MINAME.
40 005500 START FILE-1 KEY IS EQUAL TO LNAME, FNAME, MINAME
41 005600 INVALID KEY DISPLAY "NO DATA IN SYSTEM FOR "
005700 LNAME SPACE FNAME SPACE MINAME
42 005800 GO TO ERROR-ROUTINE
005900 END-START.
006000 .
006100 .
006200 ERROR-ROUTINE.
43 006300 STOP RUN.
      * * * * * E N D   O F   S O U R C E   * * * * *

```

Figure 119. START Statements Using an Externally Described File (Part 2 of 2)

Alternate Record Keys

Alternate keys are associated with alternate indexes, which can be temporary or permanent.

A temporary alternate index is one that ILE COBOL creates when the file is opened. When the file is closed, the temporary index no longer exists. By default, ILE COBOL will not create temporary indexes. You must specify the CRTARKIDX option to use temporary alternate indexes.

However, if ILE COBOL is able to find a permanent index, it still uses the permanent index instead of creating a temporary one. A permanent alternate index is one that persists even when the ILE COBOL program ends. Permanent indexes are associated with logical files, so you must create logical files before you can use permanent indexes in your COBOL program.

The DDS specification for the logical file should be the same as the specification for the physical file except for the key field(s). The key field(s) for the logical files should be defined to match the corresponding alternate key data-item. Note that the ILE COBOL program does not refer to these logical files in any way.

The use of permanent indexes will have a performance improvement over temporary ones. The length and starting position of the alternate key data-item within the record area must match the length and starting position of the corresponding DDS field. This DDS field also cannot be a keyed field since DDS key fields are associated with the prime key. If the alternate key data-item maps to multiple DDS fields, the starting position of the alternate key data-item must match the first DDS field, and the length of the alternate key data-item must be equal to the sum of the lengths of all the DDS fields that make up this key.

The EXTERNALLY-DESCRIBED-KEY clause cannot be specified for files that also have alternate keys.

The key used for any specific input-output request is known as the key of reference. The key of reference can be changed with the READ or START statements.

Processing Logical File as Indexed Files

When a logical file with multiple record formats, each having associated key fields, is processed as an indexed file in ILE COBOL, the following restrictions and considerations apply:

- The FORMAT phrase must be specified on all WRITE statements for the file unless a Record Format Selector Program exists and has been specified in the FMTSLR parameter of the Create Logical File (CRTLF) command, the Change Logical File (CHGLF) command, or the Override Database File (OVRDBF) command.
- If the access mode is RANDOM or DYNAMIC, and the DUPLICATES phrase is not specified for the file, the FORMAT phrase must be specified on all DELETE and REWRITE statements.
- When the FORMAT phrase is not specified, only the portion of the RECORD KEY data item that is common to all record formats for the file is used by the system as the key for the I/O statement. When the FORMAT phrase is specified, only the portion of the RECORD KEY data item that is defined for the specified record format is used by the system as the key.
- When *NONE is specified as the first key field for any format in a file, records can only be accessed sequentially. When a file is read randomly:
 - If a format name is specified, the first record with the specified format is returned.
 - If a format name is not specified, the first record in the file is returned.In both cases, the value of the RECORD KEY data item is ignored.
- For a program-defined key field:
 - Key fields within each record format must be contiguous.
 - The first key field for each record format must begin at the same relative position within each record.
 - The length of the RECORD KEY data item must be equal to the length of the longest key for any format in the file.
- For an EXTERNALLY-DESCRIBED-KEY:
 - Key fields within each record format can be noncontiguous.
 - The key fields can be defined at different positions within one record format.

Figure 120 on page 486 and Figure 121 on page 486 show examples of how to use DDS to describe the access path for indexed files.

.....	1.....	2.....	3.....	4.....	5.....	6.....	7.....	8
A		R	FORMATA			PFILE(ORDDTLP)		
A						TEXT('ACCESS PATH FOR INDEXED FILE')		
A		FLDA		14				
A		ORDERN		5S 0				
A		FLDB		101				
A		K	ORDERN					

Figure 120. Using Data Description Specifications to Define the Access Path for an Indexed File

Data description specifications can be used to create the access path for a program-described indexed file.

In the DDS, shown in Figure 120, for the record format FORMATA for the logical file ORDDTLL, the field ORDERN, which is five digits long, is defined as the key field. The definition of ORDERN as the key field establishes the keyed access path 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 in the FILE-CONTROL section in the SELECT clause as an indexed file.

The ILE COBOL descriptions of each field in the FD entry must agree with the corresponding description in the DDS file. The RECORD KEY data item must be defined as a five-digit numeric integer beginning in position 15 of the record.

.....	1.....	2.....	3.....	4.....	5.....	6.....	7.....	8
A		R	FORMATA			PFILE(ORDDTLP)		
A						TEXT('ACCESS PATH FOR INDEXED FILE')		
A		FLDA		14				
A		ORDERN		5S 0				
A		ITEM		5				
A		FLDB		96				
A		K	ORDERN					
A		K	ITEM					

Figure 121. Data Description Specifications for Defining the Access Path (a Composite Key) of an Indexed File

In this example, the DDS, shown in Figure 121, defines 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.

The ILE COBOL description of each field must agree with the corresponding description in the DDS file. A 10-character item beginning in position 15 of the record must be defined in the RECORD KEY clause of the file-control entry. The ILE COBOL descriptions of the DDS fields ORDERN and ITEM would be subordinate to the 10-character item defined in the RECORD KEY clause.

For more information on the use of format selector programs and on logical file
 # processing, refer to the *DB2 Universal Database for AS/400* section of the *Database*
 # *and File Systems* category in the **i5/OS Information Center** at this Web site -
 # <http://www.ibm.com/systems/i/infocenter/>.

Processing Files with Descending Key Sequences

Files created with a descending keyed sequence (in DDS) cause the READ statement NEXT, PRIOR, FIRST, and LAST phrases to work in a fashion exactly opposite that of a file with an ascending key sequence. You can specify a descending key sequence in the DDS with the DESCEND keyword in positions 45 to 80 beside a key field. In **descending key sequence**, the data is arranged in order from the highest value of the key field to the lowest value of the key field.

For example, READ FIRST retrieves the record with the highest key value, and READ LAST retrieves the record with the lowest key value. READ NEXT retrieves the record with the next lower key value. Files with a descending key sequence also cause the START qualifiers to work in the opposite manner. For example, START GREATER THAN positions the current record pointer to a record with a key less than the current key.

Processing Files with Variable Length Records

Variable length records are only supported for database files associated with device type DISK.

Describing DISK Files with Variable Length Records

You specify the Format 2 RECORD clause with the FD entry of the file to define the maximum and minimum record lengths for the file.

A simple file description entry in the Data Division that describes a sequential file with variable length records looks as follows:

```
FILE SECTION.  
FD sequential-file-name  
   RECORD IS VARYING IN SIZE  
       FROM integer-6 TO integer-7  
       DEPENDING ON data-name-1.  
01 minimum-sized-record.  
   05 minimum-sized-element      PIC X(integer-6).  
01 maximum-sized-record.  
   05 maximum-sized-element      PIC X(integer-7).  
:  
WORKING-STORAGE SECTION.  
77 data-name-1                    PIC 9(5).  
:  
:
```

The minimum record size of any record in the file is defined by *integer-6*. The maximum record size of any record in the file is defined by *integer-7*. Do not create record descriptions for the file that contain a record length that is less than that specified by *integer-6* nor a record length that is greater than that specified by *integer-7*. If any record descriptions break this rule, then a compile time error message is issued by the ILE COBOL compiler. The ILE COBOL compiler also issues a compile time error message when none of the record descriptions contain a record length that is as long as *integer-7*.

For **indexed files** that contain variable length records, the prime record key must be contained within the first 'n' character positions of the record, where 'n' is the minimum record size specified for the file. When processing the FD entry, the ILE COBOL compiler will check that any RECORD KEY falls within the fixed part of the record. If any key violates this rule, an error message is issued.

Opening DISK Files with Variable Length Records

The following conditions must be met for the OPEN statement to be successfully performed on a database file with variable length records:

- The formats being opened in the file must contain one variable length field at the end of the format
- The sum of the fixed length fields in all formats being opened must be the same
- The minimum record length must be greater than or equal to the sum of the fixed length fields for all formats, and less than or equal to the maximum record length for the file
- If the file is being opened for keyed sequence processing then the key must not contain any variable length fields.

If any of the above conditions are not satisfied, an error message will be generated, file status 39 will be returned, and the open operation will fail.

If an open operation is attempted on a database file with SHARE(*YES) which is already open but with a different record length than the current open operation, an error message will be generated and file status 90 will be returned.

Reading and Writing DISK Files with Variable Length Records

When a READ, WRITE, or REWRITE statement is performed on a variable length record, the size of that record is defined by the contents of *data-name-1*.

Refer to the Format 2 RECORD clause in the *IBM Rational Development Studio for i: ILE COBOL Reference* for a further description of how variable length records are handled.

You use the READ statement to read a variable length record from a database file. If the READ operation is successful then *data-name-1*, if specified, will hold the number of the character positions of the record just read. If the READ operation is unsuccessful then *data-name-1*, will hold the value it had before the READ operation was attempted.

When you specify the INTO phrase in the READ statement, the number of character positions in the current record that participate as the sending item in the implicit MOVE statement is determined by

- The contents of *data-name-1* if *data-name-1* is specified, or
- The number of character positions in the record just read if *data-name-1* is not specified.

When the READ statement is performed, if the number of character positions in the record that is read is less than the minimum record length specified by the record description entries for the file, the portion of the record area that is to the right of the last valid character read is filled with blanks. If the number of characters positions in the record that is read is greater than the maximum record length specified by the record description entries for the file, the record is truncated on the right to the maximum record size specified in the record description entries. A file status of 04 is returned when a record is read whose length falls outside the minimum or maximum record lengths defined in the RECORD clause in the file description entry for the file.

You use the WRITE or REWRITE statements to write a variable length record to a database file. You specify the length of the record to write in *data-name-1*. If you do not specify *data-name-1*, the length of the record to write is determined as follows:

- If the record contains an OCCURS...DEPENDING ON item, by the sum of the fixed portion and that portion of the table described by the number of occurrences at the time the WRITE statement is performed

- If the record does not contain an OCCURS...DEPENDING ON item, by the number of character positions in the record definition.

Examples of Processing DISK and DATABASE Files

The following sample programs illustrate the fundamental programming
techniques associated with each type of i5/OS file organization. These examples
are intended to be used for tutorial purposes only, and to illustrate the
input/output statements necessary for certain access methods. Other ILE COBOL
features (the use of the PERFORM statement, for example) are used only
incidentally. The programs illustrate:
• Sequential File Creation
• Sequential File Updating and Extension
• Relative File Creation
• Relative File Updating
• Relative File Retrieval
• Indexed File Creation
• Indexed File Updating.

Sequential File Creation

This program creates a sequential file of employee salary records. The input records are arranged in ascending order of employee number. The output file has the identical order.

```

Source
STMT PL SEQNBR -A 1 B.+....2....+....3....+....4....+....5....+....6....+....7..IDENTFCN S COPYNAME  CHG DATE
1 000100 IDENTIFICATION DIVISION.
2 000200 PROGRAM-ID. CRTSEQ.
000300
3 000400 ENVIRONMENT DIVISION.
4 000500 CONFIGURATION SECTION.
5 000600 SOURCE-COMPUTER. IBM-ISERIES
6 000700 OBJECT-COMPUTER. IBM-ISERIES
7 000800 INPUT-OUTPUT SECTION.
8 000900 FILE-CONTROL.
9 001000 SELECT INPUT-FILE ASSIGN TO DISK-FILEA
11 001100 FILE STATUS IS INPUT-FILE-STATUS.
12 001200 SELECT OUTPUT-FILE ASSIGN TO DISK-FILEB
14 001300 FILE STATUS IS OUTPUT-FILE-STATUS.
15 001400 DATA DIVISION.
16 001500 FILE SECTION.
17 001600 FD INPUT-FILE.
18 001700 01 INPUT-RECORD.
19 001800 05 INPUT-EMPLOYEE-NUMBER PICTURE 9(6).
20 001900 05 INPUT-EMPLOYEE-NAME PICTURE X(28).
21 002000 05 INPUT-EMPLOYEE-CODE PICTURE 9.
22 002100 05 INPUT-EMPLOYEE-SALARY PICTURE 9(6)V99.
23 002200 FD OUTPUT-FILE.
24 002300 01 OUTPUT-RECORD.
25 002400 05 OUTPUT-EMPLOYEE-NUMBER PICTURE 9(6).
26 002500 05 OUTPUT-EMPLOYEE-NAME PICTURE X(28).
27 002600 05 OUTPUT-EMPLOYEE-CODE PICTURE 9.
28 002700 05 OUTPUT-EMPLOYEE-SALARY PICTURE 9(6)V99.
002800
29 002900 WORKING-STORAGE SECTION.
30 003000 77 INPUT-FILE-STATUS PICTURE XX.
31 003100 77 OUTPUT-FILE-STATUS PICTURE XX.
32 003200 77 OP-NAME PICTURE X(7).
33 003300 01 INPUT-END PICTURE X VALUE SPACE.
34 003400 88 THE-END-OF-INPUT VALUE "E".
003500
35 003600 PROCEDURE DIVISION.
36 003700 DECLARATIVES.
003800 INPUT-ERROR SECTION.
003900 USE AFTER STANDARD ERROR PROCEDURE ON INPUT-FILE.
004000 INPUT-ERROR-PARA.
37 004100 DISPLAY "UNEXPECTED ERROR ON ", OP-NAME, "FOR INPUT-FILE".
38 004200 DISPLAY "FILE STATUS IS ", INPUT-FILE-STATUS.
39 004300 DISPLAY "PROCESSING ENDED".
40 004400 STOP RUN.
004500
004600 OUTPUT-ERROR SECTION.
004700 USE AFTER STANDARD ERROR PROCEDURE ON OUTPUT-FILE.
004800 OUTPUT-ERROR-PARA.
41 004900 DISPLAY "UNEXPECTED ERROR ON ", OP-NAME, "FOR OUTPUT-FILE".
42 005000 DISPLAY "FILE STATUS IS ", OUTPUT-FILE-STATUS.
43 005100 DISPLAY "PROCESSING ENDED".
44 005200 STOP RUN.
005300 END DECLARATIVES.

```

Figure 122. Example of a Sequential File of Employee Salary Records (Part 1 of 2)

```

5722WDS V5R4M0 060210 LN  IBM ILE COBOL          CBLGUIDE/CRTSEQ          ISERIES1 06/02/15 14:46:40      Page 3
STMT PL SEQNBR -A 1 B.+...2...+...3...+...4...+...5...+...6...+...7..IDENTFCN  S COPYNAME  CHG DATE
005400
005500 MAIN-PROGRAM SECTION.
005600 MAINLINE.
45 005700 MOVE "OPEN" TO OP-NAME.
46 005800 OPEN INPUT INPUT-FILE
005900 OUTPUT OUTPUT-FILE.
006000
47 006100 MOVE "READ" TO OP-NAME.
48 006200 READ INPUT-FILE INTO OUTPUT-RECORD
49 006300 AT END SET THE-END-OF-INPUT TO TRUE
006400 END-READ.
006500
50 006600 PERFORM UNTIL THE-END-OF-INPUT
51 006700 MOVE "WRITE" TO OP-NAME
52 006800 WRITE OUTPUT-RECORD
53 006900 MOVE "READ" TO OP-NAME
54 007000 READ INPUT-FILE INTO OUTPUT-RECORD
55 007100 AT END SET THE-END-OF-INPUT TO TRUE
007200 END-READ
007300 END-PERFORM.
007400
56 007500 MOVE "CLOSE" TO OP-NAME.
57 007600 CLOSE INPUT-FILE
007700 OUTPUT-FILE.
58 007800 STOP RUN.
***** END OF SOURCE *****

```

Figure 122. Example of a Sequential File of Employee Salary Records (Part 2 of 2)

Sequential File Updating and Extension

This program updates and extends the file created by the CRTSEQ program. The INPUT-FILE and the MASTER-FILE are each read. When a match is found between INPUT-EMPLOYEE-NUMBER and MST-EMPLOYEE-NUMBER, the input record replaces the original record. After the MASTER-FILE is processed, new employee records are added to the end of the file.

```

Source
STMT PL SEQNBR -A 1 B.+....2....+....3....+....4....+....5....+....6....+....7..IDENTFCN S COPYNAME  CHG DATE
1 000100 IDENTIFICATION DIVISION.
2 000200 PROGRAM-ID. UPDTSEQ.
   000300
3 000400 ENVIRONMENT DIVISION.
4 000500 CONFIGURATION SECTION.
5 000600 SOURCE-COMPUTER. IBM-ISERIES
6 000700 OBJECT-COMPUTER. IBM-ISERIES
7 000800 INPUT-OUTPUT SECTION.
8 000900 FILE-CONTROL.
9 001000     SELECT INPUT-FILE ASSIGN TO DISK-FILES
11 001100         FILE STATUS IS INPUT-FILE-STATUS.
12 001200     SELECT MASTER-FILE ASSIGN TO DISK-MSTFILEB
14 001300         FILE STATUS IS MASTER-FILE-STATUS.
   001400
15 001500 DATA DIVISION.
16 001600 FILE SECTION.
17 001700 FD INPUT-FILE.
18 001800 01 INPUT-RECORD.
19 001900     05 INPUT-EMPLOYEE-NUMBER      PICTURE 9(6).
20 002000     05 INPUT-EMPLOYEE-NAME     PICTURE X(28).
21 002100     05 INPUT-EMPLOYEE-CODE       PICTURE 9.
22 002200     05 INPUT-EMPLOYEE-SALARY     PICTURE 9(6)V99.
23 002300 FD MASTER-FILE.
24 002400 01 MASTER-RECORD.
25 002500     05 MST-EMPLOYEE-NUMBER      PICTURE 9(6).
26 002600     05 MST-EMPLOYEE-NAME     PICTURE X(28).
27 002700     05 MST-EMPLOYEE-CODE       PICTURE 9.
28 002800     05 MST-EMPLOYEE-SALARY     PICTURE 9(6)V99.
29 002900 WORKING-STORAGE SECTION.
30 003000 77 INPUT-FILE-STATUS          PICTURE XX.
31 003100 77 MASTER-FILE-STATUS        PICTURE XX.
32 003200 77 OP-NAME                  PICTURE X(12).
33 003300 01 INPUT-END                PICTURE X VALUE SPACE.
34 003400     88 THE-END-OF-INPUT      VALUE "E".
35 003500 01 MASTER-END              PICTURE X VALUE SPACE.
36 003600     88 THE-END-OF-MASTER    VALUE "E".
37 003700 PROCEDURE DIVISION.
38 003800 DECLARATIVES.
   003900 INPUT-ERROR SECTION.
   004000     USE AFTER STANDARD ERROR PROCEDURE ON INPUT-FILE.
   004100 INPUT-ERROR-PARA.
39 004200     DISPLAY "UNEXPECTED ERROR ON ", OP-NAME, "FOR INPUT-FILE".
40 004300     DISPLAY "FILE STATUS IS ", INPUT-FILE-STATUS.
41 004400     DISPLAY "PROCESSING ENDED".
42 004500     STOP RUN.
   004600
   004700 I-O-ERROR SECTION.
   004800     USE AFTER STANDARD ERROR PROCEDURE ON MASTER-FILE.
   004900 I-O-ERROR-PARA.
43 005000     DISPLAY "UNEXPECTED ERROR ON ", OP-NAME, "FOR MASTER-FILE".
44 005100     DISPLAY "FILE STATUS IS ", MASTER-FILE-STATUS.
45 005200     DISPLAY "PROCESSING ENDED".
46 005300     STOP RUN.

```

Figure 123. Example of a Sequential File Update Program (Part 1 of 2)

```

5722WDS V5R4M0 060210 LN  IBM ILE COBOL          CBLGUIDE/UPDTSEQ      ISERIES1  06/02/15 14:48:09      Page    3
STMT PL SEQNBR -A 1 B..+...2...+...3...+...4...+...5...+...6...+...7..IDENTFCN  S COPYNAME  CHG DATE
      005400 END DECLARATIVES.
      005500
      005600 MAIN-PROGRAM SECTION.
      005700 MAINLINE.
47     005800     MOVE "OPEN" TO OP-NAME.
48     005900     OPEN INPUT INPUT-FILE
      006000         I-O MASTER-FILE.
      006100
49     006200     PERFORM READ-INPUT-FILE.
50     006300     PERFORM READ-MASTER-FILE.
51     006400     PERFORM PROCESS-FILES UNTIL THE-END-OF-INPUT.
      006500
52     006600     MOVE "CLOSE" TO OP-NAME.
53     006700     CLOSE MASTER-FILE
      006800         INPUT-FILE.
54     006900     STOP RUN.
      007000
      007100 READ-INPUT-FILE.
55     007200     MOVE "READ" TO OP-NAME.
56     007300     READ INPUT-FILE
57     007400         AT END SET THE-END-OF-INPUT TO TRUE
      007500     END-READ.
      007600
      007700 READ-MASTER-FILE.
58     007800     MOVE "READ" TO OP-NAME.
59     007900     READ MASTER-FILE
      008000         AT END
60     008100         SET THE-END-OF-MASTER TO TRUE
61     008200         MOVE "AT END CLOSE" TO OP-NAME
62     008300         CLOSE MASTER-FILE
63     008400         MOVE "OPEN EXTEND" TO OP-NAME
64     008500         OPEN EXTEND MASTER-FILE
      008600     END-READ.
      008700
      008800 PROCESS-FILES.
65     008900     IF THE-END-OF-MASTER THEN
66     009000         WRITE MASTER-RECORD FROM INPUT-RECORD
67     009100         PERFORM READ-INPUT-FILE
      009200     ELSE
68     009300         IF MST-EMPLOYEE-NUMBER
69     009400             LESS THAN INPUT-EMPLOYEE-NUMBER THEN
      009500             PERFORM READ-MASTER-FILE
      009600         ELSE
70     009700             IF MST-EMPLOYEE-NUMBER EQUAL INPUT-EMPLOYEE-NUMBER THEN
71     009800                 MOVE "REWRITE" TO OP-NAME
72     009900                 REWRITE MASTER-RECORD FROM INPUT-RECORD
73     010000                 PERFORM READ-INPUT-FILE
74     010100                 PERFORM READ-MASTER-FILE
      010200             ELSE
75     010300                 DISPLAY "ERROR RECORD -> ", INPUT-EMPLOYEE-NUMBER
76     010400                 PERFORM READ-INPUT-FILE
      010500             END-IF
      010600         END-IF
      010700     END-IF.
      * * * * * E N D   O F   S O U R C E   * * * * *

```

Figure 123. Example of a Sequential File Update Program (Part 2 of 2)

Relative File Creation

This program creates a relative file of summary sales records using sequential access. Each record contains a five-year summary of unit and dollar sales for one week of the year; there are 52 records within the file, each representing one week.

Each input record represents the summary sales for one week of one year. The records for the first week of the last five years (in ascending order) are the first five input records. The records for the second week of the last five years are the next five input records, and so on. Thus, five input records fill one output record.

The RELATIVE KEY for the RELATIVE-FILE is not specified because it is not required for sequential access unless the START statement is used. (For updating, however,

the key is INPUT-WEEK.)

```
5722WDS V5R4M0 060210 LN IBM ILE COBOL CBLGUIDE/UPDTSEQ ISERIES1 06/02/15 14:48:09 Page 2
Source
STMT PL SEQNBR -A 1 B.+....2....+....3....+....4....+....5....+....6....+....7..IDENTFCN S COPYNAME CHG DATE
1 000100 IDENTIFICATION DIVISION.
2 000200 PROGRAM-ID. CRTREL.
000300
3 000400 ENVIRONMENT DIVISION.
4 000500 CONFIGURATION SECTION.
5 000600 SOURCE-COMPUTER. IBM-ISERIES
6 000700 OBJECT-COMPUTER. IBM-ISERIES
7 000800 INPUT-OUTPUT SECTION.
8 000900 FILE-CONTROL.
9 001000 SELECT RELATIVE-FILE ASSIGN TO DISK-FILED
11 001100 ORGANIZATION IS RELATIVE
12 001200 ACCESS IS SEQUENTIAL
13 001300 FILE STATUS RELATIVE-FILE-STATUS.
14 001400 SELECT INPUT-FILE ASSIGN TO DISK-FILEC
16 001500 ORGANIZATION IS SEQUENTIAL
17 001600 ACCESS IS SEQUENTIAL
18 001700 FILE STATUS INPUT-FILE-STATUS.
001800
19 001900 DATA DIVISION.
20 002000 FILE SECTION.
21 002100 FD RELATIVE-FILE.
22 002200 01 RELATIVE-RECORD-01.
23 002300 05 RELATIVE-RECORD OCCURS 5 TIMES INDEXED BY REL-INDEX.
24 002400 10 RELATIVE-YEAR PICTURE 99.
25 002500 10 RELATIVE-WEEK PICTURE 99.
26 002600 10 RELATIVE-UNIT-SALES PICTURE S9(6).
27 002700 10 RELATIVE-DOLLAR-SALES PICTURE S9(9)V99.
28 002800 FD INPUT-FILE.
29 002900 01 INPUT-RECORD.
30 003000 05 INPUT-YEAR PICTURE 99.
31 003100 05 INPUT-WEEK PICTURE 99.
32 003200 05 INPUT-UNIT-SALES PICTURE S9(6).
33 003300 05 INPUT-DOLLAR-SALES PICTURE S9(9)V99.
003400
34 003500 WORKING-STORAGE SECTION.
35 003600 77 RELATIVE-FILE-STATUS PICTURE XX.
36 003700 77 INPUT-FILE-STATUS PICTURE XX.
37 003800 77 OP-NAME PICTURE X(5).
38 003900 01 WORK-RECORD.
39 004000 05 WORK-YEAR PICTURE 99 VALUE 00.
40 004100 05 WORK-WEEK PICTURE 99.
41 004200 05 WORK-UNIT-SALES PICTURE S9(6).
42 004300 05 WORK-DOLLAR-SALES PICTURE S9(9)V99.
43 004400 01 INPUT-END PICTURE X VALUE SPACE.
44 004500 88 THE-END-OF-INPUT VALUE "E".
004600
45 004700 PROCEDURE DIVISION.
46 004800 DECLARATIVES.
004900 INPUT-ERROR SECTION.
005000 USE AFTER STANDARD ERROR PROCEDURE ON INPUT-FILE.
005100 INPUT-ERROR-PARA.
47 005200 DISPLAY "UNEXPECTED ERROR ON ", OP-NAME, " FOR INPUT-FILE ".
48 005300 DISPLAY "FILE STATUS IS ", INPUT-FILE-STATUS.
```

Figure 124. Example of a Relative File Program (Part 1 of 2)

```

5722WDS V5R4M0 060210 LN  IBM ILE COBOL          CBLGUIDE/CRTREL          ISERIES1 06/02/15 14:49:23      Page    3
STMT PL SEQNBR -A 1 B..+...2...+...3...+...4...+...5...+...6...+...7..IDENTFCN  S COPYNAME  CHG DATE
 49  005400  DISPLAY "PROCESSING ENDED"
 50  005500  STOP RUN.
      005600
      005700  OUTPUT-ERROR SECTION.
      005800  USE AFTER STANDARD ERROR PROCEDURE ON RELATIVE-FILE.
      005900  OUTPUT-ERROR-PARA.
 51  006000  DISPLAY "UNEXPECTED ERROR ON ", OP-NAME, " FOR RELATIVE-FILE".
 52  006100  DISPLAY "FILE STATUS IS ", RELATIVE-FILE-STATUS.
 53  006200  DISPLAY "PROCESSING ENDED"
 54  006300  STOP RUN.
      006400  END DECLARATIVES.
      006500
      006600  MAIN-PROGRAM SECTION.
      006700  MAINLINE.
 55  006800  MOVE "OPEN" TO OP-NAME.
 56  006900  OPEN INPUT INPUT-FILE
      007000  OUTPUT RELATIVE-FILE.
      007100
 57  007200  SET REL-INDEX TO 1.
 58  007300  MOVE "READ" TO OP-NAME.
 59  007400  READ INPUT-FILE
 60  007500  AT END SET THE-END-OF-INPUT TO TRUE
      007600  END-READ.
      007700
 61  007800  PERFORM UNTIL THE-END-OF-INPUT
 62  007900  MOVE INPUT-RECORD TO RELATIVE-RECORD (REL-INDEX)
 63  008000  IF REL-INDEX NOT = 5
 64  008100  SET REL-INDEX UP BY 1
      008200  ELSE
 65  008300  SET REL-INDEX TO 1
 66  008400  MOVE "WRITE" TO OP-NAME
 67  008500  WRITE RELATIVE-RECORD-01
      008600  END-IF
      008700
 68  008800  MOVE "READ" TO OP-NAME
 69  008900  READ INPUT-FILE
 70  009000  AT END SET THE-END-OF-INPUT TO TRUE
      009100  END-READ
      009200  END-PERFORM.
      009300
 71  009400  CLOSE RELATIVE-FILE
      009500  INPUT-FILE.
 72  009600  STOP RUN.
      009700
          * * * * *  E N D   O F   S O U R C E   * * * * *

```

Figure 124. Example of a Relative File Program (Part 2 of 2)

Relative File Updating

This program uses sequential access to update the file of summary sales records created in the CRTREL program. The updating program adds a record for the new year and deletes the oldest year's records from RELATIVE-FILE.

The input record represents the summary sales record for one week of the preceding year. The RELATIVE KEY for the RELATIVE-FILE is in the input record as INPUT-WEEK.

```

Source
STMT PL SEQNBR -A 1 B.+....2....+....3....+....4....+....5....+....6....+....7..IDENTFCN S COPYNAME  CHG DATE
1 000100 IDENTIFICATION DIVISION.
2 000200 PROGRAM-ID. UPDTREL.
000300
3 000400 ENVIRONMENT DIVISION.
4 000500 CONFIGURATION SECTION.
5 000600 SOURCE-COMPUTER. IBM-ISERIES
6 000700 OBJECT-COMPUTER. IBM-ISERIES
7 000800 INPUT-OUTPUT SECTION.
8 000900 FILE-CONTROL.
9 001000 SELECT RELATIVE-FILE ASSIGN TO DISK-FILED
11 001100 ORGANIZATION IS RELATIVE
12 001200 ACCESS IS SEQUENTIAL
13 001300 RELATIVE KEY INPUT-WEEK
14 001400 FILE STATUS RELATIVE-FILE-STATUS.
15 001500 SELECT INPUT-FILE ASSIGN TO DISK-FILES2
17 001600 ORGANIZATION IS SEQUENTIAL
18 001700 ACCESS IS SEQUENTIAL
19 001800 FILE STATUS INPUT-FILE-STATUS.
001900
20 002000 DATA DIVISION.
21 002100 FILE SECTION.
22 002200 FD RELATIVE-FILE.
23 002300 01 RELATIVE-RECORD PICTURE X(105).
24 002400 FD INPUT-FILE.
25 002500 01 INPUT-RECORD.
26 002600 05 INPUT-YEAR PICTURE 99.
27 002700 05 INPUT-WEEK PICTURE 99.
28 002800 05 INPUT-UNIT-SALES PICTURE S9(6).
29 002900 05 INPUT-DOLLAR-SALES PICTURE S9(9)V99.
003000
30 003100 WORKING-STORAGE SECTION.
31 003200 77 RELATIVE-FILE-STATUS PICTURE XX.
32 003300 77 INPUT-FILE-STATUS PICTURE XX.
33 003400 77 OP-NAME PICTURE X(7).
34 003500 01 WORK-RECORD.
35 003600 05 FILLER PICTURE X(21).
36 003700 05 CURRENT-WORK-YEARS PICTURE X(84).
37 003800 05 NEW-WORK-YEAR.
38 003900 10 WORK-YEAR PICTURE 99.
39 004000 10 WORK-WEEK PICTURE 99.
40 004100 10 WORK-UNIT-SALES PICTURE S9(6).
41 004200 10 WORK-DOLLAR-SALES PICTURE S9(9)V99.
42 004300 66 WORK-OUT-RECORD RENAMES
004400 CURRENT-WORK-YEARS THROUGH NEW-WORK-YEAR.
43 004500 01 INPUT-END PICTURE X VALUE SPACE.
44 004600 88 THE-END-OF-INPUT VALUE "E".
004700
45 004800 PROCEDURE DIVISION.
46 004900 DECLARATIVES.
005000 INPUT-ERROR SECTION.
005100 USE AFTER STANDARD ERROR PROCEDURE ON INPUT-FILE.
005200 INPUT-ERROR-PARA.
47 005300 DISPLAY "UNEXPECTED ERROR ON ", OP-NAME, " FOR INPUT-FILE ".

```

Figure 125. Example of a Relative File Update Program (Part 1 of 2)

```

5722WDS V5R4M0 060210 LN  IBM ILE COBOL                CBLGUIDE/UPDTREL          ISERIES1  06/02/15 14:50:35      Page    3
STMT PL SEQNBR -A 1 B..+...2...+...3...+...4...+...5...+...6...+...7..IDENTFCN  S COPYNAME  CHG DATE
48  005400  DISPLAY "FILE STATUS IS ", INPUT-FILE-STATUS.
49  005500  DISPLAY "PROCESSING ENDED"
50  005600  STOP RUN.
    005700
    005800  I-O-ERROR SECTION.
    005900      USE AFTER STANDARD ERROR PROCEDURE ON RELATIVE-FILE.
    006000  I-O-ERROR-PARA.
51  006100  DISPLAY "UNEXPECTED ERROR ON ", OP-NAME, " FOR RELATIVE-FILE".
52  006200  DISPLAY "FILE STATUS IS ", RELATIVE-FILE-STATUS.
53  006300  DISPLAY "PROCESSING ENDED"
54  006400  STOP RUN.
    006500  END DECLARATIVES.
    006600
    006700  MAIN-PROGRAM SECTION.
    006800  MAINLINE.
55  006900  MOVE "OPEN" TO OP-NAME.
56  007000  OPEN INPUT INPUT-FILE
    007100      I-O RELATIVE-FILE.
    007200
57  007300  MOVE "READ" TO OP-NAME.
58  007400  READ RELATIVE-FILE INTO WORK-RECORD
59  007500      AT END SET THE-END-OF-INPUT TO TRUE
    007600  END-READ.
60  007700  READ INPUT-FILE INTO NEW-WORK-YEAR
61  007800      AT END SET THE-END-OF-INPUT TO TRUE
    007900  END-READ.
    008000
62  008100  PERFORM UNTIL THE-END-OF-INPUT
63  008200      MOVE "REWRITE" TO OP-NAME
64  008300      REWRITE RELATIVE-RECORD FROM WORK-OUT-RECORD
    008400
65  008500      MOVE "READ" TO OP-NAME
66  008600      READ RELATIVE-FILE INTO WORK-RECORD
67  008700          AT END SET THE-END-OF-INPUT TO TRUE
    008800      END-READ
68  008900      READ INPUT-FILE INTO NEW-WORK-YEAR
69  009000          AT END SET THE-END-OF-INPUT TO TRUE
    009100      END-READ
    009200  END-PERFORM.
    009300
70  009400  MOVE "CLOSE" TO OP-NAME.
71  009500  CLOSE INPUT-FILE
    009600      RELATIVE-FILE.
72  009700  STOP RUN.
    009800
          * * * * *  E N D   O F   S O U R C E  * * * * *

```

Figure 125. Example of a Relative File Update Program (Part 2 of 2)

Relative File Retrieval

This program, using dynamic access, retrieves the summary file created by the CRTREL program.

The records of the INPUT-FILE contain one required field (INPUT-WEEK), which is the RELATIVE KEY for RELATIVE-FILE, and one optional field (END-WEEK). An input record containing data in INPUT-WEEK and spaces in END-WEEK requests a printout for that one specific RELATIVE-RECORD; the record is retrieved through random access. An input record containing data in both INPUT-WEEK and END-WEEK requests a printout of all the RELATIVE-FILE records within the RELATIVE KEY range of INPUT-WEEK through END-WEEK inclusive. These records are retrieved through sequential access.

```

Source
STMT PL SEQNBR -A 1 B.+....2....+....3....+....4....+....5....+....6....+....7..IDENTFCN S COPYNAME CHG DATE
1 000100 IDENTIFICATION DIVISION.
2 000200 PROGRAM-ID. RTRVREL.
000300
3 000400 ENVIRONMENT DIVISION.
4 000500 CONFIGURATION SECTION.
5 000600 SOURCE-COMPUTER. IBM-ISERIES
6 000700 OBJECT-COMPUTER. IBM-ISERIES
7 000800 INPUT-OUTPUT SECTION.
8 000900 FILE-CONTROL.
9 001000 SELECT RELATIVE-FILE ASSIGN TO DISK-FILED
11 001100 ORGANIZATION IS RELATIVE
12 001200 ACCESS IS DYNAMIC
13 001300 RELATIVE KEY INPUT-WEEK
14 001400 FILE STATUS IS RELATIVE-FILE-STATUS.
15 001500 SELECT INPUT-FILE ASSIGN TO DISK-FILEF
17 001600 FILE STATUS IS INPUT-FILE-STATUS.
18 001700 SELECT PRINT-FILE ASSIGN TO PRINTER-QSYSPRT
20 001800 FILE STATUS IS PRINT-FILE-STATUS.
001900
21 002000 DATA DIVISION.
22 002100 FILE SECTION.
23 002200 FD RELATIVE-FILE.
24 002300 01 RELATIVE-RECORD-01.
25 002400 05 RELATIVE-RECORD OCCURS 5 TIMES INDEXED BY REL-INDEX.
26 002500 10 RELATIVE-YEAR PICTURE 99.
27 002600 10 RELATIVE-WEEK PICTURE 99.
28 002700 10 RELATIVE-UNIT-SALES PICTURE S9(6).
29 002800 10 RELATIVE-DOLLAR-SALES PICTURE S9(9)V99.
30 002900 FD INPUT-FILE.
31 003000 01 INPUT-RECORD.
32 003100 05 INPUT-WEEK PICTURE 99.
33 003200 05 END-WEEK PICTURE 99.
34 003300 FD PRINT-FILE.
35 003400 01 PRINT-RECORD.
36 003500 05 PRINT-WEEK PICTURE 99.
37 003600 05 FILLER PICTURE X(5).
38 003700 05 PRINT-YEAR PICTURE 99.
39 003800 05 FILLER PICTURE X(5).
40 003900 05 PRINT-UNIT-SALES PICTURE ZZ,ZZ9.
41 004000 05 FILLER PICTURE X(5).
42 004100 05 PRINT-DOLLAR-SALES PICTURE $$$,$$$,$$$99.
004200
43 004300 WORKING-STORAGE SECTION.
44 004400 77 RELATIVE-FILE-STATUS PICTURE XX.
45 004500 77 INPUT-FILE-STATUS PICTURE XX.
46 004600 77 PRINT-FILE-STATUS PICTURE XX.
47 004700 77 HIGH-WEEK PICTURE 99 VALUE 53.
48 004800 77 OP-NAME PICTURE X(9).
49 004900 01 INPUT-END PICTURE X(9).
50 005000 88 THE-END-OF-INPUT VALUE "E".
005100
51 005200 PROCEDURE DIVISION.
52 005300 DECLARATIVES.

```

Figure 126. Example of a Relative File Retrieval Program (Part 1 of 3)

```

STMT PL SEQNBR -A 1 B.+...2...+...3...+...4...+...5...+...6...+...7..IDENTFCN S COPYNAME CHG DATE
005400 RELATIVE-FILE-ERROR SECTION.
005500 USE AFTER STANDARD ERROR PROCEDURE ON RELATIVE-FILE.
005600 RELATIVE-ERROR-PARA.
53 005700 DISPLAY "UNEXPECTED ERROR ON ", OP-NAME, " FOR RELATIVE-FILE".
54 005800 DISPLAY "FILE STATUS IS ", RELATIVE-FILE-STATUS.
55 005900 DISPLAY "PROCESSING ENDED"
56 006000 STOP RUN.
006100
006200 INPUT-FILE-ERROR SECTION.
006300 USE AFTER STANDARD ERROR PROCEDURE ON INPUT-FILE.
006400 INPUT-ERROR-PARA.
57 006500 DISPLAY "UNEXPECTED ERROR ON ", OP-NAME, " FOR INPUT-FILE ".
58 006600 DISPLAY "FILE STATUS IS ", INPUT-FILE-STATUS.
59 006700 DISPLAY "PROCESSING ENDED"
60 006800 STOP RUN.
006900
007000 PRINT-FILE-ERROR SECTION.
007100 USE AFTER STANDARD ERROR PROCEDURE ON PRINT-FILE.
007200 PRINT-ERROR-MSG.
61 007300 DISPLAY "UNEXPECTED ERROR ON ", OP-NAME, " FOR PRINT-FILE ".
62 007400 DISPLAY "FILE STATUS IS ", PRINT-FILE-STATUS.
63 007500 DISPLAY "PROCESSING ENDED"
64 007600 STOP RUN.
007700 END DECLARATIVES.
007800
007900 MAIN-PROGRAM SECTION.
008000 MAINLINE.
65 008100 MOVE "OPEN" TO OP-NAME.
66 008200 OPEN INPUT INPUT-FILE
008300 RELATIVE-FILE
008400 OUTPUT PRINT-FILE.
008500
67 008600 MOVE SPACES TO PRINT-RECORD.
68 008700 PERFORM READ-INPUT-FILE.
69 008800 PERFORM CONTROL-PROCESS THRU READ-INPUT-FILE
008900 UNTIL THE-END-OF-INPUT.
009000
70 009100 MOVE "CLOSE" TO OP-NAME.
71 009200 CLOSE RELATIVE-FILE
009300 INPUT-FILE
009400 PRINT-FILE.
72 009500 STOP RUN.
009600
009700 CONTROL-PROCESS.
73 009800 IF (END-WEEK = SPACES OR END-WEEK = 00)
74 009900 MOVE "READ" TO OP-NAME
75 010000 READ RELATIVE-FILE
76 010100 PERFORM PRINT-SUMMARY VARYING REL-INDEX FROM 1 BY 1
010200 UNTIL REL-INDEX > 5
010300 ELSE
77 010400 MOVE "READ" TO OP-NAME
78 010500 READ RELATIVE-FILE
79 010600 PERFORM READ-REL-SEQ
010700 UNTIL RELATIVE-WEEK(1) GREATER THAN END-WEEK
010800 END-IF.

```

Figure 126. Example of a Relative File Retrieval Program (Part 2 of 3)

```

5722WDS V5R4M0 060210 LN  IBM ILE COBOL          CBLGUIDE/RTRVREL          ISERIES1  06/02/15 14:51:40          Page      4
STMT PL SEQNBR -A 1 B.+...2...+...3...+...4...+...5...+...6...+...7..IDENTFCN S COPYNAME  CHG DATE
010900
011000 READ-INPUT-FILE.
80 011100 MOVE "READ" TO OP-NAME.
81 011200 READ INPUT-FILE
82 011300 AT END SET THE-END-OF-INPUT TO TRUE
011400 END-READ.
011500
011600 READ-REL-SEQ.
83 011700 PERFORM PRINT-SUMMARY VARYING REL-INDEX FROM 1 BY 1
011800 UNTIL REL-INDEX > 5.
84 011900 MOVE "READ NEXT" TO OP-NAME.
85 012000 READ RELATIVE-FILE NEXT RECORD
86 012100 AT END MOVE HIGH-WEEK TO RELATIVE-WEEK(1)
012200 END-READ.
012300
012400 PRINT-SUMMARY.
87 012500 MOVE RELATIVE-YEAR (REL-INDEX) TO PRINT-YEAR.
88 012600 MOVE RELATIVE-WEEK (REL-INDEX) TO PRINT-WEEK.
89 012700 MOVE RELATIVE-UNIT-SALES (REL-INDEX) TO PRINT-UNIT-SALES.
90 012800 MOVE RELATIVE-DOLLAR-SALES(REL-INDEX) TO PRINT-DOLLAR-SALES.
91 012900 MOVE "WRITE" TO OP-NAME.
92 013000 WRITE PRINT-RECORD AFTER ADVANCING 2 LINES
013100 END-WRITE.
          * * * * *  E N D   O F   S O U R C E   * * * * *

```

Figure 126. Example of a Relative File Retrieval Program (Part 3 of 3)

Indexed File Creation

This program creates an indexed file of summary records for bank depositors. The key within each indexed file record is INDEX-KEY (the depositor's account number); the input records are ordered in ascending sequence upon this key. Records are read from the input file and transferred to the indexed file record area. The indexed file record is then written.

```

Source
STMT PL SEQNBR -A 1 B.+....2....+....3....+....4....+....5....+....6....+....7..IDENTFCN S COPYNAME CHG DATE
1 000100 IDENTIFICATION DIVISION.
2 000200 PROGRAM-ID. CRTIND.
000300
3 000400 ENVIRONMENT DIVISION.
4 000500 CONFIGURATION SECTION.
5 000600 SOURCE-COMPUTER. IBM-ISERIES
6 000700 OBJECT-COMPUTER. IBM-ISERIES
7 000800 INPUT-OUTPUT SECTION.
8 000900 FILE-CONTROL.
9 001000 SELECT INDEXED-FILE ASSIGN TO DISK-INDEXFILE
11 001100 ORGANIZATION IS INDEXED
12 001200 ACCESS IS SEQUENTIAL
13 001300 RECORD KEY IS INDEX-KEY
14 001400 FILE STATUS IS INDEXED-FILE-STATUS.
15 001500 SELECT INPUT-FILE ASSIGN TO DISK-FILEG
17 001600 FILE STATUS IS INPUT-FILE-STATUS.
001700
18 001800 DATA DIVISION.
19 001900 FILE SECTION.
20 002000 FD INDEXED-FILE.
21 002100 01 INDEX-RECORD.
22 002200 05 INDEX-KEY PICTURE X(10).
23 002300 05 INDEX-FLD1 PICTURE X(10).
24 002400 05 INDEX-NAME PICTURE X(20).
25 002500 05 INDEX-BAL PICTURE S9(5)V99.
26 002600 FD INPUT-FILE.
27 002700 01 INPUT-RECORD.
28 002800 05 INPUT-KEY PICTURE X(10).
29 002900 05 INPUT-NAME PICTURE X(20).
30 003000 05 INPUT-BAL PICTURE S9(5)V99.
31 003100 WORKING-STORAGE SECTION.
32 003200 77 INDEXED-FILE-STATUS PICTURE XX.
33 003300 77 INPUT-FILE-STATUS PICTURE XX.
34 003400 77 OP-NAME PICTURE X(7).
35 003500 01 INPUT-END PICTURE X VALUE SPACES.
36 003600 88 THE-END-OF-INPUT VALUE "E".
003700
37 003800 PROCEDURE DIVISION.
38 003900 DECLARATIVES.
004000 INPUT-ERROR SECTION.
004100 USE AFTER STANDARD ERROR PROCEDURE ON INPUT-FILE.
004200 INPUT-ERROR-PARA.
39 004300 DISPLAY "UNEXPECTED ERROR ON ", OP-NAME, " FOR INPUT-FILE ".
40 004400 DISPLAY "FILE STATUS IS ", INPUT-FILE-STATUS.
41 004500 DISPLAY "PROCESSING ENDED"
42 004600 STOP RUN.
004700
004800 OUTPUT-ERROR SECTION.
004900 USE AFTER STANDARD ERROR PROCEDURE ON INDEXED-FILE.
005000 OUTPUT-ERROR-PARA.
43 005100 DISPLAY "UNEXPECTED ERROR ON ", OP-NAME, " FOR INDEXED-FILE ".
44 005200 DISPLAY "FILE STATUS IS ", INDEXED-FILE-STATUS.
45 005300 DISPLAY "PROCESSING ENDED"

```

Figure 127. Example of an Indexed File Program (Part 1 of 2)

```

5722WDS V5R4M0 060210 LN IBM ILE COBOL          CBLGUIDE/CRTIND          ISERIES1 06/02/15 14:52:38      Page 3
STMT PL SEQNBR -A 1 B.+...2....+...3....+...4....+...5....+...6....+...7..IDENTFCN S COPYNAME  CHG DATE
 46 005400 STOP RUN.
    005500 END DECLARATIVES.
    005600
    005700 MAIN-PROGRAM SECTION.
    005800 MAINLINE.
 47 005900 MOVE "OPEN" TO OP-NAME.
 48 006000 OPEN INPUT INPUT-FILE
    006100 OUTPUT INDEXED-FILE.
    006200
 49 006300 MOVE "READ" TO OP-NAME.
 50 006400 READ INPUT-FILE
 51 006500 AT END SET THE-END-OF-INPUT TO TRUE
    006600 END-READ.
    006700
 52 006800 PERFORM UNTIL THE-END-OF-INPUT
 53 006900 MOVE INPUT-KEY TO INDEX-KEY
 54 007000 MOVE INPUT-NAME TO INDEX-NAME
 55 007100 MOVE INPUT-BAL TO INDEX-BAL
 56 007200 MOVE SPACES TO INDEX-FLD1
 57 007300 MOVE "WRITE" TO OP-NAME
 58 007400 WRITE INDEX-RECORD
    007500
 59 007600 MOVE "READ" TO OP-NAME
 60 007700 READ INPUT-FILE
 61 007800 AT END SET THE-END-OF-INPUT TO TRUE
    007900 END-READ
    008000 END-PERFORM.
    008100
 62 008200 MOVE "CLOSE" TO OP-NAME.
 63 008300 CLOSE INPUT-FILE
    008400 INDEXED-FILE.
 64 008500 STOP RUN.
    008600
          * * * * * E N D   O F   S O U R C E   * * * * *

```

Figure 127. Example of an Indexed File Program (Part 2 of 2)

Indexed File Updating

This program, using dynamic access, updates the indexed file created in the CRTIND program.

The input records contain the key for the record, the depositor name, and the amount of the transaction.

When the input record is read, the program determines if it is:

- A transaction record (in which case, all fields of the record are filled)
- A record requesting sequential retrieval of a specific generic class (in which case, only the INPUT-GEN-FLD field of the input record contains data).

Random access is used for updating and printing the transaction records. Sequential access is used for retrieving and printing all records within one generic class.

```

          S o u r c e
STMT PL SEQNBR -A 1 B.+....2....+....3....+....4....+....5....+....6....+....7..IDENTFCN  S COPYNAME  CHG DATE
 1 000100 IDENTIFICATION DIVISION.
 2 000200 PROGRAM-ID. UPDTIND.
   000300
 3 000400 ENVIRONMENT DIVISION.
 4 000500 CONFIGURATION SECTION.
 5 000600 SOURCE-COMPUTER. IBM-ISERIES
 6 000700 OBJECT-COMPUTER. IBM-ISERIES
 7 000800 INPUT-OUTPUT SECTION.
 8 000900 FILE-CONTROL.
 9 001000 SELECT INDEXED-FILE ASSIGN TO DISK-INDEXFILE
11 001100 ORGANIZATION IS INDEXED
12 001200 ACCESS IS DYNAMIC
13 001300 RECORD KEY IS INDEX-KEY
14 001400 FILE STATUS IS INDEXED-FILE-STATUS.
15 001500 SELECT INPUT-FILE ASSIGN TO DISK-FILEH
17 001600 FILE STATUS IS INPUT-FILE-STATUS.
18 001700 SELECT PRINT-FILE ASSIGN TO PRINTER-OSYSPRT
20 001800 FILE STATUS IS PRINT-FILE-STATUS.
   001900
21 002000 DATA DIVISION.
22 002100 FILE SECTION.
23 002200 FD INDEXED-FILE.
24 002300 01 INDEX-RECORD.
25 002400 05 INDEX-KEY.
26 002500 10 INDEX-GEN-FLD PICTURE X(5).
27 002600 10 INDEX-DET-FLD PICTURE X(5).
28 002700 05 INDEX-FLD1 PICTURE X(10).
29 002800 05 INDEX-NAME PICTURE X(20).
30 002900 05 INDEX-BAL PICTURE S9(5)V99.
31 003000 FD INPUT-FILE.
32 003100 01 INPUT-REC.
33 003200 05 INPUT-KEY.
34 003300 10 INPUT-GEN-FLD PICTURE X(5).
35 003400 10 INPUT-DET-FLD PICTURE X(5).
36 003500 05 INPUT-NAME PICTURE X(20).
37 003600 05 INPUT-AMT PICTURE S9(5)V99.
38 003700 FD PRINT-FILE
   003800 LINAGE 12 LINES FOOTING AT 9.
39 003900 01 PRINT-RECORD-1.
40 004000 05 PRINT-KEY PICTURE X(10).
41 004100 05 FILLER PICTURE X(5).
42 004200 05 PRINT-NAME PICTURE X(20).
43 004300 05 FILLER PICTURE X(5).
44 004400 05 PRINT-BAL PICTURE $$$,$$9.99-.
45 004500 05 FILLER PICTURE X(7).
46 004600 05 PRINT-AMT PICTURE $$$,$$9.99-.
47 004700 05 FILLER PICTURE X(5).
48 004800 05 PRINT-NEW-BAL PICTURE $$$,$$9.99-.
49 004900 01 PRINT-RECORD-2 PICTURE X(89).
   005000
50 005100 WORKING-STORAGE SECTION.
51 005200 77 INDEXED-FILE-STATUS PICTURE XX.
52 005300 77 INPUT-FILE-STATUS PICTURE XX.

```

Figure 128. Example of an Indexed File Update Program (Part 1 of 4)

```

STMT PL SEQNBR -A 1 B...2...3...4...5...6...7..IDENTFCN S COPYNAME CHG DATE
53 005400 77 PRINT-FILE-STATUS PICTURE XX.
54 005500 77 OP-NAME PICTURE X(9).
55 005600 77 LINES-TO-FOOT PICTURE 99.
56 005700 01 PAGE-HEAD.
57 005800 05 FILLER PICTURE X(38) VALUE SPACES.
58 005900 05 FILLER PICTURE X(13) VALUE "UPDATE REPORT".
59 006000 05 FILLER PICTURE X(38) VALUE SPACES.
60 006100 01 COLUMN-HEAD.
61 006200 05 FILLER PICTURE X(6) VALUE "KEY ID".
62 006300 05 FILLER PICTURE X(9) VALUE SPACES.
63 006400 05 FILLER PICTURE X(4) VALUE "NAME".
64 006500 05 FILLER PICTURE X(21) VALUE SPACES.
65 006600 05 FILLER PICTURE X(11) VALUE "CUR BALANCE".
66 006700 05 FILLER PICTURE X(6) VALUE SPACES.
67 006800 05 FILLER PICTURE X(13) VALUE "UPDATE AMOUNT".
68 006900 05 FILLER PICTURE X(4) VALUE SPACES.
69 007000 05 FILLER PICTURE X(11) VALUE "NEW BALANCE".
70 007100 05 FILLER PICTURE X(4) VALUE SPACES.
71 007200 01 PAGE-FOOT.
72 007300 05 FILLER PICTURE X(81) VALUE SPACES.
73 007400 05 FILLER PICTURE A(6) VALUE "PAGE ".
74 007500 05 PG-NUMBER PICTURE 99 VALUE 00.
007600
75 007700 01 INPUT-END PICTURE X VALUE SPACE.
76 007800 88 THE-END-OF-INPUT VALUE "E".
007900
77 008000 PROCEDURE DIVISION.
78 008100 DECLARATIVES.
008200 INPUT-ERROR SECTION.
008300 USE AFTER STANDARD ERROR PROCEDURE ON INPUT-FILE.
008400 INPUT-ERROR-PARA.
79 008500 DISPLAY "UNEXPECTED ERROR ON ", OP-NAME, " FOR INPUT-FILE ".
80 008600 DISPLAY "FILE STATUS IS ", INPUT-FILE-STATUS.
81 008700 DISPLAY "PROCESSING ENDED"
82 008800 STOP RUN.
008900
009000 I-O-ERROR SECTION.
009100 USE AFTER STANDARD ERROR PROCEDURE ON INDEXED-FILE.
009200 I-O-ERROR-PARA.
83 009300 DISPLAY "UNEXPECTED ERROR ON ", OP-NAME, " FOR INDEXED-FILE ".
84 009400 DISPLAY "FILE STATUS IS ", INDEXED-FILE-STATUS.
85 009500 DISPLAY "PROCESSING ENDED"
86 009600 STOP RUN.
009700
009800 OUTPUT-ERROR SECTION.
009900 USE AFTER STANDARD ERROR PROCEDURE ON PRINT-FILE.
010000 OUTPUT-ERROR-PARA.
87 010100 DISPLAY "UNEXPECTED ERROR ON ", OP-NAME, " FOR PRINT-FILE ".
88 010200 DISPLAY "FILE STATUS IS ", PRINT-FILE-STATUS.
89 010300 DISPLAY "PROCESSING ENDED"
90 010400 STOP RUN.
010500 END DECLARATIVES.
010600
010700 MAIN-PROGRAM SECTION.
010800 MAINLINE.

```

Figure 128. Example of an Indexed File Update Program (Part 2 of 4)

```

STMT PL SEQNBR -A 1 B.+...2....+...3....+...4....+...5....+...6....+...7..IDENTFCN S COPYNAME CHG DATE
91 010900 MOVE "OPEN" TO OP-NAME.
92 011000 OPEN INPUT INPUT-FILE
    011100 I-O INDEXED-FILE
    011200 OUTPUT PRINT-FILE.
    011300
93 011400 PERFORM PAGE-START.
94 011500 PERFORM READ-INPUT-FILE.
95 011600 PERFORM PROCESS-DATA THRU READ-INPUT-FILE
    011700 UNTIL THE-END-OF-INPUT.
96 011800 PERFORM PAGE-END.
    011900
97 012000 MOVE "CLOSE" TO OP-NAME.
98 012100 CLOSE INPUT-FILE
    012200 INDEXED-FILE
    012300 PRINT-FILE.
99 012400 STOP RUN.
    012500
    012600 PROCESS-DATA.
100 012700 IF INPUT-DET-FLD EQUAL SPACES
101 012800 MOVE INPUT-GEN-FLD TO INDEX-GEN-FLD
102 012900 MOVE "START" TO OP-NAME
103 013000 START INDEXED-FILE
    013100 KEY IS NOT LESS THAN INDEX-GEN-FLD
    013200 END-START
104 013300 PERFORM SEQUENTIAL-PROCESS
    013400 UNTIL INPUT-GEN-FLD NOT EQUAL INDEX-GEN-FLD
    013500 ELSE
105 013600 MOVE INPUT-KEY TO INDEX-KEY
106 013700 MOVE "READ" TO OP-NAME
107 013800 READ INDEXED-FILE
108 013900 IF INPUT-GEN-FLD EQUAL INDEX-GEN-FLD THEN
109 014000 MOVE INDEX-KEY TO PRINT-KEY
110 014100 MOVE INDEX-NAME TO PRINT-NAME
111 014200 MOVE INDEX-BAL TO PRINT-BAL
112 014300 MOVE INPUT-AMT TO PRINT-AMT
113 014400 ADD INPUT-AMT TO INDEX-BAL
114 014500 MOVE INDEX-BAL TO PRINT-NEW-BAL
115 014600 PERFORM PRINT-DETAIL
116 014700 MOVE "REWRITE" TO OP-NAME
117 014800 REWRITE INDEX-RECORD
    014900 END-IF
    015000 END-IF.
    015100
    015200 READ-INPUT-FILE.
118 015300 MOVE "READ" TO OP-NAME.
119 015400 READ INPUT-FILE
120 015500 AT END SET THE-END-OF-INPUT TO TRUE
    015600 END-READ.
    015700
    015800 SEQUENTIAL-PROCESS.
121 015900 MOVE "READ NEXT" TO OP-NAME.
122 016000 READ INDEXED-FILE NEXT RECORD
123 016100 AT END MOVE HIGH-VALUE TO INDEX-GEN-FLD
    016200 END-READ.
124 016300 IF INPUT-GEN-FLD EQUAL INDEX-GEN-FLD THEN

```

Figure 128. Example of an Indexed File Update Program (Part 3 of 4)

```

5722WDS V5R4M0 060210 LN IBM ILE COBOL          CBLGUIDE/UPDTIND      ISERIES1 06/02/15 14:54:04      Page    5
STMT PL SEQNBR -A 1 B..+...2...+...3...+...4...+...5...+...6...+...7..IDENTFCN S COPYNAME  CHG DATE
125 016400      MOVE INDEX-KEY TO PRINT-KEY
126 016500      MOVE INDEX-NAME TO PRINT-NAME
127 016600      MOVE INDEX-BAL TO PRINT-NEW-BAL
128 016700      PERFORM PRINT-DETAIL
      016800      END-IF.
      016900
      017000 PRINT-DETAIL.
129 017100      MOVE "WRITE" TO OP-NAME.
130 017200      WRITE PRINT-RECORD-1
      017300      AT END-OF-PAGE
131 017400      PERFORM PAGE-END THROUGH PAGE-START
      017500      END-WRITE.
132 017600      MOVE SPACES TO PRINT-RECORD-1.
      017700
      017800 PAGE-END.
133 017900      MOVE "WRITE" TO OP-NAME.
134 018000      ADD 1 TO PG-NUMBER.
135 018100      SUBTRACT LINAGE-COUNTER OF PRINT-FILE FROM 12
      018200      GIVING LINES-TO-FOOT.
136 018300      MOVE SPACES TO PRINT-RECORD-1.
137 018400      WRITE PRINT-RECORD-1
      018500      AFTER ADVANCING LINES-TO-FOOT
      018600      END-WRITE.
138 018700      WRITE PRINT-RECORD-2 FROM PAGE-FOOT
      018800      BEFORE ADVANCING PAGE
      018900      END-WRITE.
      019000
      019100 PAGE-START.
139 019200      WRITE PRINT-RECORD-2 FROM PAGE-HEAD
      019300      AFTER ADVANCING 1 LINE
      019400      END-WRITE.
140 019500      MOVE SPACES TO PRINT-RECORD-2.
141 019600      WRITE PRINT-RECORD-2 FROM COLUMN-HEAD
      019700      AFTER ADVANCING 1 LINE
      019800      END-WRITE.
142 019900      MOVE SPACES TO PRINT-RECORD-2.
      * * * * *  E N D   O F   S O U R C E   * * * * *

```

Figure 128. Example of an Indexed File Update Program (Part 4 of 4)

IBM i System Files

The IBM i system has four categories of files:

- Database files
- Device files
- DDM files
- Save files.

Database files allow information to be permanently stored on the system. A database file is subdivided into groups of records called members. There are two types of database files:

- A **physical file** is a file that contains data records (similar to disk files on other systems).
- A **logical file** is a database file through which data from one or more physical files can be accessed. The format and organization of this data is different from that of the data in the physical files. Each logical file can define a different access path (index) for the data in the physical files, and can exclude and reorder the fields defined in the physical files.

A database physical file can exist on one IBM i system or on multiple IBM i systems. If a database physical file exists on more than one IBM i system, it is called a **distributed physical file** or a **distributed file**. Since a logical file is based on one or more physical files, if the underlying physical file is distributed, then the logical file is also a distributed file.

To access a distributed file from an ILE COBOL program, you OPEN the distributed file; no other intermediate file is required, and no knowledge is required of the IBM i systems that have a part of the distributed file.

Contrast this to a Distributed Data Management (DDM) file which identifies the name of a database file that exists on a remote system. In ILE COBOL, to OPEN the remote database file, you actually open the local DDM file. Thus, a DDM file combines the characteristics of a device file and a database file. As a device file, the DDM file refers to a remote location name, local location name, device name, mode, and a remote network ID to identify a remote system as the target system. The DDM file appears to the application program as a database file and serves as the access device between the ILE COBOL program and a remote file.

Since a DDM file identifies a remote database file, and since database files can be distributed files, a DDM file can refer to a distributed file.

For more information about DDM files and distributed files, refer to the *DB2*
Universal Database for AS/400 section of the *Database and File Systems* category in the
i5/OS Information Center at this Web site - <http://www.ibm.com/systems/i/infocenter/>.
#

Distributed Data Management (DDM) Files

An ILE COBOL file assigned to a device of DISK or DATABASE can refer to a DDM file.

A DDM file is a file on the local (or source) system that contains the information needed to access a data file on a target system. It is not a data file that can be accessed by a program for database operations. Instead, when an ILE COBOL program running on a source system opens a DDM file, the file information is used by DDM to locate the remote file whose data is to be accessed.

A DDM file is created by the Create DDM File (CRTDDMF) command. The DDM file is stored as a file object in a library, the same as any other file or object.

When an ILE COBOL program opens a DDM file, a DDM conversation with the target system is established. And, if the program is opening the DDM file to access records in the remote file, an open data path (ODP) to the remote file is also established.

DDM can be used to communicate between systems that are architecturally different. For example, although the architectures of the IBM i system and System/36 are different, these systems can use DDM to access files in each other's database.

The following sections discuss the behavior that is unique to DDM files, and database files access through DDM files. Other topics about database files are discussed elsewhere in this chapter.

Using DDM Files with Non-IBM i Systems

If you are using DDM among System/38[®] or System/36[®] systems as well as IBM i systems, you should be aware that the concepts for both types are similar, except in the way they point to the remote file:

- An IBM i system and a System/38 use a separate DDM file to refer to each remote file to be accessed.

- A System/36 system uses a network resource directory that contains one network resource directory entry for each remote file to be accessed.

DDM Programming Considerations

```

#           Generally, DDM file names can be specified in ILE COBOL anywhere a database
#           file can be specified, for both i5/OS and non-i5/OS target systems. This section
#           summarizes the ILE COBOL programming considerations for DDM files:
#
#           • DDM file names can be specified on the Create Bound COBOL Program
#             (CRTBNDCBL), Create COBOL Module (CRTCBLMOD), and Create COBOL
#             Program (CRTCBLPGM) commands:
#
#             – To access remote files containing source statements, on an i5/OS or a
#               non-i5/OS system, a DDM file name can be specified on the SRCFILE
#               parameter, and a member name can be specified on the SRCMBR parameter.
#
#             – For IBM i or System/38 target systems, a remote IBM i or System/38 source
#               file (and, optionally, member) can be accessed in the same manner as a local
#               source file and member.
#
#             – For non-IBM i target systems, a remote source file can be accessed if both the
#               PGM and SRCMBR parameter defaults are used on the CRTBNDCBL or
#               CRTCBLPGM command. Or, if a member name is specified, it must be the
#               same as the DDM file name specified on the SRCFILE parameter. The
#               CRTCBLMOD command follows similar rules, except that the PGM and
#               SRCMBR parameters are replaced with the MODULE and SRCMBR
#               parameters.
#
#             – To place the compiler listing in a database file on a target system, a DDM file
#               name can be specified on the PRTFILE parameter of the CRTCBLPGM
#               command.
#
#           • DDM file names can be specified as the input and output files for the ILE
#             COBOL SORT and MERGE operation.
#
#           • A DDM file can be used in the ILE COBOL COPY statement when the DDS
#             option on that statement is used to copy one or all of the externally described
#             record formats from the remote file referred to by the DDM file into the program
#             being compiled. If this is done when the remote file is not on an IBM i system or
#             a System/38, the field declares for the record descriptions will not have
#             meaningful names. Instead, all of the field names are declared as Fnnnnnn and
#             the key fields are declared as Knnnnnn.
#
#           A recommended method for describing remote files, when the target is not an
#           IBM i system or a System/38, is to have the data description specifications
#           (DDS) on the local system and enter a Create Physical File (CRTPF) command or
#           a Create Logical File (CRTLFL) command on the local system. Compile the
#           program using the local file name. Ensure that the remote system's file has the
#           corresponding field types and field lengths. To access the remote file, use the
#           Override with Database File (OVRDBF) command preceding the program, for
#           example:
#
#           OVRDBF FILE(PGMFIL) TOFILE(DDMFIL) LVLCHK(*NO)
#
#           • DDM file names can be specified on a COPY statement:
#
#             – If you do not specify the library name with the file name, the first file found
#               with that file name in the user's library list is used as the include file.
#
#             – If the target system is not an IBM i system or a System/38, a DDM file name
#               can be specified as the include file on a COPY statement, but the member
#               name must be the same as the DDM file name.
#
#           • If the target system is a System/36, ILE COBOL cannot be used to open a DDM
#             file for output if the associated remote file has logical files built over it. For

```

```
# System/36 files with logical files, the open operation (open output) will fail
# because ILE COBOL programming language attempts to clear the file before
# using it.
# • When an ILE COBOL program opens a DDM file on the source system, the
# following statements can be used to perform I/O operations on the remote file
# at the target system, for both IBM i and non-IBM i targets: CLOSE, DELETE,
# OPEN, READ, REWRITE, START, and WRITE.
```

DDM Direct (Relative) File Support

```
# An IBM i system does not support direct files as one of its file types. (An IBM i
# system creates direct files as sequential files.) However, an ILE COBOL program on
# an IBM i system can specify that a file be accessed as a direct file by specifying an
# organization of RELATIVE on the SELECT statement.
```

Keep the following in mind when working with direct files on the i5/OS:

```
# • If a file is created on the local IBM i system as a direct file by a program or user
# from a non-IBM i system, the file can be accessed as a direct file by a ILE
# COBOL program from a remote non-IBM i source system.
# • If a file is created on the local IBM i system by a program or user on the same
# IBM i system, it cannot be accessed as a direct file by a non-i5/OS because the
# IBM i target system cannot determine, in this case, whether the file is a direct or
# sequential file.
# • Any files created by a remote system can be used locally.
```

Distributed Files

Distributed files allow a database file to be spread across multiple IBM i systems, while retaining the look and capability of a single database. Performance of large queries can be enhanced by splitting database requests across multiple systems. Distributed files behave in much the same way as DATABASE files. However, since files are distributed across multiple systems, the arrival sequence or relative number cannot be relied upon, and additional time is required for the data link to pass the data between the systems whenever the remote system is accessed.

A distributed file is created like other database files, with the Create Physical File (CRTPF) command. This command has two new parameters that relate to a distributed file:

- Node group (NODGRP)
- Partitioning key (PTNKEY).

The first parameter has a value of *NONE for regular files, and the name of a node group for a distributed file. A node group is a new system object type (type *NODGRP) that specifies the names of the relational databases that will contain the records of the file. A node group is created with the Create Node Group (CRTNODGRP) command.

The records of a distributed file are divided amongst the various relational databases based on a partitioning key. The partitioning key is a field, or set of fields, from the distributed file whose value will determine in which relational database each record will be stored.

An existing physical file can be changed into a distributed file by using the Change Physical File (CHGPF) command. The two new parameters, node group and partitioning key, that were added to the CRTPF command were also added to the CHGPF command.

Open Considerations for Data Processing

A distributed file's data can be accessed in a buffered or non-buffered way. This buffering of records is in addition to other buffering, like SEQONLY processing.

The Override with Database File (OVRDBF) command has a new parameter called distributed data (DSTDTA) that has three values:

***BUFFERED**

Data may be kept in a buffer.

***PROTECTED**

Similar to *BUFFERED, but the file is locked to prevent updates by other jobs.

***CURRENT**

Data is not buffered.

The remainder of this section describes the open considerations when distributed data processing is overridden and when it is not overridden.

When Distributed Data Processing is Overridden

The following considerations apply for opening distributed files when distributed data processing *is* overridden:

- If the distributed file open operation will be for input-only processing, and records that are deleted, inserted, or updated while the file is open must be processed immediately by your program, then you must use the OVRDBF (Override with Data Base File) command to override the distributed file processing to non-buffered retrieval (*CURRENT). Non-buffered retrieval does not achieve the same performance as buffered retrieval, but it will guarantee data integrity, and provide for maximum record concurrency while you have the distributed file open.
- If the file open operation will be for update or delete operations, then you may want to use the OVRDBF (Override with Data Base File) command to override the distributed data processing to either protected buffer retrieval (*PROTECTED) or buffered retrieval (*BUFFERED).

These are the advantages and disadvantages to protected buffer retrieval:

- Achieves the same performance as buffered retrieval.
- Guarantees data integrity if your program does not delete, insert, or update records.
- Will not be allowed if another process has the distributed file open for anything other than input-only processing, which does not also include protected buffer retrieval.

These are the advantages and disadvantages to buffered retrieval:

- Achieves the same performance as protected buffer retrieval.
- Allows for maximum record concurrency while you have the distributed file open.

- Records that are deleted, inserted, or updated in the distributed file after the open might not be seen as they occur. This may cause your program to update or delete the wrong record.

When Distributed Data Processing is NOT Overridden

The following considerations apply for opening distributed files when distributed data processing is *not* overridden:

- The system will process a distributed file that is open for input-only using buffered retrieval (*BUFFERED). Buffered retrieval will achieve the best performance along with maximum record concurrency, however, you might not see all of the changes made to the file as they occur. Refer to “Input/Output Considerations for Distributed Files” for more information.
- The system will process a distributed file that is opened for output-only one record at a time. If your distributed file is opened for output-only, the DSTDTA parameter will have no effect. Also, if SEQONLY(*YES) processing has been requested, it will be changed to SEQONLY(*NO). The SEQONLY(*NO) processing will provide feedback on a record-by-record basis when the records are inserted into the file.
- The system will process a distributed file that has been opened with an option that includes update or delete using non-buffered retrieval (*CURRENT). Non-buffered retrieval ensures that you are updating or deleting the same record that would have been updated or deleted if all of the distributed file data had been contained in a non-distributed database file. Since non-buffered retrieval will be used, the best performance for the distributed file will not be achieved, but the best data integrity and the maximum record concurrency will be guaranteed.

Note: For arrival sequence distributed files, records will be retrieved in arrival sequence starting with the first node, then the second node, and so on. For duplicate key considerations, refer to “Input/Output Considerations for Distributed Files.”

- The system will process a distributed file that is opened with all operations (*INP, *OUT, *UPD, *DLT) using non-buffered retrieval (*CURRENT), since it includes both update and delete options.

Input/Output Considerations for Distributed Files

The following considerations apply to input/output operations for distributed files:

- For input of arrival sequence distributed files and keyed sequence distributed files whose keyed access paths have been ignored at open time, the records will be retrieved as follows:
 1. All records from the first node, as defined by the node group at file creation time, will be retrieved in arrival sequence from the first node.
 2. After all records from the first node have been retrieved, then all records from the second node will be retrieved in arrival sequence from the second node.
 3. After all records from the second node have been retrieved, then all records from the third node will be retrieved in arrival sequence from the third node.
 4. This will continue until the last node defined by the node group at file creation time is reached.
 5. After all records from the last node have been retrieved in arrival sequence, end-of-file is reached.

Thus, distributed files that are processed in arrival sequence will not be processed in arrival sequence across the different nodes of the distributed file.

- For input of keyed sequence distributed files whose keyed access paths have not been ignored at open time, the records are retrieved as follows:
 - The first-changed first-out (FCFO), first-in first-out (FIFO), or last-in first-out (LIFO) order of records with duplicate key values will only be valid for records that come from the same node.
 - All records with duplicate key values from the first node as defined by the node group at file creation time will be retrieved in the specified access path order.
 - After all records with duplicate key values from the first node have been retrieved, then all records with duplicate key values from the second node will be retrieved in the specified access path order.
 - After all records with duplicate key values from the second node have been retrieved, then all records with duplicate key values from the third node will be retrieved in the specified access path order.
 - This will continue until the last node as defined by the node group at the file creation time is reached.
 - After all records with duplicate key values have been retrieved from the last node in the specified access path order, the next non-duplicate key value will be retrieved.

Therefore, distributed files that have duplicate key values will not be processed in the specified access path order across the different nodes of the distributed file.

- When buffered retrieval (*BUFFERED) or protected buffered retrieval (*PROTECTED) is being used:
 - Records that are inserted or updated in the distributed file after the open might not be seen while retrieving records even if their key values come after the last record returned to your program. This is because each node has its own key position based on the last get-by-key request. “Example of How Records are Retrieved for Insert, Update, and Delete” provides an example of how duplicate key records are retrieved for insert or update.
 - Records that are deleted from the distributed file after the open might still be seen while retrieving records from the file.
 - The only difference between buffered retrieval and protected buffer retrieval is that protected buffer retrieval restricts the deleting, inserting, and updating of records in the distributed file to your job.
- For output to distributed files, the system will process insert requests one record at a time. If your distributed file open request is for output-only and SEQONLY(*YES) processing, it will be changed to SEQONLY(*NO). The single record output processing will provide feedback on a record-by-record basis when the records are inserted into the file.

Example of How Records are Retrieved for Insert, Update, and Delete

Figure 129 on page 513 shows the different record positions for a distributed file after the first get-by-key request in buffered retrieval. This get-by-key request has positioned the distributed file at the first record on each node.

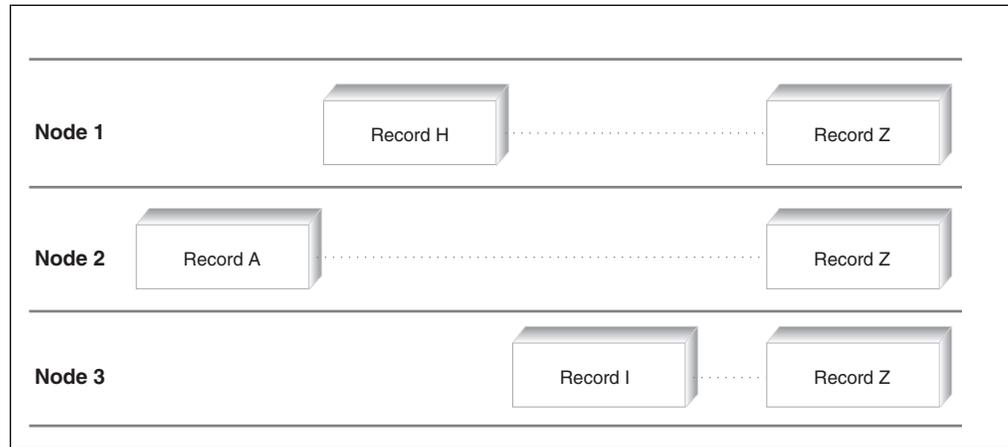


Figure 129. First Duplicate Record Key Positions Across Nodes in a Distributed File

In this example, the first get-by-key request has returned record A to your program. Because of the different record positions on the different nodes, subsequent get-by-key-next requests would not return records that had been inserted or updated on node 1 that preceded either Record H on Node 1 or Record I on Node 3. An inserted or updated record that comes after the last record returned to your program, but before the current key position for a particular node, will not be seen by your program unless the direction in which you are reading records is changed.

Records that have been deleted may also be seen by your program if they have already been positioned to and retrieved from a particular node. For example, if Record A from Node 2 has been returned to your program, Record I from Node 3 will be returned to your program even if it has been deleted prior to issuing the next get-by-key-next request set to retrieve it.

When non-buffered retrieval (*CURRENT) is being used, records that are inserted or updated in the distributed file after the open will be retrieved in the same way as they would have been for a non-distributed database file, except for duplicate key values that span nodes. Records that are inserted or updated in a distributed file after it has been opened for non-buffered retrieval also might not be seen if its key value comes before the last record that has been returned to your program. If you require that the keyed sequence input to your distributed file retrieves the same records that would have been retrieved for a non-distributed database file, except for duplicate key values that span nodes, then you should override the open of your keyed distributed file to non-buffered retrieval.

SQL Statement Additions for Distributed Data Files

New clauses have been added to the SQL CREATE TABLE statement to allow you to create distributed files with SQL. These additions are shown below:

```

>> CREATE TABLE (—column definitions—)
|
| IN {
|   —nodegroup—
|   —library—/—nodegroup—
|   —library—.—nodegroup—
| } partition key |

```

partition key:



For more information about using SQL commands in ILE COBOL programs, refer to "Including Structured Query Language (SQL) Statements in Your ILE COBOL Program".

Examples of Processing Distributed Files

In order to create a distributed file, you must do the following on each system on which parts of the distributed file will exist:

1. You need to add a relational database directory entry for the local system, and one relational database directory entry for every other system that is going to contain part of the file
2. You have to create the library that will contain the distributed file.

For the primary system, you need to do the following:

1. Create a node group which contains the names of all of the relational databases involved
2. Define the DDS for the physical file
3. Create the physical file specifying the Node Group (NODGRP) and Partitioning Key (PTNKEY) parameters
4. If you create a logical file over the distributed physical file, a distributed logical file results.

For example, suppose you have two systems, and you want each one to contain part of a distributed file. Assume:

- Your primary system is called OS400SYS1, and your other system is OS400SYS2
- The library where the distributed file will exist is DISTRIBUTE.

To create the relational database directory entries on system OS400SYS1, you would enter the following commands:

```
ADDRDBDIRE RDB(OS400SYS1) RMTLOCNAME(*LOCAL)
            TEXT('local database RDB directory entry')
ADDRDBDIRE RDB(OS400SYS2) RMTLOCNAME(AS400SYS2)
            TEXT('remote database RDB directory entry')
```

To create the library DISTRIBUTE on OS400SYS1, enter the CRTLIB command.

To create the relational database directory entries on system OS400SYS2, you would enter the following commands:

```
ADDRDBDIRE RDB(OS400SYS2) RMTLOCNAME(*LOCAL)
            TEXT('local database RDB directory entry')
ADDRDBDIRE RDB(OS400SYS1) RMTLOCNAME(AS400SYS1)
            TEXT('remote database RDB directory entry')
```

To create the library DISTRIBUTE on OS400SYS2, enter the CRTLIB command.

On your primary system, assume:

- The name of the node group, which names the relational databases that will contain the records for the distributed file, is NODEGROUP

- The name of the distributed physical file is CUSTMAST.

Then, to create the node group on system OS400SYS1, use the following command:

```
CRTNODGRP NODGRP(DISTRIBUTE/NODEGROUP) RDB(OS400SYS1 AS400SYS2)
      TEXT('node group for distributed file')
```

The DDS for the Create Physical File (CRTPF) command (contained in source file QDDSSRC, in library DISTRIBUTE) is:

.....+.....1.....+.....2.....+.....3.....+.....4.....+.....5.....+.....6.....+.....7.....+.....8				
A	R	CUSTREC		
A		CUSTOMERNO	9S 0	ALIAS(CUSTOMER_NUMBER)
A		FIRSTNAME	15A	ALIAS(CUSTOMER_FIRST_NAME)
A		LASTNAME	15A	ALIAS(CUSTOMER_LAST_NAME)
A		ADDRESS	20A	ALIAS(CUSTOMER_ADDRESS)
A		ACCOUNTNO	9S 0	ALIAS(CUSTOMER_ACCOUNT_NUMBER)

The DDS field CUSTOMERNO is used below as the partitioning key for the distributed file.

```
CRTPF FILE(DISTRIBUTE/CUSTMAST)
      SRCFILE(DISTRIBUTE/QDDSSRC) SRCMBR(CUSTMAST)
      NODGRP(DISTRIBUTE/NODEGROUP)
      PTNKEY(CUSTOMERNO)
```

When the Create Physical File (CRTPF) command completes on the primary system, the file is created on the primary system as well as on all of the other relational databases in the node group. After the file has been created, changes to the node group will no longer affect the distributed file.

Processing Files with Constraints

Data within the fields of a database physical file (SQL TABLE) can be restricted to certain values by adding a constraint relationship. There are four types of constraints:

- Referential
- Unique
- Primary key (a special case of a unique constraint)
- Check.

You can use constraint relationships to define dependencies between files. The relationships that you define are enforced by the system when changes occur to information in the files. When you define constraint relationships, you control the **referential integrity** of the data being processed.

Check constraints are validity checks that can be placed on fields (columns) in the database physical files (SQL tables), thereby increasing the integrity of your data.

When data is inserted or updated in fields with constraints, the data must first meet the validity checks placed on those fields, before the insert or update operation can be completed. If all of the constraints are not met, then the I/O request is not performed, and a message is sent back to the program indicating that a constraint has been violated. When a check constraint has been violated during the running of a COBOL I/O statement, a file status of 9W is set. If a referential constraint is violated, a file status of 9R is set.

Although only physical files can have constraints, the constraint is enforced while performing I/O on a logical file built over a physical file with constraints. Check constraints can be used for one or many fields, and can be used with field-to-field-comparisons or field-to-literal comparisons.

For more detailed information about constraints, refer to the *DB2 Universal*
Database for AS/400 section of the *Database and File Systems* category in the **i5/OS**
Information Center at this Web site - [http://www.ibm.com/systems/i5](http://www.ibm.com/systems/i5infocenter/)
[infocenter/](http://www.ibm.com/systems/i5infocenter/).

Restrictions

The following restrictions apply when adding constraints to a file or table. The file:

- Must be a database physical file
- Can have a maximum of one member
- Cannot be a program-described file
- Cannot be a source file
- Cannot reside in QTEMP
- Cannot be opened
- Cannot have uncommitted I/O changes.

Referential and check constraints have four states:

- Defined and enabled
- Defined and disabled
- Established and enabled
- Established and disabled.

Defined means the constraint definition has been added to the file, but not all of the pieces of the file are there for enforcement. For example, the file's member does not exist.

Established means the constraint definition has been added to the file, and all of the pieces of the file are there for enforcement.

Enabled means the check constraint will be enforced if the constraint is also established. If the constraint is defined, then the file member structures do not yet exist for enforcement.

Disabled means the constraint definition will not be enforced, regardless of whether the constraint is established or defined.

To define or establish a referential constraint, the parent file and the dependent file must exist. However, if the parent or dependent file has no members, the constraint is defined only (not established).

Adding, Modifying and Removing Constraints

Constraints can be added, modified, or removed using:

- SQL
- CL commands.

Through SQL, a constraint can be added to the column of a table using the CREATE TABLE statement. If the table already exists, then the ALTER TABLE statement can be used to add the constraint. The ALTER TABLE statement can also be used to DROP the constraint.

Using CL commands, the Add PF Constraint (ADDPFCST) command can be used to add or change a constraint, and the Remove PF Constraint (RMVPFCST) command can be used to remove a constraint.

Checking that Constraints Have Been Successfully Added or Removed

From an ILE COBOL program, the Retrieve File Description (QDBRTVFD) API can be used. Externally, from the operating system, the Display File Description (DSPFD) command can be used. A query of the system cross-reference file (QADBFCSST) will also show if a constraint has been added to a file.

Both the Retrieve File Description (QDBRTVFD) API and the Display File Description (DSPFD) command retrieve the file definition along with all of the constraints that have been added.

Order of Operations

The following is the order of operations for a file on which commitment control has *not* been started:

- BEFORE trigger fired
- Referential constraint processed for *RESTRICT
- Check constraint processed
- I/O operation processed
- AFTER trigger fired
- Referential constraints, other than *RESTRICT, processed.

The following is the order of operations for a file on which commitment control *has* been started:

- BEFORE trigger fired
- Referential constraint processed for *RESTRICT
- I/O operation processed
- AFTER trigger fired
- Referential constraints, other than *RESTRICT, processed
- Check constraint processed.

Handling Null Fields with Check Constraints

If a field is null-capable and used in a check constraint, then depending on the field's value, the constraint may, or may not be, affected:

- If a field (column) value in a record (row) is *not* null, then the field is used in the validation process of the check constraint to return either a status of valid or check pending.
- If the field (column) is *null*, then the field (column) value is *not* used to validate the constraint, unless the check constraint specifically tests for a null value. This means that the affect a null field will have on a check constraint is unknown.

Handling Constraint Violations

Constraints can have a status of **check pending**. A status of check pending means that the data in the record (row) violates a constraint. When a COBOL I/O statement is run, the system will ensure that a record cannot be inserted or updated that would cause a constraint to be violated. Any attempt to do so, will result in file status 9W (check constraint failure) or file status 9R (referential constraint failure). However, adding constraints where data already exists or restoring old data can cause constraint violations, and, thereby, statuses of check pending.

Once an established and enabled check constraint has been violated (has a status of check pending), data cannot be read from the file. For those insert, update, or delete operations that require a read for update, the I/O operation will not be performed. Otherwise, insert, update, and delete operations will be performed. In order to read from the file again after a check constraint has been violated, the check constraint has to be disabled using the Change PF Constraint (CHGPFCST) command.

Once an established and enabled referential constraint has a status of check pending:

- No file I/O is allowed against the dependent file
- Limited file I/O (READ/INSERT) is allowed against the parent file

To figure out what is causing the constraint violation, after the constraint has been disabled, you can use one of the following methods:

- Use the Display CHKPNP Constraint (DSPCPCST) command to check which records are causing the violation.
- Use the Work with PF Constraints (WRKPFPCST) command to find out which constraint is in check pending.
- Use the Remove PF Constraint (RMVPCST) command to remove the constraint, followed by the Add PF Constraint (ADDPFCST) command to add the constraint back on. This will list the first 20 records of the constraint that is causing the violation.

Database Features that Support Referential or Check Constraints

The following database features support referential and check constraints:

- Journaling
- Commitment control
- Distributed Data Management (DDM) files
- Distributed (multi-system) files.

Journaling

A file with referential or check constraints can be journaled, but it is not required to be. There are not any special journal entries associated with check constraints.

Commitment Control

When commitment control is active, file I/O functions follow the same rules that apply when commitment control is not active. That is, when performing I/O on a file with constraints, an insert, update, or delete is not allowed where a constraint rule would be violated. Potential violations result in notification messages. If the I/O operation completes successfully, then either a COMMIT or ROLLBACK can be performed.

Distributed Data Management (DDM)

Check constraints are supported for Distributed Data Management (DDM) files. When DDM is being used between a V4R2 and a pre-V4R2 system, then any check constraint information that may exist on the V4R2 system is not passed to the pre-V4R2 system.

When an attempt is made to propagate check constraints between a V4R2 and a pre-V4R2 system for DDM files, the following operations will either not propagate the check constraints or will fail:

- A create file or create table operation will work, but will not propagate check constraints
- An extract file definition operation will work, but will not propagate check constraints
- An ALTER TABLE statement will fail
- A Change Physical File (CHGPF) CL command will fail.

Distributed Files

Check constraints are supported for Distributed (multi-system) files. When distributed files are being used between a V4R2 and a pre-V4R2 system, then any check constraint information that may exist on the V4R2 system is not passed to the pre-V4R2 system.

When an attempt is made to propagate check constraints between a V4R2 and a pre-V4R2 system for distributed files, the following operations will fail:

- Create file or create table operation
- The Add PF Constraint (ADDPFCST) CL command
- ALTER TABLE statement
- Change Physical File (CHGPF) CL command.

Chapter 21. Using Transaction Files

This chapter describes the ILE COBOL language extensions that support workstations and program-to-program communication.

The TRANSACTION file organization allows an ILE COBOL program to communicate interactively with:

- One or more workstations
- One or more programs on a remote system
- One or more devices on a remote system.

The AS/400 system permits you to communicate with a program or device (such as Asynchronous communication types) on a remote system. For a detailed discussion of these devices, see the *ICF Programming* manual

ILE COBOL TRANSACTION files are usually externally described. If these files are
program-described, only simple display formatting can be performed. For more
information about using program-described display files, refer to the *Database and*
File Systems category in the **i5/OS Information Center** at this Web site
-http://www.ibm.com/systems/i/infocenter/.

An ILE COBOL TRANSACTION file usually uses an externally described file that contains file information and a description of the fields in the records. The records in this file can be described in an ILE COBOL program by using the Format 2 COPY statement. Refer to the *IBM Rational Development Studio for i: ILE COBOL Reference* for more information about the Format 2 COPY statement.

| Do not send packed, binary, or float data (COMP, COMP-1, COMP-2, COMP-3,
| COMP-4 or COMP-5) to a display station as output data. Such data can contain
| display station control characters that can cause unpredictable results.

Defining Transaction Files Using Data Description Specifications

You use data description specifications (DDS) to describe an externally described TRANSACTION file.

In addition to the field descriptions (such as field names and attributes), the data description specifications (DDS) for a display device file do the following:

- Specify the line number and position number entries for each field and constant to format the placement of the record on the display.
- Specify attention functions such as underlining and highlighting fields, reverse image, or a blinking cursor.
- Specify validity checking for data entered at the display workstation.
- Control display management functions such as when fields are to be erased, overlaid, or retained when new data is displayed.
- Associate indicators 01 through 99 with function keys designated as type CA or CF. If a function key is designated as CF, both the modified data record and the response indicator are returned to the program. If a function key is designated as CA, the response indicator is returned to the program, but the data record usually contains default values for input-only fields and values written to the format for hidden output/input fields. For more information about type CF and

#

CA function keys, see refer to the *Database and File Systems* category in the **i5/OS Information Center** at this Web site -<http://www.ibm.com/systems/i/infocenter/>.

- Assign an edit code (EDTCDE keyword) or edit word (EDTWRD keyword) to a field to specify how the field's values are to be displayed.
- Specify subfiles.

Display format data defines or describes a display. A display device record format contains three types of fields:

- *Input Fields:* Input fields pass 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 passes to the program. Un-initialized input fields are displayed as blanks where the work station user can enter data.
- *Output Fields:* Output fields pass from the program to the device when the program writes a record to a display. The program or the record format in the device file can provide output fields.
- *Output/Input (Both) Fields:* An output/input field is an output field that can be changed to become an input field. Output/input fields pass *from* the program when the program writes a record to a display and pass *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.

For a detailed description of a data communications file, see the *ICF Programming* manual. For more information on externally defined display files and a list of the valid data description specifications (DDS) keywords, refer to the *Database and File Systems* category in the **i5/OS Information Center** at this Web site -<http://www.ibm.com/systems/i/infocenter/>.

Figure 130 shows an example of the DDS for a display device file:

```

.....+.....1.....+.....2.....+.....3.....+.....4.....+.....5.....+.....6.....+.....7.....+.....8
A* CUSTOMER MASTER INQUIRY FILE ** CUSMINQ
A*
A          REF(CUSMSTP) 1
A          R CUSPMT      TEXT('CUSTOMER PROMPT')
A          CA01(15 'END OF PROGRAM') 2
A          1 3 'CUSTOMER MASTER INQUIRY'
A          3 3 'CUSTOMER NUMBER'
A          CUST        R      I 3 20
A 99          ERRMSG('CUSTOMER NUMBER NOT FOUND + 3
A          PRESS RESET, THEN ENTER A VALID NU+
A          MBER' 99)
A          5 3 'USE CF1 TO END PROGRAM, USE ENTE+
A          R TO RETURN TO PROMPT SCREEN'
A          R CUSFLDS    TEXT('CUSTOMER DISPLAY')
A          CA01(15 'END OF PROGRAM')
A          OVERLAY 4
A          8 3 'NAME'
A          NAME        R      8 11
A          9 3 'ADDRESS'
A          ADDR        R      9 11
A          10 3 'CITY' 5
A          CITY        R      10 11
A          11 3 'STATE' 6
A          STATE       R      11 11
A          11 21 'ZIP CODE'
A          ZIP         R      11 31
A          12 3 'A/R BALANCE'
A          ARBAL       R      12 17

```

Figure 130. Example of the Data Description Specifications for a Display Device File

This display device file contains two record formats: CUSPMT and CUSFLDS.

- 1 The attributes for the fields in this file are defined in the CUSMSTP field reference file. For example, EDTCDE(J) is defined in CUSMSTP for the field ARBAL.
- 2 The F1 key is associated with indicator 15, with which the user ends the program.
- 3 The ERRMSG keyword identifies the error message that is displayed if indicator 99 is set on in the program that uses this record format.
- 4 The OVERLAY keyword is used for the record format CUSFLDS so that the CUSPMT record on the display will not be erased when the CUSFLDS record is written to the display.
- 5 The constants such as 'Name', 'Address', and 'City' describe the fields that are written out by the program.
- 6 The line and position entries identify where the fields or constants are written on the display.

Processing an Externally Described Transaction File

When an externally described TRANSACTION file is processed, the operating system transforms data from your ILE COBOL program to the format specified for the file and displays the data. When data passes to your ILE COBOL program, the data is transformed to the format used by your ILE COBOL program.

The operating system provides device control information for performing input/output operations for the device. When an input record is requested from the device by your ILE COBOL program, the operating system issues the request,

and then removes device control information from the data before passing the data to the program. In addition, the operating system can pass indicators to your ILE COBOL program indicating which, if any, fields in the record have changed.

When your ILE COBOL program requests an output operation, it passes the output record to the operating system. The operating 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 passes to your ILE COBOL 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 positions) assigned to the fields in the DDS. Therefore, the order in which the fields are specified in the DDS and the order in which they appear on the display need not be the same.

Writing Programs That Use Transaction Files

Typically, you use a TRANSACTION file to read one record from or write one record to a display. To use a TRANSACTION file in an ILE COBOL program, you must:

- Name the file through a file control entry in the FILE-CONTROL paragraph of the Environment Division
- Describe the file through a file description entry in the Data Division
- Use extensions to Procedure Division statements that support transaction processing.

Note: Using extended ACCEPT/DISPLAY statements and TRANSACTION files in the same program is not recommended. If extended ACCEPT/DISPLAY statements are used in the same program as TRANSACTION files, then the TRANSACTION file should be closed when the extended ACCEPT/DISPLAY statements are performed. Unpredictable results will occur if an extended ACCEPT/DISPLAY statement is performed when a TRANSACTION file is open. A severe error may be generated or data on the workstation may be overlapped or intermixed.

Naming a Transaction File

To use a TRANSACTION file in an ILE COBOL program, you must name the file through a file control entry in the FILE-CONTROL paragraph. See the *IBM Rational Development Studio for i: ILE COBOL Reference* for a full description of the FILE-CONTROL paragraph.

You name the TRANSACTION file in the FILE-CONTROL paragraph as follows:
FILE-CONTROL.

```
    SELECT transaction-file-name
      ASSIGN TO WORKSTATION-display_file_name
      ORGANIZATION IS TRANSACTION
      ACCESS MODE IS SEQUENTIAL
      CONTROL AREA IS control-area-data-item.
```

You use the SELECT clause to choose a file. This file must be identified by a FD entry in the Data Division.

You use the ASSIGN clause to associate the TRANSACTION file with a display file or ICF file. You must specify a device type of WORKSTATION in the ASSIGN clause to use TRANSACTION files. If you want to use a separate indicator area for

this TRANSACTION file, you need to include the -SI attribute with the ASSIGN clause. See “Using Indicators with Transaction Files” on page 536 for further details of how to use the separate indicator area.

You must specify ORGANIZATION IS TRANSACTION in the file control entry in order to use a TRANSACTION file. This clause tells your ILE COBOL program that it will be interacting with a workstation user or another system.

You access a TRANSACTION file sequentially. You use the ACCESS MODE clause in the file control entry to tell your ILE COBOL program how to access the TRANSACTION file. You specify ACCESS MODE IS SEQUENTIAL to read or write to the TRANSACTION file in sequential order. If you do not specify the ACCESS MODE clause, sequential access is assumed.

If you want feedback on the status of an input-output request that refers to a TRANSACTION file, you define a status key data item in the file control entry using the FILE STATUS clause. When you specify the FILE STATUS clause, the system moves a value into the status key data item after each input-output request that explicitly or implicitly refers to the TRANSACTION file. The value indicates the status of the execution of the I-O statement.

You can obtain specific device-dependent and system dependent information that is used to control input-output operations for TRANSACTION files by identifying a control area data item using the CONTROL-AREA clause. You can define the data item specified by the CONTROL-AREA clause in the LINKAGE SECTION or WORKING-STORAGE SECTION with the following format:

```
01 control-area-data-item.  
   05 function-key          PIC X(2).  
   05 device-name          PIC X(10).  
   05 record-format        PIC X(10).
```

The control area can be 2, 12, or 22 bytes long. Thus, you can specify only the first 05-level element, the first two 05-level elements, or all three 05-level elements, depending of the type of information your are looking for.

The control area data item will allow you to identify:

- The function key that the operator pressed to initiate a transaction
- The name of the program device used
- The name of the DDS record format that was referenced by the last I-O statement.

Describing a Transaction File

To use a TRANSACTION file in an ILE COBOL program, you must describe the file through a file description entry in the Data Division. See *IBM Rational Development Studio for i: ILE COBOL Reference* for a full description of the File Description Entry. Use the Format 6 File Description Entry to describe a TRANSACTION file.

A file description entry in the Data Division that describes a TRANSACTION file looks as follows:

```
FD CUST-DISPLAY.  
01 DISP-REC.  
   COPY DDS-ALL-FORMATS OF CUSMINQ.
```

In ILE COBOL, TRANSACTION files are usually externally described. Create a DDS for the TRANSACTION file you want to use. Refer to “Defining Transaction Files Using Data Description Specifications” on page 521 for how to create a DDS. Then create the TRANSACTION file.

Once you have created the DDS for the TRANSACTION file and the TRANSACTION file, use the Format 2 COPY statement to describe the layout of the TRANSACTION file data record. When you compile your ILE COBOL program, the Format 2 COPY will create the Data Division statements to describe the TRANSACTION file. Use the DDS-ALL-FORMATS option of the Format 2 COPY statement to generate one storage area for all formats.

Processing a Transaction File

The following is a list of all of the Procedure Division statements that have extensions specifically for processing TRANSACTION files in an ILE COBOL program. See the *IBM Rational Development Studio for i: ILE COBOL Reference* for a detailed discussion of each of these statements.

- ACCEPT Statement - Format 6
- ACQUIRE Statement
- CLOSE Statement - Format 1
- DROP Statement
- OPEN Statement - Format 3
- READ Statement - Format 4 (Nonsubfile)
- WRITE Statement - Format 4 (Nonsubfile).

Opening a Transaction File

To process a TRANSACTION file in the Procedure Division, you must first open the file. You use the Format 3 OPEN statement to open a TRANSACTION file. A TRANSACTION file must be opened in I-O mode.

```
OPEN I-O file-name.
```

Acquiring Program Devices

You must acquire a program device for the TRANSACTION file. Once acquired, the program device is available for input and output operations. You can acquire a program device implicitly or explicitly.

You implicitly acquire one program device when you open the TRANSACTION file. If the file is an ICF file, the single implicitly acquired program device is determined by the ACQPGMDEV parameter of the CRTICFF command. If the file is a display file, the single implicitly acquired program device is determined by the first entry in the DEV parameter of the CRTDSPF command. Additional program devices must be explicitly acquired.

You explicitly acquire a program device by using the ACQUIRE statement. For an ICF file, the device must have been defined to the file with the ADDICFDEVE or OVRICFDEVE CL command before the file was opened. For display files, there is no such requirement. That is, the device named in the ACQUIRE statement does not have to be specified in the DEV parameter of the CRTDSPF command, CHGDSPF command, or the OVRDSPF command. However, when you create the display file, you must specify the number of devices that may be acquired (the default is one). For a display file, the program device name must match the display device.

```
ACQUIRE program-device-name FOR transaction-file-name.
```

Writing to a Transaction File

Once you have opened the TRANSACTION file and acquired a program device for it, you are now ready to perform input and output operations on it.

The first input/output operation you typically perform on a TRANSACTION file is to write a record to the display. This record is used to prompt the user to enter a response or some data.

You use the Format 4 WRITE statement to write a logical record to the TRANSACTION file. You simply code the WRITE statement as follows:

```
WRITE record-name FORMAT IS format-name.
```

In some situations, you may have multiple data records, each with a different format, that you want active for a TRANSACTION file. In this case, you must use the FORMAT phrase of the Format 4 WRITE statement to specify the format of the output data record you want to write to the TRANSACTION file.

If you have explicitly acquired multiple program devices for the TRANSACTION file, you must use the TERMINAL phrase of the Format 4 WRITE statement to specify the program device to which you want the output record to be sent.

You can control the line number on the display where the WRITE statement will write the output record by specifying the STARTING phrase and ROLLING phrase of the Format 4 WRITE statement. The STARTING phrase specifies the starting line number for the record formats that use the variable record start line keyword. The ROLLING phrase allows you to move lines displayed on the workstation screen. All or some of the lines on the screen can be rolled up or down.

```
WRITE record-name FORMAT IS format-name
      TERMINAL IS program-device-name
      STARTING AT LINE start-line-no
      AFTER ROLLING LINES first-line-no THRU last-line-no
      DOWN no-of-lines LINES
END-WRITE.
```

Reading from a Transaction File

You use the Format 4 READ statement to read a logical record from the TRANSACTION file. If data is available when the READ statement is executed, it is returned in the record area. The names of the record format and the program device are returned in the I-O-FEEDBACK area and in the CONTROL-AREA area.

Before you use the READ statement, you must have acquired at least one program device for the TRANSACTION file. If a READ statement is performed and there are no acquired program devices, a logic error is reported by setting the file status to 92.

You can use the READ statement in its simplest form as follows:

```
READ record-name RECORD.
```

If you have only acquired one program device, this simple form of the READ statement will always wait for data to be made available. Even if the job receives a controlled cancellation, or a wait time is specified in the WAITRCD parameter for the display file or ICF file, the program will never regain control from the READ statement.

If you have acquired multiple program devices, this simple form of the READ statement will receive data from the first invited program device that has data

available. When multiple program devices have been acquired, this simple form of the READ statement can complete without returning any data if there are no invited devices and a wait time is not specified, a controlled cancellation of the job occurs, or the specified wait time expires.

For a detailed explanation of how the READ operation is performed, refer to the section on the READ statement in the *IBM Rational Development Studio for i: ILE COBOL Reference*.

In those cases where you have acquired multiple program devices, you can explicitly specify the program device from which you read data by identifying it in the TERMINAL phrase of the READ statement.

In those cases where you want to receive the data in a specific format, you can identify this format in the FORMAT phrase of the READ statement. If the data available does not match the requested record format, a file status of 9K is set.

The following are examples of the READ statement with the TERMINAL and FORMAT phrases specified.

```
READ record-name RECORD
    FORMAT IS record-format
END-READ
READ record-name RECORD
    TERMINAL IS program-device-name
END-READ
READ record-name RECORD
    FORMAT IS record-format
    TERMINAL IS program-device-name
END-READ
```

When the READ statement is performed, the following conditions can arise:

1. Data is immediately available and the AT END condition does not exist. The AT END condition occurs when there are no invited program devices and a wait time is not specified.
2. Data is not immediately available.
3. The AT END condition exists.

You can transfer control to various statements of your ILE COBOL program from the READ statement based on the condition that results from performing the READ statement by specifying the NO DATA phrase, AT END phrase, or NOT AT END phrase.

To perform a group of statements when the READ statement completes successfully, specify the NOT AT END phrase of the READ statement.

To perform a group of statements when the data is not immediately available, specify the NO DATA phrase of the READ statement. The NO DATA phrase prevents the READ statement from waiting for data to become available.

To perform a group of statements when the AT END condition exists, specify the AT END phrase of the READ statement.

The following are examples of the READ statement with the NO DATA, NOT AT END, and AT END phrases specified.

```
READ record-name RECORD
    TERMINAL IS program-device-name
    NO DATA imperative-statement-1
```

```
END-READ
READ record-name RECORD
    TERMINAL IS program-device-name
    AT END imperative-statement-2
    NOT AT END imperative-statement-3
END-READ
```

Dropping Program Devices

Once you have finished using a program device that you had previously acquired for a TRANSACTION file, you should drop it. Dropping a program device means that it will no longer be available for input or output operations through the TRANSACTION file. Dropping a program device makes it available to other applications. You can drop a program device implicitly or explicitly.

You implicitly drop all program devices attached to a TRANSACTION file when you close the file.

You explicitly drop a program device by indicating it in the DROP statement. The device, once dropped, can be re-acquired again, if necessary.

```
DROP program-device-name FROM transaction-file-name.
```

Closing a TRANSACTION File

When you have finished using a TRANSACTION file, you must close it. Use the Format 1 CLOSE statement to close the TRANSACTION file. Once you close the file, it cannot be processed again until it is opened again.

```
CLOSE transaction-file-name.
```

Example of a Basic Inquiry Program Using Transaction Files

Figure 131 shows the associated DDS for a basic inquiry program that uses the ILE COBOL TRANSACTION file.

```

.....+.....1.....+.....2.....+.....3.....+.....4.....+.....5.....+.....6.....+.....7.....+.....8
A* CUSTOMER MASTER INQUIRY FILE ** CUSMINQ
A*
A          REF(CUSMSTP)
A          R CUSPMT          TEXT('CUSTOMER PROMPT')
A          CA01(15 'END OF PROGRAM')
A          1 3 'CUSTOMER MASTER INQUIRY'
A          3 3 'CUSTOMER NUMBER'
A          CUST          R          I 3 20
A 99          ERRMSG('CUSTOMER NUMBER NOT FOUND +
A          PRESS RESET, THEN ENTER A VALID NU+
A          MBER' 99)
A 98          ERRMSG('EOF CONDITION IN READ, +
A          PROGRAM ENDED' 98)
A          5 3 'USE F1 TO END PROGRAM, USE ENTE+
A          R TO RETURN TO PROMPT SCREEN'
A          R CUSFLDS          TEXT('CUSTOMER DISPLAY')
A          CA01(15 'END OF PROGRAM')
A          OVERLAY
A          8 3 'NAME'
A          NAME          R          8 11
A          9 3 'ADDRESS'
A          ADDR          R          9 11
A          10 3 'CITY'
A          CITY          R          10 11
A          11 3 'STATE'
A          STATE          R          11 11
A          11 21 'ZIP CODE'
A          ZIP          R          11 31
A          12 3 'A/R BALANCE'
A          ARBAL          R          12 17

```

Figure 131. Example of a TRANSACTION Inquiry Program Using a Single Display Device

The data description specifications (DDS) for the display device file (CUSMINQ) to be used by this program describe two record formats: CUSPMT and CUSFLDS.

The CUSPMT record format contains the constant 'Customer Master Inquiry', which identifies the display. It also contains the prompt 'Customer Number' and the input field (CUST) where you enter the customer number. Five underscores appear under the input field CUST on the display where you are to enter the customer number. The error message:

Customer number not found

is also included in this record format. This message is displayed if indicator 99 is set to **ON** by the program. In addition, this record format defines a function key that you can press to end the program. When you press function key F1, indicator 15 is set to **ON** in the ILE COBOL program. This indicator is then used to end the program.

The CUSFLDS record format contains the following constants:

- Name
- Address
- City
- State
- Zip Code
- Accounts Receivable Balance (A/R Balance).

These constants identify the fields to be written out from the program. This record format also describes the fields that correspond to these constants. All of these

fields are described as output fields (blank in position 38) because they are filled in by the program; you do not enter any data into these fields. To enter another customer number, press Enter in response to this record. Notice that the CUSFLDS record overlays the CUSPMT record. Therefore, when the CUSFLDS record is written to the display, the CUSPMT record remains on the display.

In addition to describing the constants, fields, and attributes for the display, the record formats also define the line numbers and horizontal positions where the constants and fields are to be displayed.

Note: The field attributes are defined in a physical file (CUSMSTP) used for field reference purposes, instead of in the DDS for the display file.

```

.....+.....1.....+.....2.....+.....3.....+.....4.....+.....5.....+.....6.....+.....7.....+.....
A* THIS IS THE CUSTOMER MASTER FILE ** CUSMSTP
A
A
A          R CUSMST                UNIQUE
A          CUST          5          TEXT('CUSTOMER MASTER RECORD')
A          NAME          25         TEXT('CUSTOMER NUMBER')
A          ADDR          20         TEXT('CUSTOMER NAME')
A          CITY          20         TEXT('CUSTOMER ADDRESS')
A          STATE         2          TEXT('CUSTOMER CITY')
A          ZIP           5 00       TEXT('STATE')
A          SRHCOD        6          TEXT('ZIP CODE')
A          CUSTYP        1 00       TEXT('CUSTOMER NUMBER SEARCH CODE')
A                                     TEXT('CUSTOMER TYPE 1=GOV 2=SCH +
A                                     3=BUS 4=PVT 5=OT')
A          ARBAL         8 02       TEXT('ACCOUNTS REC. BALANCE')
A          ORDBAL        8 02       TEXT('A/R AMT. IN ORDER FILE')
A          LSTAMT        8 02       TEXT('LAST AMT. PAID IN A/R')
A          LSTDAT        6 00       TEXT('LAST DATE PAID IN A/R')
A          CRDLMT        8 02       TEXT('CUSTOMER CREDIT LIMIT')
A          SLSYR         10 02      TEXT('CUSTOMER SALES THIS YEAR')
A          SLSLYR        10 02      TEXT('CUSTOMER SALES LAST YEAR')
A          K CUST

```

Figure 132. Data Description Specification for the Record Format CUSMST.

The data description specifications (DDS) for the database file that is used by this program describe one record format: CUSMST. Each field in the record format is described, and the CUST field is identified as the key field for the record format.

```

Source
STMT PL SEQNBR -A 1 B.+....2....+....3....+....4....+....5....+....6....+....7..IDENTFCN S COPYNAME CHG DATE
1 000100 IDENTIFICATION DIVISION.
2 000200 PROGRAM-ID. INQUIRY.
000300* SAMPLE TRANSACTION INQUIRY PROGRAM USING 1 DISPLAY DEVICE
000400
3 000500 ENVIRONMENT DIVISION.
4 000600 CONFIGURATION SECTION.
5 000700 SOURCE-COMPUTER. IBM-ISERIES.
6 000800 OBJECT-COMPUTER. IBM-ISERIES.
7 000900 INPUT-OUTPUT SECTION.
8 001000 FILE-CONTROL.
9 001100 SELECT CUST-DISPLAY
10 001200 ASSIGN TO WORKSTATION-CUSMINQ
11 001300 ORGANIZATION IS TRANSACTION
12 001400 CONTROL-AREA IS WS-CONTROL.
13 001500 SELECT CUST-MASTER
14 001600 ASSIGN TO DATABASE-CUSMSTP
15 001700 ORGANIZATION IS INDEXED
16 001800 ACCESS IS RANDOM
17 001900 RECORD KEY IS CUST OF CUSMST
18 002000 FILE STATUS IS CM-STATUS.
002100
19 002200 DATA DIVISION.
20 002300 FILE SECTION.
21 002400 FD CUST-DISPLAY.
22 002500 01 DISP-REC.
002600 COPY DDS-ALL-FORMATS OF CUSMINQ.
23 +000001 05 CUSMINQ-RECORD PIC X(80). <-ALL-FMTS
+000002* INPUT FORMAT:CUSPMT FROM FILE CUSMINQ OF LIBRARY CBLGUIDE <-ALL-FMTS
+000003* CUSTOMER PROMPT <-ALL-FMTS
24 +000004 05 CUSPMT-I REDEFINES CUSMINQ-RECORD. <-ALL-FMTS
25 +000005 06 CUSPMT-I-INDIC. <-ALL-FMTS
+000006 07 IN15 PIC 1 INDIC 15. <-ALL-FMTS
+000007* END OF PROGRAM <-ALL-FMTS
27 +000008 07 IN99 PIC 1 INDIC 99. <-ALL-FMTS
+000009* CUSTOMER NUMBER NOT FOUND PRESS RESET, THEN ENT <-ALL-FMTS
28 +000010 07 IN98 PIC 1 INDIC 98. <-ALL-FMTS
+000011* EOF CONDITION IN READ, PROGRAM ENDED <-ALL-FMTS
29 +000012 06 CUST PIC X(5). <-ALL-FMTS
+000013* CUSTOMER NUMBER <-ALL-FMTS
+000014* OUTPUT FORMAT:CUSPMT FROM FILE CUSMINQ OF LIBRARY CBLGUIDE <-ALL-FMTS
+000015* CUSTOMER PROMPT <-ALL-FMTS
30 +000016 05 CUSPMT-0 REDEFINES CUSMINQ-RECORD. <-ALL-FMTS
31 +000017 06 CUSPMT-0-INDIC. <-ALL-FMTS
32 +000018 07 IN99 PIC 1 INDIC 99. <-ALL-FMTS
+000019* CUSTOMER NUMBER NOT FOUND PRESS RESET, THEN ENT <-ALL-FMTS
33 +000020 07 IN98 PIC 1 INDIC 98. <-ALL-FMTS
+000021* EOF CONDITION IN READ, PROGRAM ENDED <-ALL-FMTS
+000022* INPUT FORMAT:CUSFLDS FROM FILE CUSMINQ OF LIBRARY CBLGUIDE <-ALL-FMTS
+000023* CUSTOMER DISPLAY <-ALL-FMTS
34 +000024 05 CUSFLDS-I REDEFINES CUSMINQ-RECORD. <-ALL-FMTS
35 +000025 06 CUSFLDS-I-INDIC. <-ALL-FMTS
36 +000026 07 IN15 PIC 1 INDIC 15. <-ALL-FMTS
+000027* END OF PROGRAM <-ALL-FMTS

```

Figure 133. Source Listing of a TRANSACTION Inquiry Program Using a Single Display Device. (Part 1 of 3)

```

5722WDS V5R4M0 060210 LN IBM ILE COBOL CBLGUIDE/INQUIRY ISERIES1 06/02/15 14:57:34 Page 3
STMT PL SEQNBR -A 1 B.+...2...+...3...+...4...+...5...+...6...+...7..IDENTFCN S COPYNAME CHG DATE
+000028* OUTPUT FORMAT:CUSFLDS FROM FILE CUSMINQ OF LIBRARY CBLGUIDE <-ALL-FMTS
+000029* CUSTOMER DISPLAY <-ALL-FMTS
37 +000030 05 CUSFLDS-0 REDEFINES CUSMINQ-RECORD. <-ALL-FMTS
38 +000031 06 NAME PIC X(25). <-ALL-FMTS
+000032* CUSTOMER NAME <-ALL-FMTS
39 +000033 06 ADDR PIC X(20). <-ALL-FMTS
+000034* CUSTOMER ADDRESS <-ALL-FMTS
40 +000035 06 CITY PIC X(20). <-ALL-FMTS
+000036* CUSTOMER CITY <-ALL-FMTS
41 +000037 06 STATE PIC X(2). <-ALL-FMTS
+000038* STATE <-ALL-FMTS
42 +000039 06 ZIP PIC S9(5). <-ALL-FMTS
+000040* ZIP CODE <-ALL-FMTS
43 +000041 06 ARBAL PIC S9(6)V9(2). <-ALL-FMTS
+000042* ACCOUNTS REC. BALANCE <-ALL-FMTS
002700
44 002800 FD CUST-MASTER.
45 002900 01 CUST-REC.
003000 COPY DDS-CUSMST OF CUSMSTP.
+000001* I-O FORMAT:CUSMST FROM FILE CUSMSTP OF LIBRARY CBLGUIDE CUSMST
+000002* CUSTOMER MASTER RECORD CUSMST
+000003* USER SUPPLIED KEY BY RECORD KEY CLAUSE CUSMST
46 +000004 05 CUSMST. CUSMST
47 +000005 06 CUST PIC X(5). CUSMST
+000006* CUSTOMER NUMBER CUSMST
48 +000007 06 NAME PIC X(25). CUSMST
+000008* CUSTOMER NAME CUSMST
49 +000009 06 ADDR PIC X(20). CUSMST
+000010* CUSTOMER ADDRESS CUSMST
50 +000011 06 CITY PIC X(20). CUSMST
+000012* CUSTOMER CITY CUSMST
51 +000013 06 STATE PIC X(2). CUSMST
+000014* STATE CUSMST
52 +000015 06 ZIP PIC S9(5) COMP-3. CUSMST
+000016* ZIP CODE CUSMST
53 +000017 06 SRHCOD PIC X(6). CUSMST
+000018* CUSTOMER NUMBER SEARCH CODE CUSMST
54 +000019 06 CUSTYP PIC S9(1) COMP-3. CUSMST
+000020* CUSTOMER TYPE 1=GOV 2=SCH 3=BUS 4=PVT 5=OT CUSMST
55 +000021 06 ARBAL PIC S9(6)V9(2) COMP-3. CUSMST
+000022* ACCOUNTS REC. BALANCE CUSMST
56 +000023 06 ORDBAL PIC S9(6)V9(2) COMP-3. CUSMST
+000024* A/R AMT. IN ORDER FILE CUSMST
57 +000025 06 LSTAMT PIC S9(6)V9(2) COMP-3. CUSMST
+000026* LAST AMT. PAID IN A/R CUSMST
58 +000027 06 LSTDAT PIC S9(6) COMP-3. CUSMST
+000028* LAST DATE PAID IN A/R CUSMST
59 +000029 06 CRDLMT PIC S9(6)V9(2) COMP-3. CUSMST
+000030* CUSTOMER CREDIT LIMIT CUSMST
60 +000031 06 SLSYR PIC S9(8)V9(2) COMP-3. CUSMST
+000032* CUSTOMER SALES THIS YEAR CUSMST
61 +000033 06 SLSLYR PIC S9(8)V9(2) COMP-3. CUSMST
+000034* CUSTOMER SALES LAST YEAR CUSMST
003100
62 003200 WORKING-STORAGE SECTION.

```

Figure 133. Source Listing of a TRANSACTION Inquiry Program Using a Single Display Device. (Part 2 of 3)

```

5722WDS V5R4M0 060210 LN IBM ILE COBOL          CBLGUIDE/INQUIRY      ISERIES1 06/02/15 14:57:34      Page 4
STMT PL SEQNBR -A 1 B.+...2....+...3....+...4....+...5....+...6....+...7...IDENTFCN S COPYNAME  CHG DATE
63 003300 01 ONE                                PIC 1 VALUE B"1".
64 003400 01 CM-STATUS                          PIC X(2).
65 003500 01 WS-CONTROL.
66 003600 02 WS-IND                             PIC X(2).
67 003700 02 WS-FORMAT                          PIC X(10).
003800
68 003900 PROCEDURE DIVISION.
69 004000 DECLARATIVES.
004100 DISPLAY-ERR-SECTION SECTION.
004200 USE AFTER STANDARD EXCEPTION PROCEDURE ON CUST-DISPLAY.
004300 DISPLAY-ERR-PARAGRAPH.
70 004400 MOVE ONE TO IN98 OF CUSPMT-0
71 004500 WRITE DISP-REC FORMAT IS "CUSPMT"
004600 END-WRITE
72 004700 CLOSE CUST-MASTER
004800 CUST-DISPLAY.
73 004900 STOP RUN.
005000 END DECLARATIVES.
005100
005200 MAIN-PROGRAM SECTION.
005300 MAINLINE.
74 005400 OPEN INPUT CUST-MASTER
005500 I-O CUST-DISPLAY.
005600
75 005700 MOVE ZERO TO IN99 OF CUSPMT-0
76 005800 WRITE DISP-REC FORMAT IS "CUSPMT" 1
005900 END-WRITE
77 006000 READ CUST-DISPLAY RECORD
006100 END-READ
006200
78 006300 PERFORM UNTIL IN15 OF CUSPMT-I IS EQUAL TO ONE
006400
006500 MOVE CUST OF CUSPMT-I TO CUST OF CUSMST
80 006600 READ CUST-MASTER RECORD 2
006700 INVALID KEY 3
81 006800 MOVE ONE TO IN99 OF CUSPMT-0
82 006900 WRITE DISP-REC FORMAT IS "CUSPMT"
007000 END-WRITE
83 007100 READ CUST-DISPLAY RECORD
007200 END-READ
007300 NOT INVALID KEY
84 007400 MOVE CORRESPONDING CUSMST TO CUSFLDS-0
*** CORRESPONDING items for statement 84:
*** NAME
*** ADDR
*** CITY
*** STATE
*** ZIP
*** ARBAL
*** End of CORRESPONDING items for statement 84
85 007500 WRITE DISP-REC FORMAT IS "CUSFLDS"
007600 END-WRITE
86 007700 READ CUST-DISPLAY RECORD
007800 END-READ
87 007900 IF IN15 OF CUSPMT-I IS NOT EQUAL TO ONE
88 008000 MOVE ZERO TO IN99 OF CUSPMT-0
89 008100 WRITE DISP-REC FORMAT IS "CUSPMT"
008200 END-WRITE
90 008300 READ CUST-DISPLAY RECORD
008400 END-READ
008500 END-IF
008600 END-READ
008700
008800 END-PERFORM
008900
91 009000 CLOSE CUST-MASTER
009100 CUST-DISPLAY.
92 009200 GOBACK.
***** END OF SOURCE *****

```

Figure 133. Source Listing of a TRANSACTION Inquiry Program Using a Single Display Device. (Part 3 of 3)

The complete source listing for this program example is shown here. In particular, note the FILE-CONTROL and FD entries and the data structures generated by the Format 2 COPY statements.

The WRITE operation at **1** writes the CUSPMT format to the display. This record prompts you to enter a customer number. If you enter a customer number and press Enter, the next READ operation then reads the record back into the program.

The READ operation at **2** uses the customer number (CUST) field to retrieve the corresponding CUSMST record from the CUSMSTP file. If no record is found in the CUSMSTP file, the INVALID KEY imperative statements at **3** are performed. Indicator 99 is set on and the message:

```
Customer number not found
```

is displayed when the format is written. The message is conditioned by indicator 99 in the DDS for the file. When you receive this message, the keyboard locks. You must press the Reset key in response to this message to unlock the keyboard. You can then enter another customer number.

If the READ operation retrieves a record from the CUSMSTP file, the WRITE operation writes the CUSFLDS record to the display workstation. This record contains the customer's name, address, and accounts receivable balance.

You then press Enter, and the program branches back to the beginning. You can enter another customer number or end the program. To end the program, press F1, which sets on indicator 15 in the program.

When indicator 15 is on, the program closes all files and processes the GOBACK statement. The program then returns control to the individual who called the ILE COBOL program.

This is the initial display written by the WRITE operation at **1**:

```
Customer Master Inquiry
Customer Number _____
Use F3 to end program, use enter key to return to prompt screen
```

This display appears if a record is found in the CUSMSTP file for the customer number entered in response to the first display:

```
Customer Master Inquiry

Customer Number  1000

Use F3 to end program, use enter key to return to prompt screen

Name      EXAMPLE WHOLESALERS LTD.
Address   ANYWHERE STREET
City      ACITY
State     IL           Zipcode  12345
A/R balance 137.02
```

This display appears if the CUSMSTP file does not contain a record for the customer number entered in response to the first display:

```
Customer Master Inquiry

Customer Number

Use F3 to end program, use enter key to return to prompt screen

Customer number not found, press reset, then enter valid number
```

Using Indicators with Transaction Files

Indicators are Boolean data items that can have the values B"0" or B"1".

When you define a record format for a file using DDS, you can condition the options using indicators; indicators can also be used to reflect particular responses. These indicators are known as OPTION and RESPONSE, respectively.

Option indicators provide options such as spacing, underlining, and allowing or requesting data transfer from your ILE COBOL program to a printer or display device. Response indicators provide response information to your ILE COBOL program from a device, such as function keys pressed by a workstation user, and whether data has been entered.

Indicators can be passed with data records in a record area, or outside the record area in a separate indicator area.

Passing Indicators in a Separate Indicator Area

```
# If you specify the file level keyword INDARA in the DDS, all indicators defined in
# the record format or formats for that file are passed to and from your ILE COBOL
# program in a separate indicator area, not in the record area. For information on
# how to specify the INDARA keyword, refer to the Database and File Systems
# category in the i5/OS Information Center at this Web site -http://www.ibm.com/systems/i/infocenter/.
#
```

The file control entry for a file that has INDARA specified in its DDS must have the separate indicator area attribute, SI, as part of the assignment-name. For example, the assignment for a file named DSPFILE is as follows:

```
FILE-CONTROL.  
  SELECT DISPPFILE  
    ASSIGN TO WORKSTATION-DSPFILE-SI  
    ORGANIZATION IS TRANSACTION  
    ACCESS IS SEQUENTIAL.
```

The advantages of using a separate indicator area are:

- The number and order of indicators used in an I/O statement for any record format in a file need not match the number and order of indicators specified in the DDS for that record format
- The program associates the indicator number in a data description entry with the appropriate indicator.

Passing Indicators in the Record Area

If the keyword `INDARA` is not used in the DDS of the file, indicators are created in the record area. When indicators are defined in a record format for a file, they are read, rewritten, and written with the data in the record area.

The number and order of indicators defined in the DDS for a record format for a file determines the number and order in which the data description entries for the indicators in the record format must be coded in your ILE COBOL program.

The file control entry for a file that does not have the `INDARA` keyword specified in the DDS associated with it must *not* have the separate indicator area attribute, `SI`, as part of the assignment-name.

If you use a `Format 2 COPY` statement to copy indicators into your ILE COBOL program, the indicators are defined in the order in which they are specified in the DDS for the file.

Examples of Using Indicators in ILE COBOL Programs

This section contains examples of ILE COBOL programs that illustrate the use of indicators in either a record area or a separate indicator area.

All of the ILE COBOL programs do the following:

1. Determine the current date.
2. If it is the first day of the month, turn on an option indicator that causes an output field to appear and blink.
3. Allow you to press function keys to terminate the program, or turn on response indicators and call programs to write daily or monthly reports.

Figure 135 on page 539 shows an ILE COBOL program that uses indicators in the record area but does not use the `INDICATORS` phrase in any I/O statement. Figure 134 on page 538 shows the associated DDS for the file.

Figure 136 on page 542 shows an ILE COBOL program that uses indicators in the record area and the `INDICATORS` phrase in the I/O statements. The associated DDS for Figure 136 is Figure 134 on page 538.

Figure 138 on page 545 shows an ILE COBOL program that uses indicators in a separate indicator area, defined in the `WORKING-STORAGE SECTION` by using the `Format 2 COPY` statement. Figure 137 on page 544 shows the associated DDS for the file.

Figure 139 on page 547 shows an ILE COBOL program that uses indicators in a separate indicator area, defined in a table in the WORKING-STORAGE SECTION. The associated DDS for the file is the same as Figure 137 on page 544.

```

.....+.....1.....+.....2.....+.....3.....+.....4.....+.....5.....+.....6.....+.....7.....+.....
A* DISPLAY FILE DDS FOR INDICATOR EXAMPLES - INDICATORS IN RECORD AREA
A* DSPFILEX
A      2 R FORMAT1
A
A
A      3 CF01(99 'END OF PROGRAM')
A      CF05(51 'DAILY REPORT')
A      CF09(52 'MONTHLY REPORT')
A*
A      4 10 10'DEPARTMENT NUMBER:'
A      5 I 10 32
A      5 01
A      20 26'PRODUCE MONTHLY REPORTS'
A      DSPATR(BL)
A*
A      6 24 01'F5 = DAILY REPORT'
A      24 26'F9 = MONTHLY REPORT'
A      24 53'F1 = TERMINATE'
A      R ERRFMT
A      98
A      6 5'INPUT-OUTPUT ERROR'

```

Figure 134. Example of a Program Using Indicators in the Record Area without Using the INDICATORS Phrase in the I/O Statement—DDS

- 1** The INDARA keyword is not used; indicators are stored in the record area with the data fields.
- 2** Record format FORMAT1 is specified.
- 3** Three indicators are associated with three function keys. Indicator 99 will be set on when you press F1, and so on.
- 4** One field is defined for input.
- 5** Indicator 01 is defined to cause the associated constant field to blink if the indicator is on.
- 6** The function (F) key definitions are documented on the workstation display.

```

Source
STMT PL SEQNBR -A 1 B.+...2...+...3...+...4...+...5...+...6...+...7..IDENTFCN S COPYNAME CHG DATE
1 000100 IDENTIFICATION DIVISION.
2 000200 PROGRAM-ID. INDIC1.
000300* SAMPLE PROGRAM WITH INDICATORS IN RECORD AREA.
000400
3 000500 ENVIRONMENT DIVISION.
4 000600 CONFIGURATION SECTION.
5 000700 SOURCE-COMPUTER. IBM-ISERIES
6 000800 OBJECT-COMPUTER. IBM-ISERIES
7 000900 INPUT-OUTPUT SECTION.
8 001000 FILE-CONTROL.
9 001100 SELECT DSPFILE
10 001200 ASSIGN TO WORKSTATION-DSPFILEX 1
11 001300 ORGANIZATION IS TRANSACTION
12 001400 ACCESS IS SEQUENTIAL.
001500
13 001600 DATA DIVISION.
14 001700 FILE SECTION.
15 001800 FD DSPFILE.
16 001900 01 DISP-REC.
002000 COPY DDS-ALL-FORMATS OF DSPFILEX. 2
17 +000001 05 DSPFILEX-RECORD PIC X(8). <-ALL-FMTS
+000002* INPUT FORMAT:FORMAT1 FROM FILE DSPFILEX OF LIBRARY CBLGUIDE <-ALL-FMTS
+000003*
18 +000004 05 FORMAT1-I REDEFINES DSPFILEX-RECORD. <-ALL-FMTS
19 +000005 06 FORMAT1-I-INDIC. <-ALL-FMTS
20 +000006 07 IN99 PIC 1 INDIC 99. 3 <-ALL-FMTS
+000007* END OF PROGRAM <-ALL-FMTS
21 +000008 07 IN51 PIC 1 INDIC 51. <-ALL-FMTS
+000009* DAILY REPORT <-ALL-FMTS
22 +000010 07 IN52 PIC 1 INDIC 52. <-ALL-FMTS
+000011* MONTHLY REPORT <-ALL-FMTS
23 +000012 06 DEPTNO PIC X(5). <-ALL-FMTS
+000013* OUTPUT FORMAT:FORMAT1 FROM FILE DSPFILEX OF LIBRARY CBLGUIDE <-ALL-FMTS
+000014*
24 +000015 05 FORMAT1-0 REDEFINES DSPFILEX-RECORD. <-ALL-FMTS
25 +000016 06 FORMAT1-0-INDIC. <-ALL-FMTS
26 +000017 07 IN01 PIC 1 INDIC 01. <-ALL-FMTS
+000018* INPUT FORMAT:ERRFMT FROM FILE DSPFILEX OF LIBRARY CBLGUIDE <-ALL-FMTS
+000019*
+000020* 05 ERRFMT-I REDEFINES DSPFILEX-RECORD. <-ALL-FMTS
+000021* OUTPUT FORMAT:ERRFMT FROM FILE DSPFILEX OF LIBRARY CBLGUIDE <-ALL-FMTS
+000022*
27 +000023 05 ERRFMT-0 REDEFINES DSPFILEX-RECORD. <-ALL-FMTS
28 +000024 06 ERRFMT-0-INDIC. <-ALL-FMTS
29 +000025 07 IN98 PIC 1 INDIC 98. <-ALL-FMTS
002100
30 002200 WORKING-STORAGE SECTION.
31 002300 01 CURRENT-DATE.
32 002400 05 CURR-YEAR PIC 9(2).
33 002500 05 CURR-MONTH PIC 9(2).
34 002600 05 CURR-DAY PIC 9(2).
35 002700 01 INDIC-AREA. 4
36 002800 05 IN01 PIC 1.

```

Figure 135. Example of a Program Using Indicators in the Record Area without Using the INDICATORS Phrase in the I/O Statement—COBOL Source Program (Part 1 of 2)

```

5722WDS V5R4M0 060210 LN IBM ILE COBOL          CBLGUIDE/INDIC1      ISERIES1 06/02/15 14:59:29      Page 3
STMT PL SEQNBR -A 1 B..+...2...+...3...+...4...+...5...+...6...+...7..IDENTFCN S COPYNAME  CHG DATE
37 002900      88 NEW-MONTH          VALUE B"1". 5
38 003000      05 IN51                PIC 1.
39 003100      88 WANT-DAILY          VALUE B"1".
40 003200      05 IN52                PIC 1.
41 003300      88 WANT-MONTHLY        VALUE B"1".
42 003400      05 IN98                PIC 1.
43 003500      88 IO-ERROR            VALUE B"1".
44 003600      05 IN99                PIC 1.
45 003700      88 NOT-END-OF-JOB        VALUE B"0".
46 003800      88 END-OF-JOB          VALUE B"1".
003900
47 004000      PROCEDURE DIVISION.
48 004100      DECLARATIVES.
004200      DISPLAY-ERR-SECTION SECTION.
004300      USE AFTER STANDARD EXCEPTION PROCEDURE ON DISPFILE.
004400      DISPLAY-ERR-PARAGRAPH.
49 004500      SET IO-ERROR TO TRUE
50 004600      MOVE CORR INDIC-AREA TO ERRFMT-0-INDIC
    *** CORRESPONDING items for statement 50:
    ***      IN98
    *** End of CORRESPONDING items for statement 50
51 004700      WRITE DISP-REC FORMAT IS "ERRFMT"
004800      END-WRITE
52 004900      CLOSE DISPFILE.
53 005000      STOP RUN.
005100      END DECLARATIVES.
005200
005300      MAIN-PROGRAM SECTION.
005400      MAINLINE.
54 005500      OPEN I-O DISPFILE.
55 005600      ACCEPT CURRENT-DATE FROM DATE.
56 005700      SET NOT-END-OF-JOB TO TRUE.
57 005800      PERFORM UNTIL END-OF-JOB
005900
58 006000      MOVE ZEROS TO INDIC-AREA 6
59 006100      IF CURR-DAY = 01 THEN
60 006200      SET NEW-MONTH TO TRUE 7
006300      END-IF
61 006400      MOVE CORR INDIC-AREA TO FORMAT1-0-INDIC 8
    *** CORRESPONDING items for statement 61:
    ***      IN01
    *** End of CORRESPONDING items for statement 61
62 006500      WRITE DISP-REC FORMAT IS "FORMAT1" 9
006600      END-WRITE
006700
63 006800      MOVE ZEROS TO INDIC-AREA
64 006900      READ DISPFILE FORMAT IS "FORMAT1" 10
007000      END-READ
65 007100      MOVE CORR FORMAT1-I-INDIC TO INDIC-AREA 11
    *** CORRESPONDING items for statement 65:
    ***      IN99
    ***      IN51
    ***      IN52
    *** End of CORRESPONDING items for statement 65
66 007200      IF WANT-DAILY THEN
67 007300      CALL "DAILY" USING DEPTNO
007400      ELSE
68 007500      IF WANT-MONTHLY THEN
69 007600      CALL "MONTHLY" USING DEPTNO 12
007700      END-IF
007800      END-IF
007900
008000      END-PERFORM.
70 008100      CLOSE DISPFILE.
71 008200      STOP RUN.
    * * * * * E N D   O F   S O U R C E   * * * * *

```

Figure 135. Example of a Program Using Indicators in the Record Area without Using the INDICATORS Phrase in the I/O Statement—COBOL Source Program (Part 2 of 2)

- 1** The separate indicator area attribute, SI, is not coded in the ASSIGN clause. As a result, the indicators form part of the record area.
- 2** The Format 2 COPY statement defines data fields and indicators in the record area.
- 3** Because the file indicators form part of the record area, response and

option indicators are defined in the order in which they are used in the DDS, and the indicator numbers are treated as documentation.

- 4** All indicators used by the program are defined with meaningful names in data description entries in the WORKING-STORAGE SECTION. Indicator numbers are omitted here because they have no effect.
- 5** For each indicator, a meaningful level-88 condition-name is associated with a value for that indicator.
- 6** Initialize group level to zeros.
- 7** IN01 in the WORKING-STORAGE SECTION is set on if it is the first day of the month.
- 8** Indicators appropriate to the output of FORMAT1 are copied to the record area.
- 9** FORMAT1 is written to the workstation display with both data and indicator values in the record area.

The INDICATORS phrase is not necessary because there is no separate indicator area and indicator values have been set in the record area through the previous MOVE CORRESPONDING statement.

- 10** FORMAT1, including both data and indicators, is read from the display.
- 11** The response indicators for FORMAT1 are copied from the record area to the data description entries in the WORKING-STORAGE SECTION.
- 12** If F5 has been pressed, a program call is processed.

```

Source
STMT PL SEQNBR -A 1 B.+....2....+....3....+....4....+....5....+....6....+....7..IDENTFCN S COPYNAME CHG DATE
1 000100 IDENTIFICATION DIVISION.
2 000200 PROGRAM-ID. INDIC2.
000300* SAMPLE PROGRAM - FILE WITH INDICATORS IN RECORD AREA
000400
3 000500 ENVIRONMENT DIVISION.
4 000600 CONFIGURATION SECTION.
5 000700 SOURCE-COMPUTER. IBM-ISERIES
6 000800 OBJECT-COMPUTER. IBM-ISERIES
7 000900 INPUT-OUTPUT SECTION.
8 001000 FILE-CONTROL.
9 001100 SELECT DSPFILE
10 001200 ASSIGN TO WORKSTATION-DSPFILEX 1
11 001300 ORGANIZATION IS TRANSACTION
12 001400 ACCESS IS SEQUENTIAL.
001500
13 001600 DATA DIVISION.
14 001700 FILE SECTION.
15 001800 FD DSPFILE.
16 001900 01 DISP-REC.
002000 COPY DDS-ALL-FORMATS OF DSPFILEX. 2
17 +000001 05 DSPFILEX-RECORD PIC X(8). <-ALL-FMTS
+000002* INPUT FORMAT:FORMAT1 FROM FILE DSPFILEX OF LIBRARY CBLGUIDE <-ALL-FMTS
+000003* <-ALL-FMTS
18 +000004 05 FORMAT1-I REDEFINES DSPFILEX-RECORD. <-ALL-FMTS
19 +000005 06 FORMAT1-I-INDIC. <-ALL-FMTS
20 +000006 07 IN99 PIC 1 INDIC 99. 3 <-ALL-FMTS
+000007* END OF PROGRAM <-ALL-FMTS
21 +000008 07 IN51 PIC 1 INDIC 51. <-ALL-FMTS
+000009* DAILY REPORT <-ALL-FMTS
22 +000010 07 IN52 PIC 1 INDIC 52. <-ALL-FMTS
+000011* MONTHLY REPORT <-ALL-FMTS
23 +000012 06 DEPTNO PIC X(5). <-ALL-FMTS
+000013* OUTPUT FORMAT:FORMAT1 FROM FILE DSPFILEX OF LIBRARY CBLGUIDE <-ALL-FMTS
+000014* <-ALL-FMTS
24 +000015 05 FORMAT1-0 REDEFINES DSPFILEX-RECORD. <-ALL-FMTS
25 +000016 06 FORMAT1-0-INDIC. <-ALL-FMTS
26 +000017 07 IN01 PIC 1 INDIC 01. <-ALL-FMTS
+000018* INPUT FORMAT:ERRFMT FROM FILE DSPFILEX OF LIBRARY CBLGUIDE <-ALL-FMTS
+000019* <-ALL-FMTS
+000020* 05 ERRFMT-I REDEFINES DSPFILEX-RECORD. <-ALL-FMTS
+000021* OUTPUT FORMAT:ERRFMT FROM FILE DSPFILEX OF LIBRARY CBLGUIDE <-ALL-FMTS
+000022* <-ALL-FMTS
27 +000023 05 ERRFMT-0 REDEFINES DSPFILEX-RECORD. <-ALL-FMTS
28 +000024 06 ERRFMT-0-INDIC. <-ALL-FMTS
29 +000025 07 IN98 PIC 1 INDIC 98. <-ALL-FMTS
002100
30 002200 WORKING-STORAGE SECTION.
31 002300 01 CURRENT-DATE.
32 002400 05 CURR-YEAR PIC 9(2).
33 002500 05 CURR-MONTH PIC 9(2).
34 002600 05 CURR-DAY PIC 9(2).
002700
35 002800 77 IND-OFF PIC 1 VALUE B"0".

```

Figure 136. Example of Program Using Indicators in the Record Area and the INDICATORS Phrase in I/O Statements—COBOL Source Program (Part 1 of 2)

```

5722WDS V5R4M0 060210 LN IBM ILE COBOL          CBLGUIDE/INDIC1      ISERIES1 06/02/15 15:00:29      Page 3
STMT PL SEQNBR -A 1 B.+...2...+...3...+...4...+...5...+...6...+...7..IDENTFCN S COPYNAME  CHG DATE
36 002900 77 IND-ON                               PIC 1      VALUE B"1".
003000
37 003100 01 RESPONSE-INDICS.
38 003200 05 END-OF-PROGRAM                         PIC 1.     4
39 003300 05 DAILY-REPORT                          PIC 1.
40 003400 05 MONTHLY-REPORT                        PIC 1.
41 003500 01 OPTION-INDICS.
42 003600 05 NEW-MONTH                             PIC 1.
43 003700 01 ERROR-INDICS.
44 003800 05 IO-ERROR                              PIC 1.
003900
45 004000 PROCEDURE DIVISION.
46 004100 DECLARATIVES.
004200 DISPLAY-ERR-SECTION SECTION.
004300 USE AFTER STANDARD EXCEPTION PROCEDURE ON DISPFILE.
004400 DISPLAY-ERR-PARAGRAPH.
47 004500 MOVE IND-ON TO IO-ERROR
48 004600 WRITE DISP-REC FORMAT IS "ERRFMT"
004700 INDICATORS ARE ERROR-INDICS
004800 END-WRITE
49 004900 CLOSE DISPFILE.
50 005000 STOP RUN.
005100 END DECLARATIVES.
005200
005300 MAIN-PROGRAM SECTION.
005400 MAINLINE.
51 005500 OPEN I-O DISPFILE.
52 005600 ACCEPT CURRENT-DATE FROM DATE.
53 005700 MOVE IND-OFF TO END-OF-PROGRAM.
54 005800 PERFORM UNTIL END-OF-PROGRAM = IND-ON
55 005900 MOVE ZEROS TO OPTION-INDICS
56 006000 IF CURR-DAY = 01 THEN 5
57 006100 MOVE IND-ON TO NEW-MONTH
006200 END-IF
58 006300 WRITE DISP-REC FORMAT IS "FORMAT1" 6
006400 INDICATORS ARE OPTION-INDICS
006500 END-WRITE
006600
59 006700 MOVE ZEROS TO RESPONSE-INDICS
60 006800 READ DISPFILE FORMAT IS "FORMAT1" 7
006900 INDICATORS ARE RESPONSE-INDICS 8
007000 END-READ
61 007100 IF DAILY-REPORT = IND-ON THEN
62 007200 CALL "DAILY" USING DEPTNO 9
007300 ELSE
63 007400 IF MONTHLY-REPORT = IND-ON THEN
64 007500 CALL "MONTHLY" USING DEPTNO
007600 END-IF
007700 END-IF
007800
007900 END-PERFORM
65 008000 CLOSE DISPFILE.
66 008100 STOP RUN.
008200
***** END OF SOURCE *****

```

Figure 136. Example of Program Using Indicators in the Record Area and the INDICATORS Phrase in I/O Statements—COBOL Source Program (Part 2 of 2)

- 1 The separate indicator area attribute, SI, is not coded in the ASSIGN clause.
- 2 The Format 2 COPY statement defines data fields and indicators in the record area.
- 3 Because the file does not have a separate indicator area, response and option indicators are defined in the order in which they are used in the DDS, and the indicator numbers are treated as documentation.
- 4 All indicators used by the program are defined with meaningful names in data description entries in the WORKING-STORAGE SECTION. Indicator numbers are omitted here because they have no effect. Indicators should be defined in the order needed by the display file.

- 5** IN01 in the WORKING-STORAGE SECTION is set on if it is the first day of the month.
- 6** FORMAT1 is written to the workstation display:
 - The INDICATORS phrase causes the contents of the variable OPTION-INDICS to be copied to the beginning of the record area.
 - Data and indicator values are written to the workstation display.
- 7** FORMAT1, including both data and indicators, is read from the workstation display.
- 8** The INDICATORS phrase causes bytes to be copied from the beginning of the record area to RESPONSE-INDICS.
- 9** If F5 has been pressed, a program call is processed.

```

.....+.....1.....+.....2.....+.....3.....+.....4.....+.....5.....+.....6.....+.....7.....+.....
A* DISPLAY FILE FOR INDICATOR EXAMPLES - INDICATORS IN SI AREA
A* DSPFILE
A
A          R FORMAT1
A
A          INDARA 1
A          CF01(99 'END OF PROGRAM')
A          CF05(51 'DAILY REPORT')
A          CF09(52 'MONTHLY REPORT')
A*
A          DEPTNO      5  I 10 32
A          01          20 26 'PRODUCE MONTHLY REPORTS'
A          DSPATR(BL)
A*
A          24 01 'F5 = DAILY REPORT'
A          24 26 'F9 = MONTHLY REPORT'
A          24 53 'F1 = TERMINATE'
A          R ERRFMT
A          98          6  5 'INPUT-OUTPUT ERROR'

```

Figure 137. Example of a Program Using Indicators in a Separate Indicator Area, Defined in WORKING-STORAGE by Using the COPY Statement ** DDS

- 1** The INDARA keyword is specified; indicators are stored in a separate indicator area, not in the record area. Except for this specification, the DDS for this file is the same as that shown in Figure 134 on page 538.

```

Source
STMT PL SEQNBR -A 1 B.+....2...+....3...+....4...+....5...+....6...+....7..IDENTFCN S COPYNAME CHG DATE
1 000100 IDENTIFICATION DIVISION.
2 000200 PROGRAM-ID. INDIC3.
000300* SAMPLE PROGRAM - FILE WITH SEPERATE INDICATORS AREA
000400
3 000500 ENVIRONMENT DIVISION.
4 000600 CONFIGURATION SECTION.
5 000700 SOURCE-COMPUTER. IBM-ISERIES
6 000800 OBJECT-COMPUTER. IBM-ISERIES
7 000900 INPUT-OUTPUT SECTION.
8 001000 FILE-CONTROL.
9 001100 SELECT DISPFIL
10 001200 ASSIGN TO WORKSTATION-DSPFILE-SI 1
11 001300 ORGANIZATION IS TRANSACTION
12 001400 ACCESS IS SEQUENTIAL.
001500
13 001600 DATA DIVISION.
14 001700 FILE SECTION.
15 001800 FD DISPFIL.
16 001900 01 DISP-REC.
002000 COPY DDS-ALL-FORMATS OF DSPFILE. 2
17 +000001 05 DSPFILE-RECORD PIC X(5). <-ALL-FMTS
+000002* INPUT FORMAT:FORMAT1 FROM FILE DSPFILE OF LIBRARY CBLGUIDE <-ALL-FMTS
+000003* <-ALL-FMTS
18 +000004 05 FORMAT1-I REDEFINES DSPFILE-RECORD. <-ALL-FMTS
19 +000005 06 DEPTNO PIC X(5). <-ALL-FMTS
+000006* OUTPUT FORMAT:FORMAT1 FROM FILE DSPFILE OF LIBRARY CBLGUIDE <-ALL-FMTS
+000007* <-ALL-FMTS
+000008* 05 FORMAT1-0 REDEFINES DSPFILE-RECORD. <-ALL-FMTS
+000009* INPUT FORMAT:ERRFMT FROM FILE DSPFILE OF LIBRARY CBLGUIDE <-ALL-FMTS
+000010* <-ALL-FMTS
+000011* 05 ERRFMT-I REDEFINES DSPFILE-RECORD. <-ALL-FMTS
+000012* OUTPUT FORMAT:ERRFMT FROM FILE DSPFILE OF LIBRARY CBLGUIDE <-ALL-FMTS
+000013* <-ALL-FMTS
+000014* 05 ERRFMT-0 REDEFINES DSPFILE-RECORD. <-ALL-FMTS
002100
20 002200 WORKING-STORAGE SECTION.
21 002300 01 CURRENT-DATE.
22 002400 05 CURR-YEAR PIC 9(2).
23 002500 05 CURR-MONTH PIC 9(2).
24 002600 05 CURR-DAY PIC 9(2).
002700
25 002800 77 IND-OFF PIC 1 VALUE B"0".
26 002900 77 IND-ON PIC 1 VALUE B"1".
003000
27 003100 01 DISPFIL-INDICS.
003200 COPY DDS-ALL-FORMATS-INDIC OF DSPFILE. 3
28 +000001 05 DSPFILE-RECORD. <-ALL-FMTS
+000002* INPUT FORMAT:FORMAT1 FROM FILE DSPFILE OF LIBRARY CBLGUIDE <-ALL-FMTS
+000003* <-ALL-FMTS
29 +000004 06 FORMAT1-I-INDIC. <-ALL-FMTS
30 +000005 07 IN51 PIC 1 INDIC 51. 4 <-ALL-FMTS
+000006* DAILY REPORT <-ALL-FMTS
31 +000007 07 IN52 PIC 1 INDIC 52. <-ALL-FMTS

```

Figure 138. COBOL Listing Using Indicators in a Separate Indicator Area (Part 1 of 2)

```

5722WDS V5R4M0 060210 LN IBM ILE COBOL          CBLGUIDE/INDIC1      ISERIES1 06/02/15 15:01:36      Page 3
  STMT PL SEQNBR -A 1 B..+...2...+...3...+...4...+...5...+...6...+...7...IDENTFCN S COPYNAME  CHG DATE
+000008*          MONTHLY REPORT                                <-ALL-FMTS
32 +000009          07 IN99          PIC 1 INDIC 99.              <-ALL-FMTS
+000010*          END OF PROGRAM                                <-ALL-FMTS
+000011* OUTPUT FORMAT:FORMAT1 FROM FILE DSPFILE OF LIBRARY CBLGUIDE <-ALL-FMTS
+000012*          <-ALL-FMTS
33 +000013          06 FORMAT1-0-INDIC.                          <-ALL-FMTS
34 +000014          07 IN01          PIC 1 INDIC 01.              <-ALL-FMTS
+000015* INPUT FORMAT:ERRFMT FROM FILE DSPFILE OF LIBRARY CBLGUIDE <-ALL-FMTS
+000016*          <-ALL-FMTS
+000017*          06 ERRFMT-I-INDIC.                             <-ALL-FMTS
+000018* OUTPUT FORMAT:ERRFMT FROM FILE DSPFILE OF LIBRARY CBLGUIDE <-ALL-FMTS
+000019*          <-ALL-FMTS
35 +000020          06 ERRFMT-0-INDIC.                            <-ALL-FMTS
36 +000021          07 IN98          PIC 1 INDIC 98.              <-ALL-FMTS
003300
37 003400 PROCEDURE DIVISION.
38 003500 DECLARATIVES.
003600 DISPLAY-ERR-SECTION SECTION.
003700 USE AFTER STANDARD EXCEPTION PROCEDURE ON DISPFILE.
003800 DISPLAY-ERR-PARAGRAPH.
39 003900 MOVE IND-ON TO IN98 IN ERRFMT-0-INDIC
40 004000 WRITE DISP-REC FORMAT IS "ERRFMT"
004100 INDICATORS ARE ERRFMT-0-INDIC
004200 END-WRITE
41 004300 CLOSE DISPFILE.
42 004400 STOP RUN.
004500 END DECLARATIVES.
004600
004700 MAIN-PROGRAM SECTION.
004800 MAINLINE.
004900
43 005000 OPEN I-0 DISPFILE.
44 005100 ACCEPT CURRENT-DATE FROM DATE.
45 005200 MOVE IND-OFF TO IN99 IN FORMAT1-I-INDIC.
46 005300 PERFORM UNTIL IN99 IN FORMAT1-I-INDIC = IND-ON
005400
47 005500 MOVE ZEROS TO FORMAT1-0-INDIC
48 005600 IF CURR-DAY = 01 THEN
49 005700 MOVE IND-ON TO IN01 IN FORMAT1-0-INDIC 5
005800 END-IF
50 005900 WRITE DISP-REC FORMAT IS "FORMAT1"
006000 INDICATORS ARE FORMAT1-0-INDIC 6
006100 END-WRITE
006200
51 006300 MOVE ZEROS TO FORMAT1-I-INDIC
52 006400 READ DISPFILE FORMAT IS "FORMAT1"
006500 INDICATORS ARE FORMAT1-I-INDIC 7
006600 END-READ
53 006700 IF IN51 IN FORMAT1-I-INDIC = IND-ON THEN
54 006800 CALL "DAILY" USING DEPTNO
006900 ELSE
55 007000 IF IN52 IN FORMAT1-I-INDIC = IND-ON THEN
56 007100 CALL "MONTHLY" USING DEPTNO 8
007200 END-IF
007300 END-IF
007400
007500 END-PERFORM
007600 CLOSE DISPFILE.
58 007700 STOP RUN.
007800
          * * * * * E N D O F S O U R C E * * * * *

```

Figure 138. COBOL Listing Using Indicators in a Separate Indicator Area (Part 2 of 2)

- 1** The separate indicator area attribute, SI, is specified in the ASSIGN clause.
- 2** The Format 2 COPY statement generates data descriptions in the record area for data fields only. The data description entries for the indicators are not generated because a separate indicator area has been specified for the file.
- 3** The Format 2 COPY statement, with the INDICATOR attribute, INDIC, defines data description entries in the WORKING-STORAGE SECTION for all indicators used in the DDS for the record format for the file.
- 4** Because the file has a separate indicator area, the indicator numbers used in the data description entries are not treated as documentation.

- 5** IN01 in the separate indicator area for FORMAT1 is set on if it is the first day of the month.
- 6** The INDICATORS phrase is required to send indicator values to the workstation display.
- 7** The INDICATORS phrase is required to receive indicator values from the workstation display. If you have pressed F5, IN51 is set on.
- 8** If IN51 has been set on, a program call is processed.

```

5722WDS V5R4M0 060210 LN IBM ILE COBOL CBLGUIDE/INDIC4 ISERIES1 06/02/15 15:02:22 Page 2
Source
STMT PL SEQNBR -A 1 B.+...2...+...3...+...4...+...5...+...6...+...7..IDENTFCN S COPYNAME CHG DATE
1 000100 IDENTIFICATION DIVISION.
2 000200 PROGRAM-ID. INDIC4.
000300* SAMPLE PROGRAM
000400* FILE WITH SEPERATE INDICATORS AREA IN WORKING STORAGE
000500
3 000600 ENVIRONMENT DIVISION.
4 000700 CONFIGURATION SECTION.
5 000800 SOURCE-COMPUTER. IBM-ISERIES
6 000900 OBJECT-COMPUTER. IBM-ISERIES
7 001000 INPUT-OUTPUT SECTION.
8 001100 FILE-CONTROL.
9 001200 SELECT DSPFILE
10 001300 ASSIGN TO WORKSTATION-DSPFILE-SI 1
11 001400 ORGANIZATION IS TRANSACTION
12 001500 ACCESS IS SEQUENTIAL.
001600
13 001700 DATA DIVISION.
14 001800 FILE SECTION.
15 001900 FD DSPFILE.
16 002000 01 DISP-REC.
002100 COPY DDS-ALL-FORMATS OF DSPFILE. 2
17 +000001 05 DSPFILE-RECORD PIC X(5).
+000002* INPUT FORMAT:FORMAT1 FROM FILE DSPFILE OF LIBRARY CBLGUIDE
+000003*
18 +000004 05 FORMAT1-I REDEFINES DSPFILE-RECORD.
19 +000005 06 DEPTNO PIC X(5).
+000006* OUTPUT FORMAT:FORMAT1 FROM FILE DSPFILE OF LIBRARY CBLGUIDE
+000007*
+000008* 05 FORMAT1-0 REDEFINES DSPFILE-RECORD.
+000009* INPUT FORMAT:ERRFMT FROM FILE DSPFILE OF LIBRARY CBLGUIDE
+000010*
+000011* 05 ERRFMT-I REDEFINES DSPFILE-RECORD.
+000012* OUTPUT FORMAT:ERRFMT FROM FILE DSPFILE OF LIBRARY CBLGUIDE
+000013*
+000014* 05 ERRFMT-0 REDEFINES DSPFILE-RECORD.
002200
20 002300 WORKING-STORAGE SECTION.
21 002400 01 CURRENT-DATE.
22 002500 05 CURR-YEAR PIC 9(2).
23 002600 05 CURR-MONTH PIC 9(2).
24 002700 05 CURR-DAY PIC 9(2).
002800
25 002900 01 INDIC-AREA.
26 003000 05 INDIC-TABLE OCCURS 99 PIC 1 INDICATOR 1. 3
27 003100 88 IND-OFF VALUE B"0".
28 003200 88 IND-ON VALUE B"1".
003300
29 003400 01 DSPFILE-INDIC-USAGE.
30 003500 05 IND-NEW-MONTH PIC 9(2) VALUE 01.
31 003600 05 IND-DAILY PIC 9(2) VALUE 51. 4
32 003700 05 IND-MONTHLY PIC 9(2) VALUE 52.
33 003800 05 IND-IO-ERROR PIC 9(2) VALUE 98.
34 003900 05 IND-EQJ PIC 9(2) VALUE 99.

```

Figure 139. Example of a Program Using Indicators in a Separate Indicator Area, Defined in a Table in WORKING-STORAGE (Part 1 of 2)

```

5722WDS V5R4M0 060210 LN IBM ILE COBOL          CBLGUIDE/INDIC4          ISERIES1 06/02/15 15:02:22      Page 3
STMT PL SEQNBR -A 1 B..+...2....+...3....+...4....+...5....+...6....+...7..IDENTFCN S COPYNAME  CHG DATE
004000
35 004100 PROCEDURE DIVISION.
36 004200 DECLARATIVES.
004300 DISPLAY-ERR-SECTION SECTION.
004400 USE AFTER STANDARD EXCEPTION PROCEDURE ON DISPFILE.
004500 DISPLAY-ERR-PARAGRAPH.
37 004600 SET IND-ON (IND-IO-ERROR) TO TRUE
38 004700 WRITE DISP-REC FORMAT IS "ERRFMT"
004800 INDICATORS ARE INDIC-TABLE
004900 END-WRITE
39 005000 CLOSE DISPFILE.
40 005100 STOP RUN.
005200 END DECLARATIVES.
005300
005400 MAIN-PROGRAM SECTION.
005500 MAINLINE.
41 005600 OPEN I-O DISPFILE.
42 005700 ACCEPT CURRENT-DATE FROM DATE.
43 005800 SET IND-OFF (IND-EOJ) TO TRUE.
44 005900 PERFORM UNTIL IND-ON (IND-EOJ)
006000
45 006100 MOVE ZEROS TO INDIC-AREA
46 006200 IF CURR-DAY = 01 THEN
47 006300 SET IND-ON (IND-NEW-MONTH) TO TRUE 5
006400 END-IF
48 006500 WRITE DISP-REC FORMAT IS "FORMAT1"
006600 INDICATORS ARE INDIC-TABLE 6
006700 END-WRITE
006800
49 006900 READ DISPFILE FORMAT IS "FORMAT1"
007000 INDICATORS ARE INDIC-TABLE 7
007100 END-READ
50 007200 IF IND-ON (IND-DAILY) THEN
51 007300 CALL "DAILY" USING DEPTNO 8
007400 ELSE
52 007500 IF IND-ON (IND-MONTHLY) THEN
53 007600 CALL "MONTHLY" USING DEPTNO
007700 END-IF
007800 END-IF
007900
008000 END-PERFORM
54 008100 CLOSE DISPFILE.
55 008200 STOP RUN.
008300
***** END OF SOURCE *****

```

Figure 139. Example of a Program Using Indicators in a Separate Indicator Area, Defined in a Table in WORKING-STORAGE (Part 2 of 2)

- 1 The separate indicator area attribute, SI, is specified in the ASSIGN clause.
- 2 The Format 2 COPY statement generates fields in the record area for data fields only.
- 3 A table of 99 Boolean data items is defined in the WORKING-STORAGE SECTION. The INDICATOR clause for this data description entry causes these data items to be associated with indicators 1 through 99 respectively. The use of such a table may result in improved performance as compared to the use of a group item with multiple subordinate entries for individual indicators.
- 4 A series of data items is defined in the WORKING-STORAGE SECTION to provide meaningful subscript names with which to refer to the table of indicators. The use of such data items is not required.
- 5 INDIC-TABLE (01) in the separate indicator area for FORMAT1 is set on if it is the first day of the month.
- 6 The INDICATOR phrase is required to send indicator values to the workstation display.

- 7** The INDICATOR phrase is required to receive indicator values from the workstation display. If F5 has been pressed, INDIC-TABLE (51) will be set on.
- 8** If INDIC-TABLE (51) has been set on, program DAILY is called.

Using Subfile Transaction Files

A **subfile** is a group of records that are read from or written to a display device. The program processes one record at a time, but the operating system and the workstation send and receive blocks of records. If more records are transmitted than can be shown on the display at one time, the workstation operator can page through the block of records without returning control to the program.

Subfiles offer a convenient way of reading and writing large numbers of similar records to and from displays. Subfiles are display files whose records can be accessed sequentially or randomly by relative key value.

For example, suppose you want to display all customers who have spent more than \$5000 with your company over the last year. You can do a query of your database and get the names of all these customers, and place them in a special file (the subfile), by performing WRITE SUBFILE operations on the subfile. When you have done this, you can write the entire contents of the subfile to the display by performing a WRITE operation on the subfile control record. Then you can read the customer list as modified by the user using a READ operation on the subfile control record, and subsequently retrieve the individual records from the subfile using READ SUBFILE operations.

Subfiles can be specified in the DDS for a display file to allow you to handle multiple records of the same type on a display. See Figure 140 on page 550 for an example of a subfile display.

Records formats to be included in a subfile are specified in the DDS for the file. The number of records that can be contained in a subfile must also be specified in the DDS. One file can contain more than one subfile; however, only twelve subfiles can be active concurrently for a device.

Defining a Subfile Using Data Description Specifications

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 descriptions for the records in the subfile. Specifications of the subfile record format on a READ, WRITE, or REWRITE causes the specified subfile record to be processed, but does not directly affect the displayed data.

Specification of the subfile control record format on a READ or WRITE statement causes the physical read, write, or setup operations of a subfile to take place. Figure 141 on page 552 shows an example of the DDS for a subfile record format and a subfile control record format.

For a description of how the records in a subfile can be displayed and for a
description of the keywords that can be specified for a subfile, refer to the *Database*
and File Systems category in the **i5/OS Information Center** at this Web site
[-http://www.ibm.com/systems/i/infocenter/](http://www.ibm.com/systems/i/infocenter/).

Table 32. Uses of Subfiles (continued)

Use	Meaning
Modification	The user modifies 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, and the records are checked as well.
Combination of Tasks	A subfile can be used as a display with modification.

```

.....1.....2.....3.....4.....5.....6.....7.....
A* THIS IS THE DISPLAY DEVICE FILE FOR PAYUPDT ** PAYUPDTD
A* ACCOUNTS RECEIVABLE INTERACTIVE PAYMENT UPDATE
A*
A
A          R SUBFILE1                      SFL 1
A                                     TEXT('SUBFILE FOR CUSTOMER PAYMENT')
A*
A          ACPMPT          4A I 5 4TEXT('ACCEPT PAYMENT')
A                                     2 VALUES('*YES' '*NO') 3
A 51                                DSPATR(RI MDT)
A N51                               DSPATR(ND PR)
A*
A          CUST           5 B 5 15TEXT('CUSTOMER NUMBER')
A 52                                4 DSPATR(RI)
A 53                                DSPATR(ND)
A 54                                DSPATR(PR)
A*
A          AMPAID        8 02B 5 24TEXT('AMOUNT PAID')
A                                     CHECK(FE) 5
A                                     AUTO(RAB) 6
A                                     CMP(GT 0) 7
A 52                                DSPATR(RI)
A 53                                DSPATR(ND)
A 54                                DSPATR(PR)
A*
A          ECPMSG        31A 0 5 37TEXT('EXCEPTION MESSAGE')
A 52                                DSPATR(RI)
A 53                                DSPATR(ND)
A 54                                DSPATR(BL)
A*
A          OVRPMT        8Y 20 5 70TEXT('OVERPAYMENT')
A                                     EDTCDE(1) 8
A 55                                DSPATR(BL) 9
A N56                               DSPATR(ND)
A*
A          STSCDE        1A H          TEXT('STATUS CODE')
A          R CONTROL1          TEXT('SUBFILE CONTROL')
A                                     SFLCTL(SUBFILE1) 10
A                                     SFLSIZ(17) 11
A                                     SFLPAG(17) 12
A 61                                SFLCLR 13
A 62                                SFLDSP 14
A 62                                SFLDSPCTL 15
A                                     OVERLAY
A                                     LOCK 16
A*
A                                     HELP(99 'HELP KEY') 17
A                                     CA12(98 'END PAYMENT UPDATE')
A                                     CA11(97 'IGNORE INPUT')
A* 18
A 99                                SFLMSG(' F11 - IGNORE INVALID INPUT+
A                                     F12 - END PAYMENT +
A                                     UPDATE')

```

Figure 141. Data Description Specifications for a Subfile Record Format (Part 1 of 2)

```

A*
A          1 2'CUSTOMER PAYMENT UPDATE PROMPT'
A          1 65'DATE'
A          1 71DATE EDTCDE(Y)
A 63      3 2'ACCEPT'
A 63      4 2'PAYMENT'
A          3 14'CUSTOMER'
A          3 26'PAYMENT'
A 64      3 37'EXCEPTION MESSAGE'
A*
A          R MESSAGE1          TEXT('MESSAGE RECORD')
A          OVERLAY
A          LOCK
A*
A 71      24 2' ACCEPT PAYMENT VALUES: (*NO
*YES)
DSPATR(RI)

```

Figure 141. Data Description Specifications for a Subfile Record Format (Part 2 of 2)

The data description specifications (DDS) for a subfile record format describe the records in the subfile:

- 1** The SFL keyword identifies the record format as a subfile.
- 2** The line and position entries define the location of the fields on the display.
- 3** The VALUES keyword specifies that the user can only specify *YES or *NO as values for the ACPMT field.
- 4** The usage entries define whether the named field is to be an output (O), input (I), output/input (B), or hidden (H) field.
- 5** The entry CHECK(FE) specifies that the user cannot skip to the next input field without pressing one of the field exit keys.
- 6** The entry AUTO(RAB) specifies that data entered into the field AMPAID is to be automatically right-justified, and the leading characters are to be filled with blanks.
- 7** The entry CMP(GT 0) specifies that the data entered for the field AMPAID is to be compared to zero to ensure that the value is greater than zero.
- 8** The EDTCDE keyword specifies the desired editing for output field OVRPMT. EDTCDE(1) indicates that the field OVRPMT is to be printed with commas, decimal point, and no sign. Also, a zero balance will be printed, and leading zeros will be suppressed.
- 9** The DSPATR keyword is used to specify the display attributes for the named field when the corresponding indicator status is true. The attributes specified are:
 - BL (blink)
 - RI (reverse image)
 - PR (protect)
 - MDT (set modified data tag)
 - ND (nondisplay).

The subfile control record format defines the attributes of the subfile, the search input field, constants, and command keys. The keywords used indicate the following:

- 10** SFLCTL identifies this record as a subfile control record and names the associated subfile record (SUBFILE1).
- 11** SFLSIZ indicates the total number of records to be included in the subfile (17).
- 12** SFLPAG indicates the total number of records in a page (17).
- 13** SFLCLR indicates when the subfile should be cleared (when indicator 61 is on).
- 14** SFLDSP indicates when to display the subfile (when indicator 62 is on).
- 15** SFLDSPCTL indicates when to display the subfile control record (when indicator 62 is on).
- 16** The LOCK keyword prevents the workstation user from using the keyboard when the CONTROL1 record format is initially displayed.
- 17** HELP allows the user to press the Help key and sets indicator 99 on.
- 18** SFLMSG identifies the constant as a message that is displayed if indicator 99 is on.

In addition to the control information, the subfile control record format defines the constants to be used as column headings for the subfile record format. Refer to Figure 141 on page 552 for an example of the subfile control record format.

Accessing Single Device Files and Multiple Device Files

A **single device file** is a device file created with only one program device defined for it. Printer files, diskette files and tape files are single device files. Display files and Intersystem Communication Function (ICF) files created with a maximum number of one program device are also single device files.

A **multiple device file** is either a display file or an Intersystem Communications Function (ICF) file. A multiple device file can acquire more than one program device. For an example of the use of multiple device files, see Figure 145 on page 558.

A display file can have multiple program devices when the MAXDEV parameter of the CRTDSPF command is greater than 1. If you specify *NONE for the DEV parameter of this command, you must supply the name of a display device *before* you use any fields that are related to the file.

For more information about how to create and use a display file, refer to the
 # *Database and File Systems* category in the **i5/OS Information Center** at this Web site
 # [-http://www.ibm.com/systems/i/infocenter/](http://www.ibm.com/systems/i/infocenter/).

ICF files can have multiple program devices when the MAXPGMDEV parameter of the CRTICFF command is greater than 1. For more information about how to create and use ICF files, see the *ICF Programming*.

ILE COBOL determines at run time whether a file is a single device file or a multiple device file, based on whether the file is *capable* of having multiple devices. The actual number of devices acquired does not affect whether a file is considered a single or multiple device file. Whether a file is a single or a multiple device file is *not* determined at compilation time; this determination is based on the current description of the display or ICF file.

For multiple device files, if a particular program device is to be used in an I/O statement, that device is specified by the TERMINAL phrase. The TERMINAL phrase can also be specified for a single device file.

The following pages contain an example illustrating the use of multiple device files. The program uses a display file, and is intended to be run in batch mode. The program acquires terminals and invites those terminals using a sign-on display. After the terminals are invited, they are polled. If nobody signs on before the wait time expires, the program ends. If you enter a valid password, you are allowed to update an employee file by calling another ILE COBOL program. Once the update is complete, the device is invited again and the terminals are polled again.

```

.....+.....1.....+.....2.....+.....3.....+.....4.....+.....5.....+.....6.....+.....7.....+.....
A*          THIS IS THE MULTIPLE DEVICE DISPLAY FILE
A*
A*          DDS FOR THE MIXED FILE MULT
A
A          R SIGNON          INVITE 1
A          0 5 20'          '
A          DSPATR(RI)
A          0 6 20'          '
A          DSPATR(RI)
A          0 6 38'          '
A          DSPATR(RI)
A          0 7 20'          '
A          DSPATR(RI)
A          0 7 27'M D F'
A          DSPATR(HI BL)
A          0 7 38'          '
A          DSPATR(RI)
A          0 9 20'          '
A          DSPATR(RI)
A          0 20 20'PLEASE LOGON'
A          DSPATR(HI)
A          PASSWORD 10A I 20 43DSPATR(PC ND)
A          WRONG    20A 0 21 43
A
A          R UPDATE
A          0 3 5'UPDATE OF PERSONNEL FILE'
A          DSPATR(BL)
A          0 7 5'TYPE IN EMPLOYEE NUMBER TO BE +
A          UPDATED'
A          NUM       7A I 7 44DSPATR(RI PC)
A
A          R EMPLOYEE
A          0 3 5'EMPLOYEE NUMBER'
A          NUM       7A B 3 25DSPATR(PC)
A          0 5 5'EMPLOYEE NAME'
A          NAME      30A B 5 25DSPATR(PC)
A          0 7 5'EMPLOYEE ADDRESS'
A          0 9 5'STREET'
A          STREET    30A B 9 25DSPATR(PC)
A          0 11 5'APARTMENT NUMBER'
A          APTNO     5A B 11 25DSPATR(PC)
A          0 13 5'CITY'
A          CITY      20A B 13 25DSPATR(PC)
A          0 15 5'PROVINCE'
A          PROV      20A B 15 25DSPATR(PC)
A
A          R RECOVERY
A          0 3 5'THE EMPLOYEE NUMBER YOU HAVE GIVEN
A          IS INVALID'
A          0 6 5'TYPE Y TO RETRY'
A          0 8 5'TYPE N TO EXIT'
A          ANSWER    1X I 10 5DSPATR(RI PC)
A          VALUES('Y' 'N')

```

Figure 142. Example of the Use of Multiple Device Files ** Display File

1 The format SIGNON has the keyword INVITE associated with it. This means that, if format SIGNON is used in a WRITE statement, the device to which it is writing will be invited.

```

.....+.....1.....+.....2.....+.....3.....+.....4.....+.....5.....+.....6.....+.....7.....+.....
A** DDS FOR THE PHYSICAL FILE PASSWORD
A*
A*
A
A          R PASSWORDS          UNIQUE
A          PASKEY          10
A          PASSWORD          10
A          K PASKEY
A

```

Figure 143. Example of the Use of Multiple Device Files ** Physical File PASSWORD

```

.....+.....1.....+.....2.....+.....3.....+.....4.....+.....5.....+.....6.....+.....7.....+.....
A* DDS FOR THE PHYSICAL FILE TERM
A* WHICH CONTAINS THE LIST OF TERMINALS
A*
A
A          R TERM
A          TERM          10

```

Figure 144. Example of the Use of Multiple Device Files ** Physical File TERM

```

Source
STMT PL SEQNBR -A 1 B.+...2...+...3...+...4...+...5...+...6...+...7..IDENTFCN S COPYNAME CHG DATE
1 000100 IDENTIFICATION DIVISION.
2 000200 PROGRAM-ID. SAMPMDF.
000300
000400*****
000500* THE FOLLOWING PROGRAM DEMONSTRATES SOME OF THE FUNCTIONS *
000600* AVAILABLE WITH MULTIPLE DEVICE FILE SUPPORT. *
000700*****
000800
3 000900 ENVIRONMENT DIVISION.
4 001000 CONFIGURATION SECTION.
5 001100 SOURCE-COMPUTER. IBM-ISERIES
6 001200 OBJECT-COMPUTER. IBM-ISERIES
7 001300 SPECIAL-NAMES. ATTRIBUTE-DATA IS ATTR. 1
9 001400 INPUT-OUTPUT SECTION.
10 001500 FILE-CONTROL.
11 001600 SELECT MULTIPLE-FILE
12 001700 ASSIGN TO WORKSTATION-MULT
13 001800 ORGANIZATION IS TRANSACTION 2
14 001900 ACCESS MODE IS SEQUENTIAL
15 002000 FILE STATUS IS MULTIPLE-FS1, MULTIPLE-FS2 3
16 002100 CONTROL-AREA IS MULTIPLE-CONTROL-AREA. 4
002200
17 002300 SELECT TERMINAL-FILE
18 002400 ASSIGN TO DATABASE-TERM
19 002500 ORGANIZATION IS SEQUENTIAL
20 002600 ACCESS IS SEQUENTIAL
21 002700 FILE STATUS IS TERMINAL-FS1.
002800
22 002900 SELECT PASSWORD-FILE
23 003000 ASSIGN TO DATABASE-PASSWORD
24 003100 ORGANIZATION IS INDEXED
25 003200 RECORD KEY IS EXTERNALLY-DESCRIBED-KEY
26 003300 ACCESS MODE IS RANDOM
27 003400 FILE STATUS IS PASSWORD-FS1.
003500
28 003600 SELECT PRINTER-FILE
29 003700 ASSIGN TO PRINTER-QPRINT.
003800
30 003900 DATA DIVISION.
31 004000 FILE SECTION.
32 004100 FD MULTIPLE-FILE.
33 004200 01 MULTIPLE-REC.
004200 COPY DDS-SIGNON OF MULT. 5
34 +000001 05 MULT-RECORD PIC X(20). SIGNON
+000002* INPUT FORMAT:SIGNON FROM FILE MULT OF LIBRARY CBLGUIDE SIGNON
+000003* SIGNON
35 +000004 05 SIGNON-I REDEFINES MULT-RECORD. SIGNON
36 +000005 06 PASSWORD PIC X(10). 6 SIGNON
+000006* OUTPUT FORMAT:SIGNON FROM FILE MULT OF LIBRARY CBLGUIDE SIGNON
+000007* SIGNON
37 +000008 05 SIGNON-O REDEFINES MULT-RECORD. SIGNON
38 +000009 06 WRONG PIC X(20). SIGNON
004300

```

Figure 145. ILE COBOL Source Listing for Multiple Device File Support (Part 1 of 5)

```

5722WDS V5R4M0 060210 LN IBM ILE COBOL CBLGUIDE/SAMPMDF ISERIES1 06/02/15 15:04:02 Page 3
STMT PL SEQNBR -A 1 B.+...2...+...3...+...4...+...5...+...6...+...7..IDENTFCN S COPYNAME CHG DATE
39 004400 FD TERMINAL-FILE.
40 004500 01 TERMINAL-REC.
004500 COPY DDS-ALL-FORMATS OF TERM.
41 +000001 05 TERM-RECORD PIC X(10). <-ALL-FMTS
+000002* I-O FORMAT:TERM FROM FILE TERM OF LIBRARY CBLGUIDE <-ALL-FMTS
+000003* <-ALL-FMTS
42 +000004 05 TERM REDEFINES TERM-RECORD. <-ALL-FMTS
43 +000005 06 TERM PIC X(10). <-ALL-FMTS
004600
44 004700 FD PASSWORD-FILE.
45 004800 01 PASSWORD-REC.
004800 COPY DDS-ALL-FORMATS OF PASSWORD.
46 +000001 05 PASSWORD-RECORD PIC X(20). <-ALL-FMTS
+000002* I-O FORMAT:PASSWORDS FROM FILE PASSWORD OF LIBRARY CBLGUIDE <-ALL-FMTS
+000003* <-ALL-FMTS
+000004*THE KEY DEFINITIONS FOR RECORD FORMAT PASSWORDS <-ALL-FMTS
+000005* NUMBER NAME RETRIEVAL ALTSEQ <-ALL-FMTS
+000006* 0001 PASSKEY NAME ASCENDING NO <-ALL-FMTS
47 +000007 05 PASSWORDS REDEFINES PASSWORD-RECORD. <-ALL-FMTS
48 +000008 06 PASSKEY PIC X(10). <-ALL-FMTS
49 +000009 06 PASSWORD PIC X(10). <-ALL-FMTS
004900
50 005000 FD PRINTER-FILE.
51 005100 01 PRINTER-REC.
52 005200 05 PRINTER-RECORD PIC X(132).
005300
53 005400 WORKING-STORAGE SECTION.
005500
005600*****
005700* DECLARE THE FILE STATUS FOR EACH FILE *
005800*****
005900
54 006000 01 MULTIPLE-FS1 PIC X(2) VALUE SPACES.
55 006100 01 MULTIPLE-FS2. 7
56 006200 05 MULTIPLE-MAJOR PIC X(2) VALUE SPACES.
57 006300 05 MULTIPLE-MINOR PIC X(2) VALUE SPACES.
58 006400 01 TERMINAL-FS1 PIC X(2) VALUE SPACES.
59 006500 01 PASSWORD-FS1 PIC X(2) VALUE SPACES.
006600
006700*****
006800* DECLARE STRUCTURE FOR HOLDING FILE ATTRIBUTES *
006900*****
007000
60 007100 01 STATION-ATTR.
61 007200 05 STATION-TYPE PIC X(1). 8
62 007300 05 STATION-SIZE PIC X(1).
63 007400 05 STATION-LOC PIC X(1).
64 007500 05 FILLER PIC X(1).
65 007600 05 STATION-ACQUIRE PIC X(1).
66 007700 05 STATION-INVITE PIC X(1).
67 007800 05 STATION-DATA PIC X(1).
68 007900 05 STATION-STATUS PIC X(1).
69 008000 05 STATION-DISPLAY PIC X(1).
70 008100 05 STATION-KEYBOARD PIC X(1).
71 008200 05 STATION-SIGNON PIC X(1).

```

Figure 145. ILE COBOL Source Listing for Multiple Device File Support (Part 2 of 5)

```

STMT PL SEQNBR -A 1 B.+...2...+...3...+...4...+...5...+...6...+...7..IDENTFCN S COPYNAME CHG DATE
72 008300 05 FILLER PIC X(5).
008400
008500*****
008600* DECLARE THE CONTROL AREA FOR MULTIPLE-FILE *
008700*****
008800
73 008900 01 MULTIPLE-CONTROL-AREA.
74 009000 05 MULTIPLE-KEY-FEEDBACK PIC X(2) VALUE SPACES.
75 009100 05 MULTIPLE-DEVICE-NAME PIC X(10) VALUE SPACES.
76 009200 05 MULTIPLE-FORMAT-NAME PIC X(10) VALUE SPACES.
009300
009400*****
009500* DECLARE ERROR REPORT VARIABLES *
009600*****
009700
77 009800 01 HEADER-LINE.
78 009900 05 FILLER PIC X(60) VALUE SPACES.
79 010000 05 FILLER PIC X(72)
010100 VALUE "MDF ERROR REPORT".
80 010200 01 DETAIL-LINE.
81 010300 05 FILLER PIC X(15) VALUE SPACES.
82 010400 05 DESCRIPTION PIC X(25) VALUE SPACES.
83 010500 05 DETAIL-VALUE PIC X(92) VALUE SPACES.
010600
010700*****
010800* DECLARE COUNTERS, FLAGS AND STORAGE VARIABLES *
010900*****
011000
84 011100 01 CURRENT-TERMINAL PIC X(10) VALUE SPACES.
85 011200 01 TERMINAL-ARRAY.
86 011300 05 LIST-OF-TERMINALS OCCURS 250 TIMES.
87 011400 07 DEVICE-NAME PIC X(10).
88 011500 01 COUNTER PIC 9(3) VALUE IS 1.
89 011600 01 NO-OF-TERMINALS PIC 9(3) VALUE IS 1.
90 011700 01 TERMINAL-LIST-FLAG PIC 1.
91 011800 88 END-OF-TERMINAL-LIST VALUE IS B"1".
92 011900 88 NOT-END-OF-TERMINAL-LIST VALUE IS B"0".
93 012000 01 NO-DATA-FLAG PIC 1.
94 012100 88 NO-DATA-AVAILABLE VALUE IS B"1".
95 012200 88 DATA-AVAILABLE VALUE IS B"0".
012300
96 012400 PROCEDURE DIVISION.
012500
97 012600 DECLARATIVES.
012700
012800 MULTIPLE-SECTION SECTION.
012900 USE AFTER STANDARD EXCEPTION PROCEDURE ON MULTIPLE-FILE.
013000
013100 MULTIPLE-PARAGRAPH.
98 013200 WRITE PRINTER-REC FROM HEADER-LINE AFTER ADVANCING PAGE.
99 013300 MOVE "FILE NAME IS:" TO DESCRIPTION OF DETAIL-LINE.
100 013400 MOVE "MULTIPLE FILE" TO DETAIL-VALUE OF DETAIL-LINE.
101 013500 WRITE PRINTER-REC FROM DETAIL-LINE AFTER ADVANCING 5 LINES.
102 013600 MOVE "FILE STATUS IS:" TO DESCRIPTION OF DETAIL-LINE.
103 013700 MOVE MULTIPLE-FS1 TO DETAIL-VALUE OF DETAIL-LINE.

```

Figure 145. ILE COBOL Source Listing for Multiple Device File Support (Part 3 of 5)

```

5722WDS V5R4M0 060210 LN IBM ILE COBOL CBLGUIDE/SAMPMDF ISERIES1 06/02/15 15:04:02 Page 5
STMT PL SEQNBR -A 1 B..+...2...+...3...+...4...+...5...+...6...+...7..IDENTFCN S COPYNAME CHG DATE
104 013800 WRITE PRINTER-REC FROM DETAIL-LINE AFTER ADVANCING 2 LINES.
105 013900 MOVE "EXTENDED STATUS IS:" TO DESCRIPTION OF DETAIL-LINE. 9
106 014000 MOVE MULTIPLE-FS2 TO DETAIL-VALUE OF DETAIL-LINE.
107 014100 WRITE PRINTER-REC FROM DETAIL-LINE AFTER ADVANCING 2 LINES.
108 014200 ACCEPT STATION-ATTR FROM ATTR. 10
109 014300 MOVE "FILE ATTRIBUTES ARE:" TO DESCRIPTION OF DETAIL-LINE.
110 014400 MOVE STATION-ATTR TO DETAIL-VALUE OF DETAIL-LINE.
111 014500 WRITE PRINTER-REC FROM DETAIL-LINE AFTER ADVANCING 2 LINES.
112 014600 STOP RUN.
014700
014800 TERMINAL-SECTION SECTION.
014900 USE AFTER STANDARD EXCEPTION PROCEDURE ON TERMINAL-FILE.
015000 TERMINAL-PARAGRAPH.
113 015100 WRITE PRINTER-REC FROM HEADER-LINE AFTER ADVANCING PAGE.
114 015200 MOVE "FILE NAME IS:" TO DESCRIPTION OF DETAIL-LINE.
115 015300 MOVE "TERMINAL FILE" TO DETAIL-VALUE OF DETAIL-LINE.
116 015400 WRITE PRINTER-REC FROM DETAIL-LINE AFTER ADVANCING 5 LINES.
117 015500 MOVE "FILE STATUS IS:" TO DESCRIPTION OF DETAIL-LINE.
118 015600 MOVE TERMINAL-FS1 TO DETAIL-VALUE OF DETAIL-LINE.
119 015700 WRITE PRINTER-REC FROM DETAIL-LINE AFTER ADVANCING 2 LINES.
120 015800 STOP RUN.
015900
016000 PASSWORD-SECTION SECTION.
016100 USE AFTER STANDARD EXCEPTION PROCEDURE ON PASSWORD-FILE.
016200 PASSWORD-PARAGRAPH.
121 016300 WRITE PRINTER-REC FROM HEADER-LINE AFTER ADVANCING PAGE.
122 016400 MOVE "FILE NAME IS:" TO DESCRIPTION OF DETAIL-LINE.
123 016500 MOVE "PASSWORD FILE" TO DETAIL-VALUE OF DETAIL-LINE.
124 016600 WRITE PRINTER-REC FROM DETAIL-LINE AFTER ADVANCING 5 LINES.
125 016700 MOVE "FILE STATUS IS:" TO DESCRIPTION OF DETAIL-LINE.
126 016800 MOVE PASSWORD-FS1 TO DETAIL-VALUE OF DETAIL-LINE.
127 016900 WRITE PRINTER-REC FROM DETAIL-LINE AFTER ADVANCING 2 LINES.
128 017000 STOP RUN.
017100
017200 END DECLARATIVES.
017300
017400*****
017500* MAIN PROGRAM LOGIC BEGINS HERE *
017600*****
017700
017800 MAIN-PROGRAM SECTION.
017900 MAINLINE.
129 018000 OPEN I-O MULTIPLE-FILE 11
018100 INPUT TERMINAL-FILE
018200 I-O PASSWORD-FILE
018300 OUTPUT PRINTER-FILE.
018400
130 018500 MOVE 1 TO COUNTER.
131 018600 SET NOT-END-OF-TERMINAL-LIST TO TRUE.
018700*****
018800* Fill Terminal List
018900*****
132 019000 PERFORM UNTIL END-OF-TERMINAL-LIST
133 019100 READ TERMINAL-FILE RECORD
019200 INTO LIST-OF-TERMINALS(COUNTER)

```

Figure 145. ILE COBOL Source Listing for Multiple Device File Support (Part 4 of 5)

```

5722WDS V5R4M0 060210 LN  IBM ILE COBOL          CBLGUIDE/SAMPMPDF      ISERIES1  06/02/15 15:04:02      Page      6
STMT PL SEQNBR -A 1 B.+...2....+...3....+...4....+...5....+...6....+...7..IDENTFCN  S COPYNAME  CHG DATE
019300          AT END
134  019400          SET END-OF-TERMINAL-LIST TO TRUE
135  019500          SUBTRACT 1 FROM COUNTER
136  019600          MOVE COUNTER TO NO-OF-TERMINALS
019700          END-READ
137  019800          ADD 1 TO COUNTER
019900          END-PERFORM.
020000*****
020100* Acquire and invite terminals
020200*****
138  020300          PERFORM VARYING COUNTER FROM 1 BY 1
020400          UNTIL COUNTER GREATER THAN NO-OF-TERMINALS
139  020500          ACQUIRE LIST-OF-TERMINALS(COUNTER) FOR MULTIPLE-FILE 12
140  020600          WRITE MULTIPLE-REC 13
020700          FORMAT IS "SIGNON"
020800          TERMINAL IS LIST-OF-TERMINALS(COUNTER)
020900          END-WRITE
021000          END-PERFORM.
021100
141  021200          MOVE 1 TO COUNTER.
142  021300          SET DATA-AVAILABLE TO TRUE.
021400*****
021500* Poll terminals
021600*****
143  021700          PERFORM UNTIL NO-DATA-AVAILABLE
144  021800          READ MULTIPLE-FILE RECORD 14
145  021900          IF MULTIPLE-FS2 EQUAL "310" THEN
146  022000          SET NO-DATA-AVAILABLE TO TRUE 15
022100          END-IF
147  022200          IF DATA-AVAILABLE THEN
148  022300          MOVE MULTIPLE-DEVICE-NAME TO CURRENT-TERMINAL
022400*****
022500* Validate Password 16
022600*****
149  022700          MOVE CURRENT-TERMINAL TO PASSKEY OF PASSWORD-REC
150  022800          READ PASSWORD-FILE RECORD
151  022900          IF PASSWORD OF SIGNON-I EQUAL
023000          PASSWORD OF PASSWORD-REC THEN
152  023100          CALL "UPDT" USING CURRENT-TERMINAL
153  023200          MOVE SPACES TO WRONG OF SIGNON-O
023300          ELSE
154  023400          MOVE "INVALID PASSWORD" TO WRONG OF SIGNON-O
023500          END-IF
155  023600          WRITE MULTIPLE-REC
023700          FORMAT IS "SIGNON"
023800          TERMINAL IS CURRENT-TERMINAL
023900          END-WRITE
024000          END-IF
024100          END-PERFORM.
024200*****
024300* Drop terminals
024400*****
156  024500          PERFORM VARYING COUNTER FROM 1 BY 1
024600          UNTIL COUNTER GREATER THAN NO-OF-TERMINALS
157  024700          DROP LIST-OF-TERMINALS(COUNTER) FROM MULTIPLE-FILE 17
024800          END-PERFORM.
024900
158  025000          CLOSE MULTIPLE-FILE
025100          TERMINAL-FILE
025200          PASSWORD-FILE
025300          PRINTER-FILE.
159  025400          STOP RUN.
025500
          * * * * *  E N D   O F   S O U R C E  * * * * *

```

Figure 145. ILE COBOL Source Listing for Multiple Device File Support (Part 5 of 5)

- 1 ATTR is the mnemonic-name associated with the function-name ATTRIBUTE-DATA. ATTR is used in the ACCEPT statement to obtain attribute data for the TRANSACTION file MULTIPLE-FILE. See item 10.
- 2 File MULT must have been created using the CRTDSPF command, where the DEV parameter has a value of *NONE and the MAXDEV parameter has a value greater than 1. The WAITRCD parameter specifies the wait time for READ operations on the file. The WAITRCD parameter must have a value greater than 0.

- 3** MULTIPLE-FS2 is the extended file status for the TRANSACTION file MULTIPLE-FILE. This variable has been declared in the WORKING-STORAGE section of the program. See item **7**.
- 4** MULTIPLE-CONTROL-AREA is the control area for the TRANSACTION file MULTIPLE-FILE. This variable is used to determine which program device was used to sign on. See item **15**.
- 5** The data description for MULTIPLE-REC has been defined using the COPY DDS statement.

Note: Only the fields that are copied are named fields. Refer to the DDS of this example for comments regarding the DDS used.
- 6** Format SIGNON is the format with the INVITE keyword. This is the format that will be used to invite devices via the WRITE statement.
- 7** This is the declaration for the extended file-status MULTIPLE-FS2. It is a 4-byte field that is subdivided into a major return code (first 2 bytes) and a minor return code (last 2 bytes).
- 8** STATION-ATTR is where the ACCEPT statement stores the attribute data for the TRANSACTION file MULTIPLE-FILE. See item **10**.
- 9** In this statement, the extended file status MULTIPLE-FS2 is being written.
- 10** This statement accepts attribute data for the TRANSACTION file MULTIPLE-FILE. Since the FOR phrase is not specified with the ACCEPT statement, the last program device is used.
- 11** This statement opens the TRANSACTION file MULTIPLE-FILE. Because the ACQPGMDEV parameter of the CRTDSPF command has the value *NONE, no program devices are implicitly acquired when this file is opened.
- 12** This statement acquires the program device contained in the variable LIST-OF-TERMINALS (COUNTER), for the TRANSACTION file MULTIPLE-FILE.
- 13** This WRITE statement is inviting the program device specified in the TERMINAL phrase. The format SIGNON has the DDS keyword INVITE associated with it. Refer to item **14**.
- 14** This READ statement will read from any invited program device. See item **13**. If the wait time expires before anyone inputs to the invited devices, the extended file status will be set to "0310" and processing will continue. See item **15**.
- 15** In this statement, the extended file status for MULTIPLE-FILE is being checked to see if the wait time expired.
- 16** The program device name stored in the control area is used to determine which program device was used to sign on. See item **4**.
- 17** This DROP statement detaches the program device contained in the variable LIST-OF-TERMINALS from the TRANSACTION file MULTIPLE-FILE.

Writing Programs That Use Subfile Transaction Files

Typically, you use a subfile TRANSACTION file to read a group of records from or write a group of record to a display device. To use a subfile TRANSACTION file in an ILE COBOL program, you must:

- Name the file through a file control entry in the FILE-CONTROL paragraph of the Environment Division
- Describe the file through a file description entry in the Data Division
- Use extensions to Procedure Division statements that support transaction processing.

Naming a Subfile Transaction File

To use a subfile TRANSACTION file in your ILE COBOL program, you must name the file through a file control entry in the FILE-CONTROL paragraph. See the *IBM Rational Development Studio for i: ILE COBOL Reference* for a full description of the FILE-CONTROL paragraph.

You name the TRANSACTION file in the FILE-CONTROL paragraph as follows:

```
FILE-CONTROL.
  SELECT transaction-file-name
  ASSIGN TO WORKSTATION-display_file_name
  ORGANIZATION IS TRANSACTION
  ACCESS MODE IS DYNAMIC
  RELATIVE KEY IS relative-key-data-item
  CONTROL AREA IS control-area-data-item.
```

You use the SELECT clause to choose a file. This file must be identified by a FD entry in the Data Division.

You use the ASSIGN clause to associate the TRANSACTION file with a display file. You must specify a device type of WORKSTATION in the ASSIGN clause to use TRANSACTION files. If you want to use a separate indicator area for this TRANSACTION file, you need to include the -SI attribute with the ASSIGN clause. See “Using Indicators with Transaction Files” on page 536 for further details of how to use the separate indicator area.

You must specify ORGANIZATION IS TRANSACTION in the file control entry in order to use a TRANSACTION file. This clause tells your ILE COBOL program that it will be interacting with a workstation user or another system.

You access a subfile TRANSACTION file dynamically. Dynamic access allows you to read or write records to the file sequentially or randomly, depending on the form of the specific input-output request. Subfiles are the only TRANSACTION files that can be accessed randomly. You use the ACCESS MODE clause in the file control entry to tell your ILE COBOL program how to access the TRANSACTION file. You must specify ACCESS MODE IS DYNAMIC to read or write to the subfile TRANSACTION file.

When using subfiles, you must provide a relative key. Use the RELATIVE KEY clause to identify the relative key data item. The relative key data item specifies the relative record number for a specific record in a subfile.

If you want feedback on the status of an input/output request that refers to a TRANSACTION file, you define a status key data item in the file control entry using the FILE STATUS clause. When you specify the FILE STATUS clause, the system moves a value into the status key data item after each input-output request that explicitly or implicitly refers to the TRANSACTION file. The value indicates the status of the execution of the I-O statement.

You can obtain specific device-dependent and system dependent information that is used to control input-output operations for TRANSACTION files by identifying

a control area data item using the CONTROL-AREA clause. You can define the data item specified by the CONTROL-AREA clause in the LINKAGE SECTION or WORKING-STORAGE SECTION with the following format:

```
01 control-area-data-item.  
   05 function-key          PIC X(2).  
   05 device-name          PIC X(10).  
   05 record-format        PIC X(10).
```

The control area can be 2, 12, or 22 bytes long. Thus, you can specify only the first 05-level element, the first two 05-level elements, or all three 05-level elements, depending of the type of information you are looking for.

The control area data item will allow you to identify:

- The function key that the operator pressed to initiate a transaction
- The name of the program device used
- The name of the DDS record format that was referenced by the last I-O statement.

Describing a Subfile Transaction File

To use a TRANSACTION file in your ILE COBOL program, you must describe the file through a file description entry in the Data Division. See the *IBM Rational Development Studio for i: ILE COBOL Reference* for a full description of the File Description Entry. Use the Format 6 File Description Entry to describe a TRANSACTION file.

A file description entry in the Data Division that describes a TRANSACTION file looks as follows:

```
FD CUST-DISPLAY.  
01 DISP-REC.  
   COPY DDS-ALL-FORMATS OF CUSMINQ.
```

In ILE COBOL, TRANSACTION files are usually externally described. Create a DDS for the TRANSACTION file you want to use. Refer to “Defining Transaction Files Using Data Description Specifications” on page 521 for how to create a DDS. Then create the TRANSACTION file.

Once you have created the DDS for the TRANSACTION file and the TRANSACTION file, use the Format 2 COPY statement to describe the layout of the TRANSACTION file data record. When you compile your ILE COBOL program, the Format 2 COPY will create the Data Division statements to describe the TRANSACTION file. Use the DDS-ALL-FORMATS option of the Format 2 COPY statement to generate one storage area for all formats.

Processing a Subfile Transaction File

The following is a list of all of the Procedure Division statements that have extensions specifically for processing TRANSACTION files in an ILE COBOL program. See the *IBM Rational Development Studio for i: ILE COBOL Reference* for a detailed discussion of each of these statements.

- ACCEPT Statement - Format 6
- ACQUIRE Statement
- CLOSE Statement - Format 1
- DROP Statement
- OPEN Statement - Format 3
- READ Statement - Format 5 (Subfile)
- REWRITE Statement - Format 2 (Subfile)

- WRITE Statement - Format 5 (Subfile).

Opening a Subfile Transaction File

To process a TRANSACTION file in the Procedure Division, you must first open the file. You use the Format 3 OPEN statement to open a TRANSACTION file. A TRANSACTION file must be opened in I-O mode.

```
OPEN I-O file-name.
```

Acquiring Program Devices

You must acquire a program device for the TRANSACTION file. Once acquired, the program device is available for input and output operations. You can acquire a program device implicitly or explicitly.

You implicitly acquire one program device when you open the TRANSACTION file. If the file is a display file, the single implicitly acquired program device is determined by the first entry in the DEV parameter of the CRTDSPF command. Additional program devices must be explicitly acquired.

You explicitly acquire a program device by using the ACQUIRE statement. For display files, the device named in the ACQUIRE statement does not have to be specified in the DEV parameter of the CRTDSPF command, CHGDSPF command, or the OVRDSPF command. However, when you create the display file, you must specify the number of devices that may be acquired (the default is one). For a display file, the program device name must match the display device.

```
ACQUIRE program-device-name FOR transaction-file-name.
```

Writing to a Subfile Transaction File

Once you have opened the TRANSACTION file and acquired a program device for it, you are now ready to perform input and output operations on it.

The first input/output operation you typically perform on a TRANSACTION file is to write a record to the display. This record is used to prompt the user to enter a response or some data.

You use the Format 5 WRITE statement to write a logical record to the subfile TRANSACTION file. You simply code the WRITE statement as follows:

```
WRITE SUBFILE record-name FORMAT IS format-name.
```

In some situations, you may have multiple data records, each with a different format, that you want active for a TRANSACTION file. In this case, you must use the FORMAT phrase of the Format 5 WRITE statement to specify the format of the output data record you want to write to the TRANSACTION file.

If you have explicitly acquired multiple program devices for the TRANSACTION file, you must use the TERMINAL phrase of the Format 5 WRITE statement to specify the program device's subfile to which you want the output record to be sent.

```
WRITE SUBFILE record-name  
      FORMAT IS format-name  
      TERMINAL IS program-device-name  
END-WRITE.
```

Before or after filling the subfile TRANSACTION file with records using the Format 5 WRITE statement, you can write the subfile control record to the program device using the Format 4 WRITE statement. Refer to "Writing to a Transaction File" on page 527 for a description of how to use the Format 4 WRITE

statement to write to a TRANSACTION file. Writing the subfile control record could cause the display of either the subfile control record, the subfile records, or both the subfile control record and subfile records.

Reading from a Subfile Transaction File

You use the Format 4 READ statement to read a subfile control record. Refer to “Reading from a Transaction File” on page 527 for a description of how to use the Format 4 READ statement to read to a TRANSACTION file. Reading the subfile control record physically transfers records from the program device so that they can be made available to the subfile.

Once the records are available to the subfile, you use the Format 5 READ statement to read a specified record from the subfile TRANSACTION file. The Format 5 READ statement can only be used to read a format that is a subfile record; it cannot be used for communications devices.

Before you use the READ statement, you must have acquired at least one program device for the TRANSACTION file. If a READ statement is performed and there are no acquired program devices, a logic error is reported by setting the file status to 92.

You can read a subfile sequentially or randomly.

#

To read a subfile sequentially, you must specify the NEXT MODIFIED phrase in the Format 5 READ statement. When the NEXT MODIFIED phrase is specified, the record made available is the first record in the subfile that has been modified. For information about how a subfile record is marked as being modified, refer to the *Database and File Systems* category in the **i5/OS Information Center** at this Web site -<http://www.ibm.com/systems/i/infocenter/>.

If there are no next modified subfile records available, the AT END condition exists, the file status is set to 12, and the value of the RELATIVE KEY data item is set to the key of the last record in the subfile.

When reading a subfile sequentially, you should also specify the AT END phrase in the Format 5 READ statement. The AT END phrase allows you to specify an imperative statement to be executed when the AT END condition arises.

```
READ SUBFILE subfile-name NEXT MODIFIED RECORD
      AT END imperative-statement
END-READ
```

To read a subfile randomly, you must specify, in the RELATIVE KEY data item, the relative record number of the subfile record you want to read and you must not specify the NEXT MODIFIED phrase in the Format 5 READ statement. When the NEXT MODIFIED phrase is not specified, the record made available is the record in the subfile with a relative key record number that corresponds to the value of the RELATIVE KEY data item. If the RELATIVE KEY data item, at the time that the READ statement is performed, contains a value that does not correspond to a relative record number for the subfile, the INVALID KEY condition exists.

When reading a subfile randomly, you should also specify the INVALID KEY phrase in the Format 5 READ statement. The INVALID KEY phrase allows you to specify an imperative statement to be executed when the INVALID KEY condition arises.

```
READ SUBFILE subfile-name RECORD
      INVALID KEY imperative-statement
END-READ
```

For a detailed explanation of how the READ operation is performed, refer to the section on the READ statement in the *IBM Rational Development Studio for i: ILE COBOL Reference*.

In those cases where you have acquired multiple program devices, you can explicitly specify the program device from which you read data by identifying it in the TERMINAL phrase of the READ statement.

In those cases where you want to receive the data in a specific format, you can identify this format in the FORMAT phrase of the READ statement. If the data available does not match the requested record format, a file status of 9K is set.

The following are examples of the READ statement with the TERMINAL and FORMAT phrases specified.

```
READ SUBFILE subfile-name RECORD
      FORMAT IS record-format
END-READ
READ SUBFILE subfile-name RECORD
      TERMINAL IS program-device-name
END-READ
READ SUBFILE subfile-name RECORD
      FORMAT IS record-format
      TERMINAL IS program-device-name
END-READ
```

Replacing (Rewriting) a Subfile Record

Once you have read and modified a subfile record, you can replace it in the subfile using the REWRITE statement.

```
REWRITE SUBFILE record-name
      FORMAT IS record-format
      TERMINAL IS program-device-name
END-REWRITE
```

The record replaced in the subfile is the record in the subfile accessed by the previous successful READ operation.

Dropping Program Devices

Once you have finished using a program device that you had previously acquired for a TRANSACTION file, you should drop it. Dropping a program device means that it will no longer be available for input or output operations through the TRANSACTION file. Dropping a program device makes it available to other applications. You can drop a program device implicitly or explicitly.

You implicitly drop all program devices attached to a TRANSACTION file when you close the file.

You explicitly drop a program device by indicating it in the DROP statement. The device, once dropped, can be re-acquired again, if necessary.

```
DROP program-device-name FROM transaction-file-name.
```

Closing a Subfile Transaction File

When you have finished using a subfile TRANSACTION file, you must close it. Use the Format 1 CLOSE statement to close the TRANSACTION file. Once you close the file, it cannot be processed again until it is opened again.

CLOSE transaction-file-name.

Example of Using WRITE SUBFILE in an Order Inquiry Program

Figure 149 on page 573 shows an example of an order inquiry program, ORDINQ, that uses subfiles. The associated DDS is also shown, except for the DDS for the customer master file, CUSMSTP. Refer to Figure 132 on page 531 for the DDS for CUSMSTP.

ORDINQ displays all the detail order records for the requested order number. The program prompts you to enter the order number that is to be reviewed. The order number is checked against the order header file, ORDHDRP. If the order number exists, the customer number accessed from the order header file is checked against the customer master file, CUSMSTP. All detail order records in ORDDTLP for the requested order are read and written to the subfile. A write for the subfile control record format is processed, and the detail order records in the subfile are displayed for you to review. You end the program by pressing F12.

```

.....+.....1.....+.....2.....+.....3.....+.....4.....+.....5.....+.....6.....+.....7.....+.....
A** PHYSICAL ORDDTLP      ORDER DETAIL FILE
A
A                               UNIQUE
A*
A      R ORDDTL           TEXT('ORDER DETAIL RECORD')
A*
A      CUST              5      CHECK(MF)
A                               COLHDG('CUSTOMER' 'NUMBER')
A*
A      ORDERN            5 0     COLHDG('ORDER' 'NUMBER')
A*
A      LINNUM            3 0
A                               COLHDG('LINE' 'NO')
A                               TEXT('LINE NUMBER OF LINE IN ORDER'
A                               )
A*
A      ITEM              5 0     CHECK(M10)
A                               COLHDG('ITEM' 'NUMBER')
A      QTYORD            3 0
A                               COLHDG('QUANTITY' 'ORDERED')
A                               TEXT('QUANTITY ORDERED')
A*
A      DESCRP            30      COLHDG('ITEM' 'DESCRIPTION')
A*
A      PRICE             6 2     CMP(GT 0)
A                               COLHDG('PRICE')
A                               TEXT('SELLING PRICE')
A                               EDTCDE(J)
A      EXTENS            8 2     COLHDG('EXTENSION')
A                               TEXT('EXTENSION AMOUNT OF QTYORD X
A                               PRICE')
A*
A      WHSLOC            3      CHECK(MF)
A                               COLHDG('BIN' 'NO.')
A*
A      ORDDAT            6 0     TEXT('DATE ORDER WAS ENTERED')
A*
A      CUSTYP            1 0     RANGE(1 5)
A                               COLHDG('CUST' 'TYPE')
A                               TEXT('CUSTOMER TYPE 1=GOV 2=SCH   +
A                               3=BUS 4=PVT 5=OT')
A*
A      STATE             2      CHECK(MF)
A                               COLHDG('STATE')
A*
A      ACTMTH            2 0     COLHDG('ACCT' 'MTH')
A                               TEXT('ACCOUNTING MONTH OF SALE')
A*
A      ACTYR             2 0     COLHDG('ACCT' 'YEAR')
A                               TEXT('ACCOUNTING YEAR OF SALE')
A
A      K ORDERN
A      K LINNUM

```

Figure 146. Data Description Specifications for an Order Inquiry Program - Order Detail File

	1	2	3	4	5	6	7
A*	ORDINQD						EXISTING ORDER REVIEW DISPLAY FILE
A							
A*							
A	R SUB1						SFL
A	ITEM	5	0	10	2		TEXT('ITEM NUMBER')
A	QTYORD	3	0	10	9		TEXT('QUANTITY ORDERED')
A	DESCRP	30		10	14		TEXT('ITEM DESCRIPTION')
A	PRICE	6	2	10	46		TEXT('SELLING PRICE')
A	EXTENS	8	2	10	56		EDTCDE(J)
A							TEXT('EXTENSION AMOUNT OF QTYORD +
A							X PRICE')
A	R SUBCTL1						SFLCTL(SUB1)
A	58						SFLCLR
A	57						SFLDSP
A	N58						SFLDSPCTL
A							SFLSIZ(57)
A							SFLPAG(14)
A	57						SFLEND
A							OVERLAY
A							LOCK
A	N45						ROLLUP(97 'CONTINUE DISPLAY')
A	AON47						CA12(98 'END OF PROGRAM')
A							SETOFF(57 'DISPLAY SUBFILE')
A							SETOFF(58 'OFF = DISPLAY SUBCTL1 0+
A							N = CLEAR SUBFILE')
A					1	2	'EXISTING ORDER INQUIRY'
A					3	2	'ORDER'
A	ORDERN	5Y	0B	3	8		TEXT('ORDER NUMBER')
A	61						ERRMSG('ORDER NUMBER NOT FOUND' 61)
A	47						ERRMSG('NO LINE FOR THIS ORDER' 47)
A	62						ERRMSG('NO CUSTOMER RECORD' 62)
A					4	2	'DATE'
A	ORDDAT	6	0	4	7		TEXT('DATE ORDER WAS ENTERED')
A					5	2	'CUST #'
A	CUST	5		5	9		TEXT('CUSTOMER NUMBER')
A	NAME	25		3	16		TEXT('CUSTOMER NAME')
A	ADDR	20		4	16		TEXT('CUSTOMER ADDRESS')
A	CITY	20		5	16		TEXT('CUSTOMER CITY')
A	STATE	2		6	16		TEXT('CUSTOMER STATE')
A	ZIP	5	0	6	31		TEXT('ZIP CODE')
A					1	44	'TOTAL'
A	ORDAMT	8	2	1	51		TEXT('TOTAL AMOUNT OF ORDER')
A					2	44	'STATUS'
A	STSORD	12			2	51	
A					3	44	'OPEN'
A	STSOPN	12			3	51	
A					4	44	'CUSTOMER ORDER'
A	CUSORD	15		4	59		TEXT('CUSTOMER PURCHASE ORDER +
A							NUMBER')
A					5	44	'SHIP VIA'
A	SHPVIA	15		5	59		TEXT('SHIPPING INSTRUCTIONS')
A					6	44	'PRINTED DATE'
A	PRTDAT	6	0	6	57		TEXT('DATE ORDER WAS PRINTED')
A					7	29	'INVOICE'

Figure 147. Data Description Specifications for an Order Inquiry Program - Order Review File (Part 1 of 2)

A	INVNUM	5 0	7 38	TEXT('INVOICE NUMBER')
A			7 64	'MTH'
A	ACTMTH	2 0	7 68	TEXT('ACCOUNTING MONTH OF SALE')
A			7 72	'YEAR'
A	ACTYR	2 0	7 77	TEXT('ACCOUNTING YEAR OF SALE')
A			8 2	'ITEM'
A			8 8	'QTY'
A			8 14	'ITEM DESCRIPTION'
A			8 46	'PRICE'
A			8 55	'EXTENSION'

Figure 147. Data Description Specifications for an Order Inquiry Program - Order Review File (Part 2 of 2)

.....+.....1.....+.....2.....+.....3.....+.....4.....+.....5.....+.....6.....+.....7.....+.....				
A* THIS IS THE ORDER HEADER FILE ** ORDHDRP				
A				
A				UNIQUE
A	R ORDHDR			TEXT('ORDER HEADER RECORD')
A	CUST	5		TEXT('CUSTOMER NUMBER')
A	ORDERN	5 00		TEXT('ORDER NUMBER')
A	ORDDAT	6 00		TEXT('DATE ORDER ENTERED')
A	CUSORD	15		TEXT('CUSTOMER PURCHASE ORDER +
A				NUMBER')
A	SHPVIA	15		TEXT('SHIPPING INSTRUCTIONS')
A	ORDSTS	1 00		TEXT('ORDER SATAUS 1PCS 2CNT + 3CHK 4RDY 5PRT 6PCK')
A	OPRNAM	10		TEXT('OPERATOR WHO ENTERED ORD')
A	ORDAMT	8 02		TEXT('DOLLAR AMOUNT OF ORDER')
A	CUSTYP	1 00		TEXT('CUSTOMER TYPE 1=GOV 2=SCH +
A				3=BUS 4=PVT 5=OT')
A	INVNUM	5 00		TEXT('INVOICE NUMBER')
A	PRTDAT	6 00		TEXT('DATE ORDER WAS PRINTED')
A	OPNSTS	1 00		TEXT('ORDER OPEN STATUS 1=OPEN + 2= CLOSE 3=CANCEL')
A	TOTLIN	3 00		TEXT('TOTAL LINE ITEMS IN ORDER')
A	ACTMTH	2 00		TEXT('ACCOUNTING MONTH OF SALE')
A	ACTYR	2 00		TEXT('ACCOUNTING YEAR OF SALE')
A	STATE	2		TEXT('STATE')
A	AMPAID	8 02		TEXT('AMOUNT PAID')
K	ORDERN			

Figure 148. Data Description Specifications for an Order Inquiry Program - Order Header File

```

Source
STMT PL SEQNBR -A 1 B.+....2...+....3...+....4...+....5...+....6...+....7..IDENTFCN S COPYNAME CHG DATE
1 000100 IDENTIFICATION DIVISION.
2 000200 PROGRAM-ID. ORDINQ.
000300* SAMPLE ORDER INQUIRY PROGRAM
000400
3 000500 ENVIRONMENT DIVISION.
4 000600 CONFIGURATION SECTION.
5 000700 SOURCE-COMPUTER. IBM-ISERIES
6 000800 OBJECT-COMPUTER. IBM-ISERIES
7 000900 INPUT-OUTPUT SECTION.
8 001000 FILE-CONTROL.
9 001100 SELECT ORDER-HEADER-FILE
10 001200 ASSIGN TO DATABASE-ORDHDRP
11 001300 ORGANIZATION IS INDEXED
12 001400 ACCESS MODE IS RANDOM
13 001500 RECORD KEY IS ORDERN OF ORDER-HEADER-RECORD.
14 001600 SELECT ORDER-DETAIL-FILE
15 001700 ASSIGN TO DATABASE-ORDDTLP
16 001800 ORGANIZATION IS INDEXED
17 001900 ACCESS IS DYNAMIC
18 002000 RECORD KEY IS ORDER-DETAIL-RECORD-KEY.
19 002100 SELECT CUSTOMER-MASTER-FILE
20 002200 ASSIGN TO DATABASE-CUSMSTP
21 002300 ORGANIZATION IS INDEXED
22 002400 ACCESS IS RANDOM
23 002500 RECORD KEY IS CUST OF CUSTOMER-MASTER-RECORD.
24 002600 SELECT EXISTING-ORDER-DISPLAY-FILE
25 002700 ASSIGN TO WORKSTATION-ORDINQD
26 002800 ORGANIZATION IS TRANSACTION
27 002900 ACCESS IS DYNAMIC
28 003000 RELATIVE KEY IS SUBFILE-RECORD-NUMBER
29 003100 FILE STATUS IS STATUS-CODE-ONE.
003200
30 003300 DATA DIVISION.
31 003400 FILE SECTION.
32 003500 FD ORDER-HEADER-FILE.
33 003600 01 ORDER-HEADER-RECORD.
003700 COPY DDS-ORDHDR OF ORDHDRP.
+000001* I-O FORMAT:ORDHDR FROM FILE ORDHDRP OF LIBRARY CBLGUIDE ORDHDR
+000002* ORDER HEADER RECORD ORDHDR
+000003* USER SUPPLIED KEY BY RECORD KEY CLAUSE ORDHDR
34 000004 05 ORDHDR. ORDHDR
35 +000005 06 CUST PIC X(5). ORDHDR
+000006* CUSTOMER NUMBER ORDHDR
36 +000007 06 ORDERN PIC S9(5) COMP-3. ORDHDR
+000008* ORDER NUMBER ORDHDR
37 +000009 06 ORDDAT PIC S9(6) COMP-3. ORDHDR
+000010* DATE ORDER ENTERED ORDHDR
38 +000011 06 CUSORD PIC X(15). ORDHDR
+000012* CUSTOMER PURCHASE ORDER NUMBER ORDHDR
39 +000013 06 SHPVIA PIC X(15). ORDHDR
+000014* SHIPPING INSTRUCTIONS ORDHDR
40 +000015 06 ORDSTS PIC S9(1) COMP-3. ORDHDR
+000016* ORDER SATAUS 1PCS 2CNT 3CHK 4RDY 5PRT 6PCK ORDHDR

```

Figure 149. Example of an Order Inquiry Program (Part 1 of 10)

```

STMT PL SEQNBR -A 1 B.+. . . . 2. . . . 3. . . . 4. . . . 5. . . . 6. . . . 7. . . . IDENTFCN S COPYNAME CHG DATE
41 +000017 06 OPRNAM PIC X(10). ORDHDR
+000018* OPERATOR WHO ENTERED ORD ORDHDR
42 +000019 06 ORDAMT PIC S9(6)V9(2) COMP-3. ORDHDR
+000020* DOLLAR AMOUNT OF ORDER ORDHDR
43 +000021 06 CUSTYP PIC S9(1) COMP-3. ORDHDR
+000022* CUSTOMER TYPE 1=GOV 2=SCH 3=BUS 4=PVT 5=OT ORDHDR
44 +000023 06 INVNUM PIC S9(5) COMP-3. ORDHDR
+000024* INVOICE NUMBER ORDHDR
45 +000025 06 PRDAT PIC S9(6) COMP-3. ORDHDR
+000026* DATE ORDER WAS PRINTED ORDHDR
46 +000027 06 OPNSTS PIC S9(1) COMP-3. ORDHDR
+000028* ORDER OPEN STATUS 1=OPEN 2= CLOSE 3=CANCEL ORDHDR
47 +000029 06 TOTLIN PIC S9(3) COMP-3. ORDHDR
+000030* TOTAL LINE ITEMS IN ORDER ORDHDR
48 +000031 06 ACTMTH PIC S9(2) COMP-3. ORDHDR
+000032* ACCOUNTING MONTH OF SALE ORDHDR
49 +000033 06 ACTYR PIC S9(2) COMP-3. ORDHDR
+000034* ACCOUNTING YEAR OF SALE ORDHDR
50 +000035 06 STATE PIC X(2). ORDHDR
+000036* STATE ORDHDR
51 +000037 06 AMPAID PIC S9(6)V9(2) COMP-3. ORDHDR
+000038* AMOUNT PAID ORDHDR
003800
52 003900 FD ORDER-DETAIL-FILE.
53 004000 01 ORDER-DETAIL-RECORD.
004100 COPY DDS-ORDDTL OF ORDDTL.
+000001* I-O FORMAT:ORDDTL FROM FILE ORDDTLP OF LIBRARY CBLGUIDE ORDDTL
+000002* ORDER DETAIL RECORD ORDDTL
+000003* USER SUPPLIED KEY BY RECORD KEY CLAUSE ORDDTL
54 +000004 05 ORDDTL. ORDDTL
55 +000005 06 CUST PIC X(5). ORDDTL
+000006* CUSTOMER NUMBER ORDDTL
56 +000007 06 ORDERN PIC S9(5) COMP-3. ORDDTL
+000008* ORDER NUMBER ORDDTL
57 +000009 06 LINNUM PIC S9(3) COMP-3. ORDDTL
+000010* LINE NUMBER OF LINE IN ORDER ORDDTL
58 +000011 06 ITEM PIC S9(5) COMP-3. ORDDTL
+000012* ITEM NUMBER ORDDTL
59 +000013 06 QTYORD PIC S9(3) COMP-3. ORDDTL
+000014* QUANTITY ORDERED ORDDTL
60 +000015 06 DESCRP PIC X(30). ORDDTL
+000016* ITEM DESCRIPTION ORDDTL
61 +000017 06 PRICE PIC S9(4)V9(2) COMP-3. ORDDTL
+000018* SELLING PRICE ORDDTL
62 +000019 06 EXTENS PIC S9(6)V9(2) COMP-3. ORDDTL
+000020* EXTENSION AMOUNT OF QTYORD X PRICE ORDDTL
63 +000021 06 WHSLOC PIC X(3). ORDDTL
+000022* BIN NO. ORDDTL
64 +000023 06 ORDDAT PIC S9(6) COMP-3. ORDDTL
+000024* DATE ORDER WAS ENTERED ORDDTL
65 +000025 06 CUSTYP PIC S9(1) COMP-3. ORDDTL
+000026* CUSTOMER TYPE 1=GOV 2=SCH 3=BUS 4=PVT 5=OT ORDDTL
66 +000027 06 STATE PIC X(2). ORDDTL
+000028* STATE ORDDTL
67 +000029 06 ACTMTH PIC S9(2) COMP-3. ORDDTL

```

Figure 149. Example of an Order Inquiry Program (Part 2 of 10)

STMT	PL	SEQNBR	-A	1	B.	+	...	2	...	3	...	4	...	5	...	6	...	7	...	IDENTFCN	S	COPYNAME	CHG DATE
		+000030*																					
68		+000031		06	ACTYR									PIC S9(2)								COMP-3.	ORDDTL
		+000032*																					ORDDTL
69		004200	66	ORDER-DETAIL-RECORD-KEY	RENAMES	ORDERN	THRU	LINNUM.															
		004300																					
70		004400	FD	CUSTOMER-MASTER-FILE.																			
71		004500	01	CUSTOMER-MASTER-RECORD.																			
		004600		COPY DDS-CUSMST OF CUSMSTP.																			
		+000001*		I-O FORMAT:CUSMST		FROM FILE	CUSMSTP																CUSMST
		+000002*																					CUSMST
		+000003*		USER SUPPLIED KEY BY RECORD KEY CLAUSE																			CUSMST
72		+000004		05	CUSMST.																		CUSMST
73		+000005		06	CUST																		CUSMST
		+000006*																					CUSMST
74		+000007		06	NAME																		CUSMST
		+000008*																					CUSMST
75		+000009		06	ADDR																		CUSMST
		+000010*																					CUSMST
76		+000011		06	CITY																		CUSMST
		+000012*																					CUSMST
77		+000013		06	STATE																		CUSMST
		+000014*																					CUSMST
78		+000015		06	ZIP																		CUSMST
		+000016*																					CUSMST
79		+000017		06	SRHCO																		CUSMST
		+000018*																					CUSMST
80		+000019		06	CUSTYP																		CUSMST
		+000020*																					CUSMST
81		+000021		06	ARBAL																		CUSMST
		+000022*																					CUSMST
82		+000023		06	ORDBAL																		CUSMST
		+000024*																					CUSMST
83		+000025		06	LSTAMT																		CUSMST
		+000026*																					CUSMST
84		+000027		06	LSTDAT																		CUSMST
		+000028*																					CUSMST
85		+000029		06	CRDLMT																		CUSMST
		+000030*																					CUSMST
86		+000031		06	SLSYR																		CUSMST
		+000032*																					CUSMST
87		+000033		06	SLSLYR																		CUSMST
		+000034*																					CUSMST
		004700																					
88		004800	FD	EXISTING-ORDER-DISPLAY-FILE.																			
89		004900	01	EXISTING-ORDER-DISPLAY-RECORD.																			
		005000		COPY DDS-ALL-FORMATS OF ORDINQD.																			
90		+000001		05	ORDINQD-RECORD																		<-ALL-FMTS
		+000002*		I-O FORMAT:SUB1		FROM FILE	ORDINQD																<-ALL-FMTS
		+000003*																					<-ALL-FMTS
91		+000004		05	SUB1																		<-ALL-FMTS
92		+000005		06	ITEM																		<-ALL-FMTS
		+000006*																					<-ALL-FMTS
93		+000007		06	QTYORD																		<-ALL-FMTS
		+000008*																					<-ALL-FMTS
94		+000009		06	DESCRP																		<-ALL-FMTS

Figure 149. Example of an Order Inquiry Program (Part 3 of 10)

```

STMT PL SEQNBR -A 1 B..+...2...+...3...+...4...+...5...+...6...+...7..IDENTFCN S COPYNAME CHG DATE
+000010*          ITEM DESCRIPTION                                <-ALL-FMTS
95  +000011          06 PRICE PIC S9(4)V9(2).                    <-ALL-FMTS
+000012*          SELLING PRICE                                <-ALL-FMTS
96  +000013          06 EXTENS PIC S9(6)V9(2).                  <-ALL-FMTS
+000014*          EXTENSION AMOUNT OF QTYORD X PRICE          <-ALL-FMTS
+000015* INPUT FORMAT:SUBCTL1 FROM FILE ORDINQD OF LIBRARY CBLGUIDE <-ALL-FMTS
+000016*          <-ALL-FMTS
97  +000017          05 SUBCTL1-I REDEFINES ORDINQD-RECORD.    <-ALL-FMTS
98  +000018          06 SUBCTL1-I-INDIC.                       <-ALL-FMTS
99  +000019          07 IN97 PIC 1 INDIC 97.                   <-ALL-FMTS
+000020*          CONTINUE DISPLAY                             <-ALL-FMTS
100 +000021          07 IN98 PIC 1 INDIC 98.                    <-ALL-FMTS
+000022*          END OF PROGRAM                               <-ALL-FMTS
101 +000023          07 IN57 PIC 1 INDIC 57.                   <-ALL-FMTS
+000024*          DISPLAY SUBFILE                              <-ALL-FMTS
102 +000025          07 IN58 PIC 1 INDIC 58.                   <-ALL-FMTS
+000026*          OFF = DISPLAY SUBCTL1 ON = CLEAR SUBFILE    <-ALL-FMTS
103 +000027          07 IN61 PIC 1 INDIC 61.                   <-ALL-FMTS
+000028*          ORDER NUMBER NOT FOUND                     <-ALL-FMTS
104 +000029          07 IN47 PIC 1 INDIC 47.                   <-ALL-FMTS
+000030*          NO LINE FOR THIS ORDER                      <-ALL-FMTS
105 +000031          07 IN62 PIC 1 INDIC 62.                   <-ALL-FMTS
+000032*          NO CUSTOMER RECORD                           <-ALL-FMTS
106 +000033          06 ORDERN PIC S9(5).                       <-ALL-FMTS
+000034*          ORDER NUMBER                                 <-ALL-FMTS
+000035* OUTPUT FORMAT:SUBCTL1 FROM FILE ORDINQD OF LIBRARY CBLGUIDE <-ALL-FMTS
+000036*          <-ALL-FMTS
107 +000037          05 SUBCTL1-0 REDEFINES ORDINQD-RECORD.    <-ALL-FMTS
108 +000038          06 SUBCTL1-0-INDIC.                       <-ALL-FMTS
109 +000039          07 IN58 PIC 1 INDIC 58.                   <-ALL-FMTS
+000040*          OFF = DISPLAY SUBCTL1 ON = CLEAR SUBFILE    <-ALL-FMTS
110 +000041          07 IN57 PIC 1 INDIC 57.                   <-ALL-FMTS
+000042*          DISPLAY SUBFILE                              <-ALL-FMTS
111 +000043          07 IN45 PIC 1 INDIC 45.                   <-ALL-FMTS
112 +000044          07 IN47 PIC 1 INDIC 47.                   <-ALL-FMTS
+000045*          NO LINE FOR THIS ORDER                      <-ALL-FMTS
113 +000046          07 IN61 PIC 1 INDIC 61.                   <-ALL-FMTS
+000047*          ORDER NUMBER NOT FOUND                     <-ALL-FMTS
114 +000048          07 IN62 PIC 1 INDIC 62.                   <-ALL-FMTS
+000049*          NO CUSTOMER RECORD                           <-ALL-FMTS
115 +000050          06 ORDERN PIC S9(5).                       <-ALL-FMTS
+000051*          ORDER NUMBER                                 <-ALL-FMTS
116 +000052          06 ORDDAT PIC S9(6).                       <-ALL-FMTS
+000053*          DATE ORDER WAS ENTERED                     <-ALL-FMTS
117 +000054          06 CUST PIC X(5).                          <-ALL-FMTS
+000055*          CUSTOMER NUMBER                             <-ALL-FMTS
118 +000056          06 NAME PIC X(25).                         <-ALL-FMTS
+000057*          CUSTOMER NAME                               <-ALL-FMTS
119 +000058          06 ADDR PIC X(20).                         <-ALL-FMTS
+000059*          CUSTOMER ADDRESS                           <-ALL-FMTS
120 +000060          06 CITY PIC X(20).                         <-ALL-FMTS
+000061*          CUSTOMER CITY                               <-ALL-FMTS
121 +000062          06 STATE PIC X(2).                         <-ALL-FMTS
+000063*          CUSTOMER STATE                             <-ALL-FMTS
122 +000064          06 ZIP PIC S9(5).                          <-ALL-FMTS

```

Figure 149. Example of an Order Inquiry Program (Part 4 of 10)

```

5722WDS V5R4M0 060210 LN IBM ILE COBOL CBLGUIDE/ORDINQ ISERIES1 06/02/15 15:06:50 Page 6
STMT PL SEQNBR -A 1 B.+...2...+...3...+...4...+...5...+...6...+...7..IDENTFCN S COPYNAME CHG DATE
+000065* ZIP CODE <-ALL-FMTS
123 +000066 06 ORDAMT PIC S9(6)V9(2). <-ALL-FMTS
+000067* TOTAL AMOUNT OF ORDER <-ALL-FMTS
124 +000068 06 STSORD PIC X(12). <-ALL-FMTS
125 +000069 06 STSOPN PIC X(12). <-ALL-FMTS
126 +000070 06 CUSORD PIC X(15). <-ALL-FMTS
+000071* CUSTOMER PURCHASE ORDER NUMBER <-ALL-FMTS
127 +000072 06 SHPVIA PIC X(15). <-ALL-FMTS
+000073* SHIPPING INSTRUCTIONS <-ALL-FMTS
128 +000074 06 PRDAT PIC S9(6). <-ALL-FMTS
+000075* DATE ORDER WAS PRINTED <-ALL-FMTS
129 +000076 06 INVNUM PIC S9(5). <-ALL-FMTS
+000077* INVOICE NUMBER <-ALL-FMTS
130 +000078 06 ACTMTH PIC S9(2). <-ALL-FMTS
+000079* ACCOUNTING MONTH OF SALE <-ALL-FMTS
131 +000080 06 ACTYR PIC S9(2). <-ALL-FMTS
+000081* ACCOUNTING YEAR OF SALE <-ALL-FMTS
005100
132 005200 WORKING-STORAGE SECTION.
133 005300 01 EXISTING-ORDER-DISPLAY-KEY.
134 005400 05 SUBFILE-RECORD-NUMBER PIC 9(2)
005500 VALUE ZERO.
005600
135 005700 01 ORDER-STATUS-COMMENT-VALUES.
136 005800 05 FILLER PIC X(12)
005900 VALUE "1-IN PROCESS".
137 006000 05 FILLER PIC X(12)
006100 VALUE "2-CONTINUED ".
138 006200 05 FILLER PIC X(12)
006300 VALUE "3-CREDIT CHK".
139 006400 05 FILLER PIC X(12)
006500 VALUE "4-READY PRT ".
140 006600 05 FILLER PIC X(12)
006700 VALUE "5-PRINTED ".
141 006800 05 FILLER PIC X(12)
006900 VALUE "6-PICKED ".
142 007000 05 FILLER PIC X(12)
007100 VALUE "7-INVOICED ".
143 007200 05 FILLER PIC X(12)
007300 VALUE "8-INVALID ".
144 007400 05 FILLER PIC X(12)
007500 VALUE "9-CANCELED ".
007600
145 007700 01 ORDER-STATUS-COMMENT-TABLE
007800 REDEFINES ORDER-STATUS-COMMENT-VALUES.
146 007900 05 ORDER-STATUS OCCURS 9 TIMES.
147 008000 10 ORDER-STATUS-COMMENT PIC X(12).
008100
148 008200 01 OPEN-STATUS-COMMENT-VALUES.
149 008300 05 FILLER PIC X(12)
008400 VALUE "1-OPEN ".
150 008500 05 FILLER PIC X(12)
008600 VALUE "2-CLOSED ".
151 008700 05 FILLER PIC X(12)
008800 VALUE "3-CANCELED ".

```

Figure 149. Example of an Order Inquiry Program (Part 5 of 10)

```

STMT PL SEQNBR -A 1 B.+...2....+...3....+...4....+...5....+...6....+...7..IDENTFCN S COPYNAME CHG DATE
008900
152 009000 01 OPEN-STATUS-COMMENT-TABLE
009100 REDEFINES OPEN-STATUS-COMMENT-VALUES.
153 009200 05 OPEN-STATUS OCCURS 3 TIMES.
154 009300 10 OPEN-STATUS-COMMENT PIC X(12).
009400
155 009500 01 ERRHDL-PARAMETERS.
156 009600 05 STATUS-CODE-ONE PIC X(2).
157 009700 88 SUBFILE-IS-FULL VALUE "0M".
009800
158 009900 01 ERRPGM-PARAMETERS.
159 010000 05 DISPLAY-PARAMETER PIC X(8)
010100 VALUE "ORD220D ".
160 010200 05 DUMMY-ONE PIC X(6)
010300 VALUE SPACES.
161 010400 05 DUMMY-TWO PIC X(8)
010500 VALUE SPACES.
162 010600 05 STATUS-CODE-TWO.
163 010700 10 PRIMARY PIC X(1).
164 010800 10 SECONDARY PIC X(1).
165 010900 10 FILLER PIC X(5)
011000 VALUE SPACES.
011100
166 011200 01 SWITCH-AREA.
167 011300 05 SW01 PIC 1.
168 011400 88 NO-MORE-DETAIL-LINE-ITEMS VALUE B"1".
169 011500 88 MORE-DETAIL-LINE-ITEMS-EXIST VALUE B"0".
170 011600 05 SW02 PIC 1.
171 011700 88 WRITE-DISPLAY VALUE B"1".
172 011800 88 READ-DISPLAY VALUE B"0".
173 011900 05 SW03 PIC 1.
174 012000 88 SUBCTL1-FORMAT VALUE B"1".
175 012100 88 NOT-SUBCTL1-FORMAT VALUE B"0".
176 012200 05 SW04 PIC 1.
177 012300 88 SUB1-FORMAT VALUE B"1".
178 012400 88 NOT-SUB1-FORMAT VALUE B"0".
012500
179 012600 01 INDICATOR-AREA.
180 012700 05 IN98 PIC 1 INDIC 98.
181 012800 88 END-OF-EXISTING-ORDER-INQUIRY VALUE B"1".
182 012900 05 IN97 PIC 1 INDIC 97.
183 013000 88 CONTINUE-DETAIL-LINES-DISPLAY VALUE B"1".
184 013100 05 IN62 PIC 1 INDIC 62.
185 013200 88 CUSTOMER-NOT-FOUND VALUE B"1".
186 013300 88 CUSTOMER-EXIST VALUE B"0".
187 013400 05 IN61 PIC 1 INDIC 61.
188 013500 88 ORDER-NOT-FOUND VALUE B"1".
189 013600 88 ORDER-EXIST VALUE B"0".
190 013700 05 IN58 PIC 1 INDIC 58.
191 013800 88 CLEAR-SUBFILE VALUE B"1".
192 013900 88 DISPLAY-SUBFILE-CONTROL VALUE B"0".
193 014000 05 IN57 PIC 1 INDIC 57.
194 014100 88 DISPLAY-SUBFILE VALUE B"1".
195 014200 05 IN47 PIC 1 INDIC 47.
196 014300 88 NO-DETAIL-LINES-FOR-ORDER VALUE B"1".

```

Figure 149. Example of an Order Inquiry Program (Part 6 of 10)

```

5722WDS V5R4M0 060210 LN IBM ILE COBOL CBLGUIDE/ORDINQ ISERIES1 06/02/15 15:06:50 Page 8
STMT PL SEQNBR -A 1 B.+...2....+...3....+...4....+...5....+...6....+...7..IDENTFCN S COPYNAME CHG DATE
197 014400 88 DETAIL-LINES-FOR-ORDER-EXIST VALUE B"0".
198 014500 05 IN45 PIC 1 INDIC 45.
199 014600 88 END-OF-ORDER VALUE B"1".
014700
200 014800 PROCEDURE DIVISION.
014900
201 015000 DECLARATIVES.
015100 TRANSACTION-ERROR SECTION.
015200 USE AFTER STANDARD ERROR PROCEDURE
015300 EXISTING-ORDER-DISPLAY-FILE.
015400 WORK-STATION-ERROR-HANDLER.
202 015500 IF NOT (SUBFILE-IS-FULL) THEN
203 015600 DISPLAY "WORK-STATION ERROR" STATUS-CODE-ONE
015700 END-IF.
015800 END DECLARATIVES.
015900
016000 MAIN-PROGRAM SECTION.
016100 MAINLINE.
204 016200 OPEN INPUT ORDER-HEADER-FILE
016300 ORDER-DETAIL-FILE
016400 CUSTOMER-MASTER-FILE
016500 I-O EXISTING-ORDER-DISPLAY-FILE.
205 016600 MOVE SPACES TO CUST OF SUBCTL1-0
016700 NAME OF SUBCTL1-0
016800 ADDR OF SUBCTL1-0
016900 CITY OF SUBCTL1-0
017000 STATE OF SUBCTL1-0
017100 STSORD OF SUBCTL1-0
017200 STSOPN OF SUBCTL1-0
017300 CUSORD OF SUBCTL1-0.
206 017400 MOVE ZEROS TO ORDDAT OF SUBCTL1-0
017500 ORDDAT OF SUBCTL1-0
017600 ZIP OF SUBCTL1-0
017700 ORDAMT OF SUBCTL1-0
017800 PRDAT OF SUBCTL1-0
017900 INVNUM OF SUBCTL1-0
018000 ACTMTH OF SUBCTL1-0
018100 ACTYR OF SUBCTL1-0.
207 018200 MOVE B"0" TO INDICATOR-AREA.
208 018300 SET READ-DISPLAY
018400 NOT-SUBCTL1-FORMAT
018500 NOT-SUB1-FORMAT TO TRUE.
209 018600 MOVE CORR INDICATOR-AREA TO SUBCTL1-0-INDIC.
*** CORRESPONDING items for statement 209:
*** IN62
*** IN61
*** IN58
*** IN57
*** IN47
*** IN45
*** End of CORRESPONDING items for statement 209
210 018700 WRITE EXISTING-ORDER-DISPLAY-RECORD FORMAT IS "SUBCTL1"
018800 END-WRITE
211 018900 READ EXISTING-ORDER-DISPLAY-FILE RECORD.
212 019000 MOVE CORR SUBCTL1-I-INDIC TO INDICATOR-AREA.

```

Figure 149. Example of an Order Inquiry Program (Part 7 of 10)

```

STMT PL SEQNBR -A 1 B.+...2...+...3...+...4...+...5...+...6...+...7..IDENTFCN S COPYNAME CHG DATE
*** CORRESPONDING items for statement 212:
*** IN97
*** IN98
*** IN57
*** IN58
*** IN61
*** IN47
*** IN62
*** End of CORRESPONDING items for statement 212
019100
213 019200 PERFORM EXISTING-ORDER-INQUIRY
019300 UNTIL END-OF-EXISTING-ORDER-INQUIRY.
019400
214 019500 CLOSE ORDER-HEADER-FILE
019600 ORDER-DETAIL-FILE
019700 CUSTOMER-MASTER-FILE
019800 EXISTING-ORDER-DISPLAY-FILE.
215 019900 STOP RUN.
020000
020100 EXISTING-ORDER-INQUIRY.
216 020200 IF CONTINUE-DETAIL-LINES-DISPLAY THEN
217 020300 PERFORM READ-NEXT-ORDER-DETAIL-RECORD
218 020400 IF MORE-DETAIL-LINE-ITEMS-EXIST THEN
219 020500 IF ORDERN OF ORDER-DETAIL-RECORD IS NOT EQUAL TO
020600 ORDERN OF ORDER-HEADER-RECORD THEN
220 020700 SET DISPLAY-SUBFILE TO TRUE
221 020800 SET NO-DETAIL-LINES-FOR-ORDER TO TRUE
020900 ELSE
222 021000 PERFORM SUBFILE-SET-UP
021100 END-IF
021200 ELSE
223 021300 SET DISPLAY-SUBFILE TO TRUE
224 021400 SET NO-DETAIL-LINES-FOR-ORDER TO TRUE
021500 END-IF
021600 ELSE
225 021700 PERFORM ORDER-NUMBER-VALIDATION
021800 END-IF
226 021900 MOVE CORR INDICATOR-AREA TO SUBCTL1-0-INDIC.
*** CORRESPONDING items for statement 226:
*** IN62
*** IN61
*** IN58
*** IN57
*** IN47
*** IN45
*** End of CORRESPONDING items for statement 226
227 022000 SET WRITE-DISPLAY TO TRUE.
228 022100 SET SUBCTL1-FORMAT TO TRUE.
229 022200 WRITE EXISTING-ORDER-DISPLAY-RECORD FORMAT IS "SUBCTL1".
230 022300 READ EXISTING-ORDER-DISPLAY-FILE RECORD.
231 022400 MOVE CORR SUBCTL1-I-INDIC TO INDICATOR-AREA.
*** CORRESPONDING items for statement 231:
*** IN97
*** IN98
*** IN57

```

Figure 149. Example of an Order Inquiry Program (Part 8 of 10)

```

STMT PL SEQNBR -A 1 B.+...2....+...3....+...4....+...5....+...6....+...7..IDENTFCN S COPYNAME CHG DATE
***      IN58
***      IN61
***      IN47
***      IN62
*** End of CORRESPONDING items for statement 231
022500
022600 ORDER-NUMBER-VALIDATION.
232 022700 PERFORM READ-ORDER-HEADER-FILE.
233 022800 IF ORDER-EXIST THEN
234 022900     PERFORM READ-CUSTOMER-MASTER-FILE
235 023000     IF CUSTOMER-EXIST THEN
236 023100         PERFORM READ-FIRST-ORDER-DETAIL-RECORD
237 023200         IF DETAIL-LINES-FOR-ORDER-EXIST THEN
238 023300             PERFORM SUBFILE-SET-UP
023400             END-IF
023500         END-IF
023600     END-IF.
023700
023800 READ-ORDER-HEADER-FILE.
239 023900 MOVE ORDERN OF SUBCTL1-I OF EXISTING-ORDER-DISPLAY-RECORD
024000     TO ORDERN OF ORDER-HEADER-RECORD.
240 024100 READ ORDER-HEADER-FILE
241 024200     INVALID KEY SET ORDER-NOT-FOUND TO TRUE
024300     END-READ.
024400
024500 READ-CUSTOMER-MASTER-FILE.
242 024600 MOVE CUST OF ORDER-HEADER-RECORD
024700     TO CUST OF CUSTOMER-MASTER-RECORD.
243 024800 READ CUSTOMER-MASTER-FILE
244 024900     INVALID KEY SET CUSTOMER-NOT-FOUND TO TRUE
025000     END-READ.
025100
025200 READ-FIRST-ORDER-DETAIL-RECORD.
245 025300 MOVE ORDERN OF ORDER-HEADER-RECORD
025400     TO ORDERN OF ORDER-DETAIL-RECORD.
246 025500 MOVE 1 TO LINNUM OF ORDER-DETAIL-RECORD.
247 025600 READ ORDER-DETAIL-FILE
248 025700     INVALID KEY SET NO-DETAIL-LINES-FOR-ORDER TO TRUE
025800     END-READ.
025900
026000 SUBFILE-SET-UP.
249 026100 SET CLEAR-SUBFILE TO TRUE.
250 026200 MOVE CORR INDICATOR-AREA TO SUBCTL1-0-INDIC.
*** CORRESPONDING items for statement 250:
***      IN62
***      IN61
***      IN58
***      IN57
***      IN47
***      IN45
*** End of CORRESPONDING items for statement 250
251 026300 SET WRITE-DISPLAY TO TRUE.
252 026400 SET SUBCTL1-FORMAT TO TRUE.
253 026500 WRITE EXISTING-ORDER-DISPLAY-RECORD FORMAT IS "SUBCTL1"
026600 END-WRITE

```

Figure 149. Example of an Order Inquiry Program (Part 9 of 10)

```

5722WDS V5R4M0 060210 LN IBM ILE COBOL          CBLGUIDE/ORDINQ          ISERIES1 06/02/15 15:06:50      Page 11
STMT PL SEQNBR -A 1 B..+...2...+...3...+...4...+...5...+...6...+...7..IDENTFCN S COPYNAME  CHG DATE
254 026700 SET DISPLAY-SUBFILE-CONTROL TO TRUE.
255 026800 PERFORM BUILD-DISPLAY-SUBFILE
026900 UNTIL NO-MORE-DETAIL-LINE-ITEMS OR SUBFILE-IS-FULL.
256 027000 MOVE CORR ORDHDR OF ORDER-HEADER-RECORD
027100 TO SUBCTL1-0 OF EXISTING-ORDER-DISPLAY-RECORD.
*** CORRESPONDING items for statement 256:
*** CUST
*** ORDERN
*** ORDDAT
*** CUSORD
*** SHPVIA
*** ORDAMT
*** INVNUM
*** PRTDAT
*** ACTMTH
*** ACTYR
*** STATE
*** End of CORRESPONDING items for statement 256
257 027200 MOVE CORR CUSMST OF CUSTOMER-MASTER-RECORD
027300 TO SUBCTL1-0 OF EXISTING-ORDER-DISPLAY-RECORD.
*** CORRESPONDING items for statement 257:
*** CUST
*** NAME
*** ADDR
*** CITY
*** STATE
*** ZIP
*** End of CORRESPONDING items for statement 257
258 027400 MOVE ORDER-STATUS(ORDSTS) TO STSORD.
259 027500 MOVE OPEN-STATUS(OPNSTS) TO STSOPN.
260 027600 SET MORE-DETAIL-LINE-ITEMS-EXIST TO TRUE.
261 027700 MOVE ZEROS TO SUBFILE-RECORD-NUMBER.
027800
027900 BUILD-DISPLAY-SUBFILE.
262 028000 MOVE CORR ORDDTL OF ORDER-DETAIL-RECORD
028100 TO SUB1 OF EXISTING-ORDER-DISPLAY-RECORD.
*** CORRESPONDING items for statement 262:
*** ITEM
*** QTYORD
*** DESCRP
*** PRICE
*** EXTENS
*** End of CORRESPONDING items for statement 262
263 028200 SET WRITE-DISPLAY TO TRUE.
264 028300 SET SUB1-FORMAT TO TRUE.
265 028400 ADD 1 TO SUBFILE-RECORD-NUMBER.
266 028500 WRITE SUBFILE EXISTING-ORDER-DISPLAY-RECORD FORMAT IS "SUB1"
028600 END-WRITE
267 028700 IF SUBFILE-IS-FULL THEN
268 028800 SET DISPLAY-SUBFILE TO TRUE
028900 ELSE
269 029000 PERFORM READ-NEXT-ORDER-DETAIL-RECORD
270 029100 IF MORE-DETAIL-LINE-ITEMS-EXIST THEN
271 029200 IF ORDERN OF ORDER-DETAIL-RECORD IS NOT EQUAL TO
029300 ORDERN OF ORDER-HEADER-RECORD THEN
272 029400 SET DISPLAY-SUBFILE TO TRUE
273 029500 SET NO-MORE-DETAIL-LINE-ITEMS TO TRUE
029600 END-IF
029700 END-IF
029800 END-IF.
029900
030000 READ-NEXT-ORDER-DETAIL-RECORD.
274 030100 READ ORDER-DETAIL-FILE NEXT RECORD
275 030200 AT END SET DISPLAY-SUBFILE TO TRUE
276 030300 SET NO-MORE-DETAIL-LINE-ITEMS TO TRUE
030400 END-READ.
***** END OF SOURCE *****

```

Figure 149. Example of an Order Inquiry Program (Part 10 of 10)

This is the initial order-entry prompt display written to the workstation:

```

Existing Order Entry                Total 000000000
Status
Order 12400                        Open
Date 000000                        Customer order
Cust #                               Ship via
                                      Printed date 000000
                                      Invoice 000000      Mth 00 Year 00
Item Qty  Item Description          Price  Extension

```

This display appears if there are detail order records for the customer whose order number was entered in the first display:

```

Existing Order Entry                Total 007426656
Status 7-INVOICED
Order 17924  ABC HARDWARE LTD.      Open 2-CLOSED
Date 110896  123 ANYWHERE AVE.     Customer order TESTCS17933001I
Cust # 11200  TORONTO               Ship via TRUCKCO
                                      Printed date 110896
                                      M4K 0A0
                                      Invoice 17924      Mth 12 Year 88
Item Qty  Item Description          Price  Extension
33001 003 TORQUE WRENCH 75LB 14 INCH 009115 273.45
33100 001 TORQUE WRENCH W/GAUGE 200 LB 015777 650.95
44529 004 WOOD CHISEL - 3 1/4      006840 56.87
44958 002 POWER DRILL 1/2 REV      008200 797.50
46102 001 WROUGHT IRON RAILING 4FTX6FT 007930 237.75
46201 001 WROUGHT IRON HAND RAIL 6FT 007178 77.35
47902 002 ESCUTCHEON BRASS 15X4 INCHES 044488 213.00

```

This display appears if the ORDHDRP file does not contain a record for the order number entered on the first display:

```

Existing Order Entry                Total 000000000
Status
Order 12400                        Open
Date 000000                        Customer order
Cust #                               Ship via
                                      Printed date 000000
                                      Invoice 000000      Mth 00 Year 00
Item Qty  Item Description          Price  Extension

```

Order number not found

Example of Using READ SUBFILE...NEXT MODIFIED and REWRITE SUBFILE in a Payment Update Program

Figure 152 on page 587 shows an example of a payment update program, PAYUPDT. For the related DDS, see Figure 150 on page 584 and Figure 151 on page 585. For the related display-screen examples, see page "Customer Payment Display" on page 599. For the DDS for the customer master file, CUSMSTP, refer to Figure 132 on page 531.

In this example, payments from customers are registered. The clerk is prompted to enter one or more customer numbers and the amount of money to be credited to each customer's account. The program checks the customer number and unconditionally accepts any payment for an existing customer who has invoices outstanding. If an overpayment will result from the amount of the payment from a customer, the clerk is given the option of accepting or rejecting the payment. If no customer record exists for a customer number, an error message is issued. Payments can be entered until the clerk ends the program by pressing F12.

```

.....+.....1.....+.....2.....+.....3.....+.....4.....+.....5.....+.....6.....+.....7.....+.....
A** THIS IS THE ORDER HEADER LOGICAL FILE ** ORDHDR
A
A
A          R ORDHDR                UNIQUE
A*          PFILE(ORDHDRP)
A          CUST
A          INVNUM
A          ORDERN
A          ORDDAT
A          CUSORD
A          SHPVIA
A          ORDSTS
A          OPRNAM
A          ORDAMT
A          CUSTYP
A          PRTDAT
A          OPNSTS
A          TOTLIN
A          ACTMTH
A          ACTYR
A          STATE
A          AMPAID
A          K CUST
A          K INVNUM

```

Figure 150. Example of a Data Description Specification for a Payment Update Program - Logical Order File

```

.....1.....2.....3.....4.....5.....6.....7.....
A* THIS IS THE DISPLAY DEVICE FILE FOR PAYUPDT ** PAYUPDTD
A* ACCOUNTS RECEIVABLE INTERACTIVE PAYMENT UPDATE
A*
A
A          R SUBFILE1                      SFL
A          TEXT('SUBFILE FOR CUSTOMER PAYMENT')
A*
A          ACPMPT          4A I 5 4TEXT('ACCEPT PAYMENT')
A          VALUES('*YES' '*NO')
A 51          DSPATR(RI MDT)
A N51         DSPATR(ND PR)
A*
A          CUST           5 B 5 15TEXT('CUSTOMER NUMBER')
A 52          DSPATR(RI)
A 53          DSPATR(ND)
A 54          DSPATR(PR)
A*
A          AMPAID        8 02B 5 24TEXT('AMOUNT PAID')
A          CHECK(FE)
A          AUTO(RAB)
A          CMP(GT 0)
A 52          DSPATR(RI)
A 53          DSPATR(ND)
A 54          DSPATR(PR)
A*
A          ECPMSG        31A 0 5 37TEXT('EXCEPTION MESSAGE')
A 52          DSPATR(RI)
A 53          DSPATR(ND)
A 54          DSPATR(BL)
A*
A          OVRPMT        8Y 20 5 70TEXT('OVERPAYMENT')
A          EDTCDE(1)
A 55          DSPATR(BL)
A N56         DSPATR(ND)
A*

```

Figure 151. Example of a Data Description Specification for a Payment Update Program - Display Device File (Part 1 of 2)

A	STSCDE	1A H	TEXT('STATUS CODE')
A	R CONTROL1		TEXT('SUBFILE CONTROL')
A			SFLCTL(SUBFILE1)
A			SFLSIZ(17)
A			SFLPAG(17)
A	61		SFLCLR
A	62		SFLDSP
A	62		SFLDSPCTL
A			OVERLAY
A			LOCK
A*			
A			HELP(99 'HELP KEY')
A			CA12(98 'END PAYMENT UPDATE')
A			CA11(97 'IGNORE INPUT')
A*			
A	99		SFLMSG(' F11 - IGNORE INVALID INPUT+
A			F12 - END PAYMENT +
A			UPDATE')
A*			
A			1 2'CUSTOMER PAYMENT UPDATE PROMPT'
A			1 65'DATE'
A			1 71DATE EDTCDE(Y)
A	63		3 2'ACCEPT'
A	63		4 2'PAYMENT'
A			3 14'CUSTOMER'
A			3 26'PAYMENT'
A	64		3 37'EXCEPTION MESSAGE'
A*			
A	R MESSAGE1		TEXT('MESSAGE RECORD')
A			OVERLAY
A			LOCK
A*			
A	71		24 2' ACCEPT PAYMENT VALUES: (*NO *YES)
			DSPATR(RI)

Figure 151. Example of a Data Description Specification for a Payment Update Program - Display Device File (Part 2 of 2)

```

Source
STMT PL SEQNBR -A 1 B.+....2....+....3....+....4....+....5....+....6....+....7..IDENTFCN S COPYNAME CHG DATE
1 000100 PROCESS APOST
2 000200 IDENTIFICATION DIVISION.
2 000300 PROGRAM-ID. PAYUPDT.
000400
3 000500 ENVIRONMENT DIVISION.
4 000600 CONFIGURATION SECTION.
5 000700 SOURCE-COMPUTER. IBM-ISERIES
6 000800 OBJECT-COMPUTER. IBM-ISERIES
7 000900 INPUT-OUTPUT SECTION.
8 001000 FILE-CONTROL.
9 001100 SELECT CUSTOMER-INVOICE-FILE
10 001200 ASSIGN TO DATABASE-ORDHDRL
11 001300 ORGANIZATION IS INDEXED
12 001400 ACCESS MODE IS SEQUENTIAL
13 001500 RECORD KEY IS COMP-KEY
14 001600 FILE STATUS IS STATUS-CODE-ONE.
15 001700 SELECT CUSTOMER-MASTER-FILE
16 001800 ASSIGN TO DATABASE-CUSMSTP
17 001900 ORGANIZATION IS INDEXED
18 002000 ACCESS IS RANDOM
19 002100 RECORD KEY IS CUST OF CUSTOMER-MASTER-RECORD.
20 002200 SELECT PAYMENT-UPDATE-DISPLAY-FILE
21 002300 ASSIGN TO WORKSTATION-PAYUPDTD
22 002400 ORGANIZATION IS TRANSACTION
23 002500 ACCESS IS DYNAMIC
24 002600 RELATIVE KEY IS REL-NUMBER
25 002700 FILE STATUS IS STATUS-CODE-ONE
26 002800 CONTROL-AREA IS WS-CONTROL.
002900
27 003000 DATA DIVISION.
28 003100 FILE SECTION.
29 003200 FD CUSTOMER-INVOICE-FILE.
30 003300 01 CUSTOMER-INVOICE-RECORD.
003400 COPY DDS-ORDHDR OF ORDHDRL.
+000001* I-O FORMAT:ORDHDR FROM FILE ORDHDRL OF LIBRARY CBLGUIDE ORDHDR
+000002* ORDHDR
+000003* USER SUPPLIED KEY BY RECORD KEY CLAUSE ORDHDR
31 +000004 05 ORDHDR. ORDHDR
32 +000005 06 CUST PIC X(5). ORDHDR
+000006* CUSTOMER NUMBER ORDHDR
33 +000007 06 INVNUM PIC S9(5) COMP-3. ORDHDR
+000008* INVOICE NUMBER ORDHDR
34 +000009 06 ORDERN PIC S9(5) COMP-3. ORDHDR
+000010* ORDER NUMBER ORDHDR
35 +000011 06 ORDDAT PIC S9(6) COMP-3. ORDHDR
+000012* DATE ORDER ENTERED ORDHDR
36 +000013 06 CUSORD PIC X(15). ORDHDR
+000014* CUSTOMER PURCHASE ORDER NUMBER ORDHDR
37 +000015 06 SHPVIA PIC X(15). ORDHDR
+000016* SHIPPING INSTRUCTIONS ORDHDR
38 +000017 06 ORDSTS PIC S9(1) COMP-3. ORDHDR
+000018* ORDER SATAUS 1PCS 2CNT 3CHK 4RDY 5PRT 6PCK ORDHDR
39 +000019 06 OPRNAM PIC X(10). ORDHDR

```

Figure 152. Source Listing of a Payment Update Program Example (Part 1 of 13)

```

STMT PL SEQNBR -A 1 B.+...2....+...3....+...4....+...5....+...6....+...7..IDENTFCN S COPYNAME CHG DATE
+000020* OPERATOR WHO ENTERED ORD ORDHDR
40 +000021 06 ORDAMT PIC S9(6)V9(2) COMP-3. ORDHDR
+000022* DOLLAR AMOUNT OF ORDER ORDHDR
41 +000023 06 CUSTYP PIC S9(1) COMP-3. ORDHDR
+000024* CUSTOMER TYPE 1=GOV 2=SCH 3=BUS 4=PVT 5=OT ORDHDR
42 +000025 06 PRDAT PIC S9(6) COMP-3. ORDHDR
+000026* DATE ORDER WAS PRINTED ORDHDR
43 +000027 06 OPNSTS PIC S9(1) COMP-3. ORDHDR
+000028* ORDER OPEN STATUS 1=OPEN 2= CLOSE 3=CANCEL ORDHDR
44 +000029 06 TOTLIN PIC S9(3) COMP-3. ORDHDR
+000030* TOTAL LINE ITEMS IN ORDER ORDHDR
45 +000031 06 ACTMTH PIC S9(2) COMP-3. ORDHDR
+000032* ACCOUNTING MONTH OF SALE ORDHDR
46 +000033 06 ACTYR PIC S9(2) COMP-3. ORDHDR
+000034* ACCOUNTING YEAR OF SALE ORDHDR
47 +000035 06 STATE PIC X(2). ORDHDR
+000036* STATE ORDHDR
48 +000037 06 AMPAID PIC S9(6)V9(2) COMP-3. ORDHDR
+000038* AMOUNT PAID ORDHDR
49 003500 66 COMP-KEY RENAMES CUST THRU INVMNUM.
003600
50 003700 FD CUSTOMER-MASTER-FILE.
51 003800 01 CUSTOMER-MASTER-RECORD.
003900 COPY DDS-CUSMST OF CUSMSTP.
+000001* I-O FORMAT:CUSMST FROM FILE CUSMSTP OF LIBRARY CBLGUIDE CUSMST
+000002* CUSTOMER MASTER RECORD CUSMST
+000003* USER SUPPLIED KEY BY RECORD KEY CLAUSE CUSMST
52 +000004 05 CUSMST. CUSMST
53 +000005 06 CUST PIC X(5). CUSMST
+000006* CUSTOMER NUMBER CUSMST
54 +000007 06 NAME PIC X(25). CUSMST
+000008* CUSTOMER NAME CUSMST
55 +000009 06 ADDR PIC X(20). CUSMST
+000010* CUSTOMER ADDRESS CUSMST
56 +000011 06 CITY PIC X(20). CUSMST
+000012* CUSTOMER CITY CUSMST
57 +000013 06 STATE PIC X(2). CUSMST
+000014* STATE CUSMST
58 +000015 06 ZIP PIC S9(5) COMP-3. CUSMST
+000016* ZIP CODE CUSMST
59 +000017 06 SRHCOD PIC X(6). CUSMST
+000018* CUSTOMER NUMBER SEARCH CODE CUSMST
60 +000019 06 CUSTYP PIC S9(1) COMP-3. CUSMST
+000020* CUSTOMER TYPE 1=GOV 2=SCH 3=BUS 4=PVT 5=OT CUSMST
61 +000021 06 ARBAL PIC S9(6)V9(2) COMP-3. CUSMST
+000022* ACCOUNTS REC. BALANCE CUSMST
62 +000023 06 ORDBAL PIC S9(6)V9(2) COMP-3. CUSMST
+000024* A/R AMT. IN ORDER FILE CUSMST
63 +000025 06 LSTAMT PIC S9(6)V9(2) COMP-3. CUSMST
+000026* LAST AMT. PAID IN A/R CUSMST
64 +000027 06 LSTDAT PIC S9(6) COMP-3. CUSMST
+000028* LAST DATE PAID IN A/R CUSMST
65 +000029 06 CRDLMT PIC S9(6)V9(2) COMP-3. CUSMST
+000030* CUSTOMER CREDIT LIMIT CUSMST
66 +000031 06 SLSYR PIC S9(8)V9(2) COMP-3. CUSMST
    
```

Figure 152. Source Listing of a Payment Update Program Example (Part 2 of 13)

```

5722WDS V5R4M0 060210 LN IBM ILE COBOL CBLGUIDE/PAYUPDT ISERIES1 06/02/15 15:08:37 Page 4
STMT PL SEQNBR -A 1 B.+...2....+...3....+...4....+...5....+...6....+...7..IDENTFCN S COPYNAME CHG DATE
+000032* CUSTOMER SALES THIS YEAR CUSMST
67 +000033 06 SLSLYR PIC S9(8)V9(2) COMP-3. CUSMST
+000034* CUSTOMER SALES LAST YEAR CUSMST
004000
68 004100 FD PAYMENT-UPDATE-DISPLAY-FILE.
69 004200 01 PAYMENT-UPDATE-DISPLAY-RECORD.
004300 COPY DDS-ALL-FORMATS OF PAYUPDTD.
70 +000001 05 PAYUPDTD-RECORD PIC X(59). <-ALL-FMTS
+000002* INPUT FORMAT:SUBFILE1 FROM FILE PAYUPDTD OF LIBRARY CBLGUIDE <-ALL-FMTS
+000003* SUBFILE FOR CUSTOMER PAYMENT <-ALL-FMTS
71 +000004 05 SUBFILE1-I REDEFINES PAYUPDTD-RECORD. <-ALL-FMTS
72 +000005 06 ACPMPT PIC X(4). <-ALL-FMTS
+000006* ACCEPT PAYMENT <-ALL-FMTS
73 +000007 06 CUST PIC X(5). <-ALL-FMTS
+000008* CUSTOMER NUMBER <-ALL-FMTS
74 +000009 06 AMPAID PIC S9(6)V9(2). <-ALL-FMTS
+000010* AMOUNT PAID <-ALL-FMTS
75 +000011 06 ECPMSG PIC X(31). <-ALL-FMTS
+000012* EXCEPTION MESSAGE <-ALL-FMTS
76 +000013 06 OVRPMT PIC S9(6)V9(2). <-ALL-FMTS
+000014* OVERPAYMENT <-ALL-FMTS
77 +000015 06 STSCDE PIC X(1). <-ALL-FMTS
+000016* STATUS CODE <-ALL-FMTS
+000017* OUTPUT FORMAT:SUBFILE1 FROM FILE PAYUPDTD OF LIBRARY CBLGUIDE <-ALL-FMTS
+000018* SUBFILE FOR CUSTOMER PAYMENT <-ALL-FMTS
78 +000019 05 SUBFILE1-O REDEFINES PAYUPDTD-RECORD. <-ALL-FMTS
79 +000020 06 SUBFILE1-O-INDIC. <-ALL-FMTS
80 +000021 07 IN51 PIC 1 INDIC 51. <-ALL-FMTS
81 +000022 07 IN52 PIC 1 INDIC 52. <-ALL-FMTS
82 +000023 07 IN53 PIC 1 INDIC 53. <-ALL-FMTS
83 +000024 07 IN54 PIC 1 INDIC 54. <-ALL-FMTS
84 +000025 07 IN55 PIC 1 INDIC 55. <-ALL-FMTS
85 +000026 07 IN56 PIC 1 INDIC 56. <-ALL-FMTS
86 +000027 06 CUST PIC X(5). <-ALL-FMTS
+000028* CUSTOMER NUMBER <-ALL-FMTS
87 +000029 06 AMPAID PIC S9(6)V9(2). <-ALL-FMTS
+000030* AMOUNT PAID <-ALL-FMTS
88 +000031 06 ECPMSG PIC X(31). <-ALL-FMTS
+000032* EXCEPTION MESSAGE <-ALL-FMTS
89 +000033 06 OVRPMT PIC S9(6)V9(2). <-ALL-FMTS
+000034* OVERPAYMENT <-ALL-FMTS
90 +000035 06 STSCDE PIC X(1). <-ALL-FMTS
+000036* STATUS CODE <-ALL-FMTS
+000037* INPUT FORMAT:CONTROL1 FROM FILE PAYUPDTD OF LIBRARY CBLGUIDE <-ALL-FMTS
+000038* SUBFILE CONTROL <-ALL-FMTS
91 +000039 05 CONTROL1-I REDEFINES PAYUPDTD-RECORD. <-ALL-FMTS
92 +000040 06 CONTROL1-I-INDIC. <-ALL-FMTS
93 +000041 07 IN99 PIC 1 INDIC 99. <-ALL-FMTS
+000042* HELP KEY <-ALL-FMTS
94 +000043 07 IN98 PIC 1 INDIC 98. <-ALL-FMTS
+000044* END PAYMENT UPDATE <-ALL-FMTS
95 +000045 07 IN97 PIC 1 INDIC 97. <-ALL-FMTS
+000046* IGNORE INPUT <-ALL-FMTS
+000047* OUTPUT FORMAT:CONTROL1 FROM FILE PAYUPDTD OF LIBRARY CBLGUIDE <-ALL-FMTS
+000048* SUBFILE CONTROL <-ALL-FMTS

```

Figure 152. Source Listing of a Payment Update Program Example (Part 3 of 13)

```

5722WDS V5R4M0 060210 LN IBM ILE COBOL CBLGUIDE/PAYUPDT ISERIES1 06/02/15 15:08:37 Page 5
STMT PL SEQNBR -A 1 B.+. . . . 2. . . . +. . . . 3. . . . +. . . . 4. . . . +. . . . 5. . . . +. . . . 6. . . . +. . . . 7. . . . IDENTFCN S COPYNAME CHG DATE
96 +000049 05 CONTROL1-0 REDEFINES PAYUPDTD-RECORD. <-ALL-FMTS
97 +000050 06 CONTROL1-0-INDIC. <-ALL-FMTS
98 +000051 07 IN61 PIC 1 INDIC 61. <-ALL-FMTS
99 +000052 07 IN62 PIC 1 INDIC 62. <-ALL-FMTS
100 +000053 07 IN99 PIC 1 INDIC 99. <-ALL-FMTS
+000054* HELP KEY <-ALL-FMTS
101 +000055 07 IN63 PIC 1 INDIC 63. <-ALL-FMTS
102 +000056 07 IN64 PIC 1 INDIC 64. <-ALL-FMTS
+000057* INPUT FORMAT:MESSAGE1 FROM FILE PAYUPDTD OF LIBRARY CBLGUIDE <-ALL-FMTS
+000058* MESSAGE RECORD <-ALL-FMTS
+000059* 05 MESSAGE1-I REDEFINES PAYUPDTD-RECORD. <-ALL-FMTS
+000060* OUTPUT FORMAT:MESSAGE1 FROM FILE PAYUPDTD OF LIBRARY CBLGUIDE <-ALL-FMTS
+000061* MESSAGE RECORD <-ALL-FMTS
103 +000062 05 MESSAGE1-0 REDEFINES PAYUPDTD-RECORD. <-ALL-FMTS
104 +000063 06 MESSAGE1-0-INDIC. <-ALL-FMTS
105 +000064 07 IN71 PIC 1 INDIC 71. <-ALL-FMTS
004400
106 004500 WORKING-STORAGE SECTION.
004600
107 004700 01 REL-NUMBER PIC 9(05)
004800 VALUE ZEROS.
004900
108 005000 01 WS-CONTROL.
109 005100 05 WS-IND PIC X(02).
110 005200 05 WS-FORMAT PIC X(10).
111 005300 01 SYSTEM-DATE.
112 005400 05 SYSTEM-YEAR PIC 99.
113 005500 05 SYSTEM-MONTH PIC 99.
114 005600 05 SYSTEM-DAY PIC 99.
115 005700 01 PROGRAM-DATE.
116 005800 05 PROGRAM-MONTH PIC 99.
117 005900 05 PROGRAM-DAY PIC 99.
118 006000 05 PROGRAM-YEAR PIC 99.
119 006100 01 FILE-DATE REDEFINES PROGRAM-DATE
006200 PIC S9(6).
120 006300 01 EXCEPTION-STATUS.
121 006400 05 STATUS-CODE-ONE PIC XX.
122 006500 88 SUBFILE-IS-FULL VALUE '0M'.
123 006600 01 EXCEPTION-MESSAGES.
124 006700 05 MESSAGE-ONE PIC X(31)
006800 VALUE 'CUSTOMER DOES NOT EXIST '.
125 006900 05 MESSAGE-TWO PIC X(31)
007000 VALUE 'NO INVOICES EXIST FOR CUSTOMER '.
126 007100 05 MESSAGE-THREE PIC X(31)
007200 VALUE 'CUSTOMER HAS AN OVER PAYMENT OF'.
127 007300 01 PROGRAM-VARIABLES.
128 007400 05 AMOUNT-OWED PIC S9(6)V99.
129 007500 05 AMOUNT-PAID PIC S9(6)V99.
130 007600 05 INVOICE-BALANCE PIC S9(6)V99.
131 007700 01 ERRPGM-PARAMETERS.
132 007800 05 DISPLAY-PARAMETER PIC X(8)
007900 VALUE 'PAYUPDTD'.
133 008000 05 DUMMY-ONE PIC X(6)
008100 VALUE SPACES.
134 008200 05 DUMMY-TWO PIC X(6)

```

Figure 152. Source Listing of a Payment Update Program Example (Part 4 of 13)

```

STMT PL SEQNBR -A 1 B.+. . . . 2. . . . +. . . . 3. . . . +. . . . 4. . . . +. . . . 5. . . . +. . . . 6. . . . +. . . . 7. . . . IDENTFCN S COPYNAME CHG DATE
008300                                VALUE SPACES.
135 008400 05 STATUS-CODE-TWO.
136 008500 10 PRIMARY PIC X(1).
137 008600 10 SECONDARY PIC X(1).
138 008700 10 FILLER PIC X(5)
008800                                VALUE SPACES.
139 008900 05 DUMMY-THREE PIC X(10)
009000                                VALUE SPACES.
009100
140 009200 01 SWITCH-AREA.
141 009300 05 SW01 PIC 1.
142 009400 88 WRITE-DISPLAY VALUE B'1'.
143 009500 88 READ-DISPLAY VALUE B'0'.
144 009600 05 SW02 PIC 1.
145 009700 88 SUBFILE1-FORMAT VALUE B'1'.
146 009800 88 NOT-SUBFILE1-FORMAT VALUE B'0'.
147 009900 05 SW03 PIC 1.
148 010000 88 CONTROL1-FORMAT VALUE B'1'.
149 010100 88 NOT-CONTROL1-FORMAT VALUE B'0'.
150 010200 05 SW04 PIC 1.
151 010300 88 NO-MORE-TRANSACTIONS-EXIST VALUE B'1'.
152 010400 88 TRANSACTIONS-EXIST VALUE B'0'.
153 010500 05 SW05 PIC 1.
154 010600 88 CUSTOMER-NOT-FOUND VALUE B'1'.
155 010700 88 CUSTOMER-EXIST VALUE B'0'.
156 010800 05 SW06 PIC 1.
157 010900 88 NO-MORE-INVOICES-EXIST VALUE B'1'.
158 011000 88 CUSTOMER-INVOICE-EXIST VALUE B'0'.
159 011100 05 SW07 PIC 1.
160 011200 88 NO-MORE-PAYMENT-EXIST VALUE B'1'.
161 011300 88 PAYMENT-EXIST VALUE B'0'.
162 011400 05 SW08 PIC 1.
163 011500 88 INPUT-ERRORS-EXIST VALUE B'1'.
164 011600 88 NO-INPUT-ERRORS-EXIST VALUE B'0'.
165 011700 05 SW09 PIC 1.
166 011800 88 OVER-PAYMENT-DISPLAYED-ONCE VALUE B'1'.
167 011900 88 OVER-PAYMENT-NOT-DISPLAYED VALUE B'0'.
012000
168 012100 01 INDICATOR-AREA.
169 012200 05 IN99 PIC 1 INDIC 99.
170 012300 88 HELP-IS-NEEDED VALUE B'1'.
171 012400 88 HELP-IS-NOT-NEEDED VALUE B'0'.
172 012500 05 IN98 PIC 1 INDIC 98.
173 012600 88 END-OF-PAYMENT-UPDATE VALUE B'1'.
174 012700 05 IN97 PIC 1 INDIC 97.
175 012800 88 IGNORE-INPUT VALUE B'1'.
176 012900 05 IN51 PIC 1 INDIC 51.
177 013000 88 DISPLAY-ACCEPT-PAYMENT VALUE B'1'.
178 013100 88 DO-NOT-DISPLAY-ACCEPT-PAYMENT VALUE B'0'.
179 013200 05 IN52 PIC 1 INDIC 52.
180 013300 88 REVERSE-FIELD-IMAGE VALUE B'1'.
181 013400 88 DO-NOT-REVERSE-FIELD-IMAGE VALUE B'0'.
182 013500 05 IN53 PIC 1 INDIC 53.
183 013600 88 DO-NOT-DISPLAY-FIELD VALUE B'1'.
184 013700 88 DISPLAY-FIELD VALUE B'0'.

```

Figure 152. Source Listing of a Payment Update Program Example (Part 5 of 13)

```

STMT PL SEQNBR -A 1 B.+. . . . 2. . . . +. . . . 3. . . . +. . . . 4. . . . +. . . . 5. . . . +. . . . 6. . . . +. . . . 7. . . . IDENTFCN S COPYNAME CHG DATE
185 013800 05 IN54 PIC 1 INDIC 54.
186 013900 88 PROTECT-INPUT-FIELD VALUE B'1'.
187 014000 88 DO-NOT-PROTECT-INPUT-FIELD VALUE B'0'.
188 014100 05 IN55 PIC 1 INDIC 55.
189 014200 88 MAKE-FIELD-BLINK VALUE B'1'.
190 014300 88 DO-NOT-MAKE-FIELD-BLINK VALUE B'0'.
191 014400 05 IN56 PIC 1 INDIC 56.
192 014500 88 DISPLAY-OVER-PAYMENT VALUE B'1'.
193 014600 88 DO-NOT-DISPLAY-OVER-PAYMENT VALUE B'0'.
194 014700 05 IN61 PIC 1 INDIC 61.
195 014800 88 CLEAR-SUBFILE VALUE B'1'.
196 014900 88 DO-NOT-CLEAR-SUBFILE VALUE B'0'.
197 015000 05 IN62 PIC 1 INDIC 62.
198 015100 88 DISPLAY-SCREEN VALUE B'1'.
199 015200 88 DO-NOT-DISPLAY-SCREEN VALUE B'0'.
200 015300 05 IN63 PIC 1 INDIC 63.
201 015400 88 DISPLAY-ACCEPT-HEADING VALUE B'1'.
202 015500 88 DO-NOT-DISPLAY-ACCEPT-HEADING VALUE B'0'.
203 015600 05 IN64 PIC 1 INDIC 64.
204 015700 88 DISPLAY-EXCEPTION VALUE B'1'.
205 015800 88 DO-NOT-DISPLAY-EXCEPTION VALUE B'0'.
206 015900 05 IN71 PIC 1 INDIC 71.
207 016000 88 DISPLAY-ACCEPT-MESSAGE VALUE B'1'.
208 016100 88 DO-NOT-DISPLAY-ACCEPT-MESSAGE VALUE B'0'.
016200
209 016300 PROCEDURE DIVISION.
016400
210 016500 DECLARATIVES.
016600
016700 TRANSACTION-ERROR SECTION.
016800 USE AFTER STANDARD ERROR PROCEDURE
016900 PAYMENT-UPDATE-DISPLAY-FILE.
017000 WORK-STATION-ERROR-HANDLER.
211 017100 IF NOT (SUBFILE-IS-FULL) THEN
212 017200 DISPLAY 'ERROR IN PAYMENT-UPDATE' STATUS-CODE-ONE
017300 END-IF.
017400 END DECLARATIVES.
017500
017600 MAIN-PROGRAM SECTION.
017700 MAINLINE.
213 017800 OPEN I-O CUSTOMER-INVOICE-FILE
017900 CUSTOMER-MASTER-FILE
018000 PAYMENT-UPDATE-DISPLAY-FILE.
018100
214 018200 MOVE ALL B'0' TO INDICATOR-AREA
018300 SWITCH-AREA.
215 018400 ACCEPT SYSTEM-DATE FROM DATE
018500 END-ACCEPT.
216 018600 MOVE SYSTEM-YEAR TO PROGRAM-YEAR.
217 018700 MOVE SYSTEM-MONTH TO PROGRAM-MONTH.
218 018800 MOVE SYSTEM-DATE TO PROGRAM-DAY.
219 018900 SET WRITE-DISPLAY
019000 CONTROL1-FORMAT
019100 DO-NOT-DISPLAY-OVER-PAYMENT
019200 DO-NOT-PROTECT-INPUT-FIELD

```

Figure 152. Source Listing of a Payment Update Program Example (Part 6 of 13)

```

5722WDS V5R4M0 060210 LN IBM ILE COBOL CBLGUIDE/PAYUPDT ISERIES1 06/02/15 15:08:37 Page 8
STMT PL SEQNBR -A 1 B..+...2...+...3...+...4...+...5...+...6...+...7..IDENTFCN S COPYNAME CHG DATE
019300 DO-NOT-REVERSE-FIELD-IMAGE
019400 DO-NOT-MAKE-FIELD-BLINK
019500 CLEAR-SUBFILE TO TRUE.
220 019600 MOVE CORR INDICATOR-AREA TO CONTROL1-0-INDIC.
*** CORRESPONDING items for statement 220:
*** IN99
*** IN61
*** IN62
*** IN63
*** IN64
*** End of CORRESPONDING items for statement 220
221 019700 WRITE PAYMENT-UPDATE-DISPLAY-RECORD
019800 FORMAT IS 'CONTROL1'
019900 END-WRITE.
222 020000 SET DO-NOT-CLEAR-SUBFILE TO TRUE.
223 020100 PERFORM INITIALIZE-SUBFILE-RECORD 17 TIMES.
224 020200 SET DISPLAY-SCREEN TO TRUE.
225 020300 MOVE CORR INDICATOR-AREA TO CONTROL1-0-INDIC.
*** CORRESPONDING items for statement 225:
*** IN99
*** IN61
*** IN62
*** IN63
*** IN64
*** End of CORRESPONDING items for statement 225
226 020400 WRITE PAYMENT-UPDATE-DISPLAY-RECORD
020500 FORMAT IS 'CONTROL1'
020600 END-WRITE.
227 020700 READ PAYMENT-UPDATE-DISPLAY-FILE RECORD
020800 FORMAT IS 'CONTROL1'
020900 END-READ.
228 021000 MOVE CORR CONTROL1-I-INDIC TO INDICATOR-AREA.
*** CORRESPONDING items for statement 228:
*** IN99
*** IN98
*** IN97
*** End of CORRESPONDING items for statement 228
021100
021200 PERFORM PROCESS-TRANSACTION-FILE
021300 UNTIL END-OF-PAYMENT-UPDATE.
021400
230 021500 CLOSE CUSTOMER-INVOICE-FILE
021600 CUSTOMER-MASTER-FILE
021700 PAYMENT-UPDATE-DISPLAY-FILE.
231 021800 STOP RUN.
021900
022000 PROCESS-TRANSACTION-FILE.
232 022100 IF HELP-IS-NOT-NEEDED THEN
233 022200 IF IGNORE-INPUT THEN
234 022300 SET WRITE-DISPLAY
022400 CONTROL1-FORMAT
022500 CLEAR-SUBFILE
022600 DISPLAY-FIELD
022700 DO-NOT-DISPLAY-OVER-PAYMENT
022800 DO-NOT-PROTECT-INPUT-FIELD

```

Figure 152. Source Listing of a Payment Update Program Example (Part 7 of 13)

```

STMT PL SEQNBR -A 1 B.+...2...+...3...+...4...+...5...+...6...+...7..IDENTFCN S COPYNAME CHG DATE
022900          DO-NOT-REVERSE-FIELD-IMAGE
023000          DO-NOT-DISPLAY-ACCEPT-PAYMENT
023100          DO-NOT-DISPLAY-ACCEPT-HEADING
023200          DO-NOT-DISPLAY-ACCEPT-MESSAGE
023300          DO-NOT-MAKE-FIELD-BLINK TO TRUE
235 023400      MOVE CORR INDICATOR-AREA TO CONTROL1-0-INDIC
          *** CORRESPONDING items for statement 235:
          ***      IN99
          ***      IN61
          ***      IN62
          ***      IN63
          ***      IN64
          *** End of CORRESPONDING items for statement 235
236 023500      WRITE PAYMENT-UPDATE-DISPLAY-RECORD
          023600          FORMAT IS 'CONTROL1'
          023700      END-WRITE
237 023800      SET DO-NOT-CLEAR-SUBFILE TO TRUE
238 023900      MOVE 0 TO REL-NUMBER
239 024000      PERFORM INITIALIZE-SUBFILE-RECORD 17 TIMES
          024100      ELSE
240 024200          SET TRANSACTIONS-EXIST
          024300          DO-NOT-DISPLAY-ACCEPT-HEADING
          024400          DO-NOT-DISPLAY-ACCEPT-MESSAGE
          024500          DO-NOT-DISPLAY-EXCEPTION TO TRUE
241 024600      PERFORM READ-MODIFIED-SUBFILE-RECORD
242 024700      PERFORM TRANSACTION-VALIDATION
          024800          UNTIL NO-MORE-TRANSACTIONS-EXIST
243 024900      SET NO-INPUT-ERRORS-EXIST TO TRUE
244 025000      PERFORM TEST-FOR-RECORD-INPUT-ERRORS
          025100          VARYING REL-NUMBER
          025200          FROM 1
          025300          BY 1
          025400          UNTIL REL-NUMBER IS GREATER THAN 17
          025500          OR INPUT-ERRORS-EXIST
245 025600      IF NO-INPUT-ERRORS-EXIST THEN
246 025700          IF OVER-PAYMENT-DISPLAYED-ONCE THEN
247 025800              SET WRITE-DISPLAY
          025900              CONTROL1-FORMAT
          026000              DO-NOT-DISPLAY-OVER-PAYMENT
          026100              DO-NOT-PROTECT-INPUT-FIELD
          026200              DO-NOT-REVERSE-FIELD-IMAGE
          026300              DO-NOT-MAKE-FIELD-BLINK
          026400              DO-NOT-DISPLAY-ACCEPT-PAYMENT
          026500              DO-NOT-DISPLAY-ACCEPT-HEADING
          026600              DO-NOT-DISPLAY-ACCEPT-MESSAGE
          026700              DO-NOT-DISPLAY-EXCEPTION
          026800              CLEAR-SUBFILE
          026900              DISPLAY-FIELD
          027000              TO TRUE
248 027100      MOVE CORR INDICATOR-AREA TO CONTROL1-0-INDIC
          *** CORRESPONDING items for statement 248:
          ***      IN99
          ***      IN61
          ***      IN62
          ***      IN63

```

Figure 152. Source Listing of a Payment Update Program Example (Part 8 of 13)

```

STMT PL SEQNBR -A 1 B.+...2...+...3...+...4...+...5...+...6...+...7..IDENTFCN S COPYNAME CHG DATE
***      IN64
*** End of CORRESPONDING items for statement 248
249    027200      WRITE PAYMENT-UPDATE-DISPLAY-RECORD
      027300          FORMAT IS 'CONTROL1'
      027400      END-WRITE
250    027500      SET DO-NOT-CLEAR-SUBFILE TO TRUE
251    027600      MOVE 0 TO REL-NUMBER
252    027700      PERFORM INITIALIZE-SUBFILE-RECORD 17 TIMES
      027800      ELSE
253    027900          SET OVER-PAYMENT-DISPLAYED-ONCE TO TRUE
      028000      END-IF
      028100          END-IF
      028200      END-IF
      028300      END-IF.
254    028400      SET WRITE-DISPLAY, DISPLAY-SCREEN TO TRUE.
255    028500      MOVE CORR INDICATOR-AREA TO MESSAGE1-0-INDIC.
      *** CORRESPONDING items for statement 255:
      ***      IN71
      *** End of CORRESPONDING items for statement 255
256    028600      WRITE PAYMENT-UPDATE-DISPLAY-RECORD
      028700          FORMAT IS 'MESSAGE1'
      028800      END-WRITE.
257    028900      SET WRITE-DISPLAY, CONTROL1-FORMAT TO TRUE.
258    029000      MOVE CORR INDICATOR-AREA TO CONTROL1-0-INDIC.
      *** CORRESPONDING items for statement 258:
      ***      IN99
      ***      IN61
      ***      IN62
      ***      IN63
      ***      IN64
      *** End of CORRESPONDING items for statement 258
259    029100      WRITE PAYMENT-UPDATE-DISPLAY-RECORD
      029200          FORMAT IS 'CONTROL1'
      029300      END-WRITE.
260    029400      READ PAYMENT-UPDATE-DISPLAY-FILE RECORD
      029500          FORMAT IS 'CONTROL1'
      029600      END-READ.
261    029700      MOVE CORR CONTROL1-I-INDIC TO INDICATOR-AREA.
      *** CORRESPONDING items for statement 261:
      ***      IN99
      ***      IN98
      ***      IN97
      *** End of CORRESPONDING items for statement 261
      029800
      029900      READ-MODIFIED-SUBFILE-RECORD.
262    030000      READ SUBFILE PAYMENT-UPDATE-DISPLAY-FILE
      030100          NEXT MODIFIED RECORD FORMAT IS 'SUBFILE1'
263    030200      AT END SET NO-MORE-TRANSACTIONS-EXIST TO TRUE
      030300      END-READ.
      030400
      030500      TEST-FOR-RECORD-INPUT-ERRORS.
264    030600      READ SUBFILE PAYMENT-UPDATE-DISPLAY-FILE RECORD
      030700          FORMAT IS 'SUBFILE1'
      030800      END-READ.
265    030900      IF STSCDE OF SUBFILE1-I IS EQUAL TO '1' THEN

```

Figure 152. Source Listing of a Payment Update Program Example (Part 9 of 13)

```

STMT PL SEQNBR -A 1 B.+...2....+...3....+...4....+...5....+...6....+...7..IDENTFCN S COPYNAME CHG DATE
266 031000 SET INPUT-ERRORS-EXIST TO TRUE
031100 END-IF.
031200
031300 TRANSACTION-VALIDATION.
267 031400 MOVE CUST OF SUBFILE1-I OF PAYMENT-UPDATE-DISPLAY-RECORD
031500 TO CUST OF CUSTOMER-MASTER-RECORD.
268 031600 SET CUSTOMER-EXIST TO TRUE.
269 031700 READ CUSTOMER-MASTER-FILE
270 031800 INVALID KEY SET CUSTOMER-NOT-FOUND TO TRUE
031900 END-READ.
271 032000 IF CUSTOMER-EXIST THEN
272 032100 MOVE CUST OF CUSMST TO CUST OF ORDHDR
273 032200 MOVE ZEROES TO INVNUM
274 032300 SET CUSTOMER-INVOICE-EXIST TO TRUE
275 032400 PERFORM START-ON-CUSTOMER-INVOICE-FILE
276 032500 IF CUSTOMER-INVOICE-EXIST THEN
277 032600 PERFORM READ-CUSTOMER-INVOICE-RECORD
278 032700 IF CUSTOMER-INVOICE-EXIST THEN
279 032800 PERFORM CUSTOMER-MASTER-FILE-UPDATE
280 032900 MOVE AMPAID OF SUBFILE1-I TO AMOUNT-PAID
281 033000 SET PAYMENT-EXIST TO TRUE
282 033100 PERFORM PAYMENT-UPDATE
033200 UNTIL NO-MORE-INVOICES-EXIST
033300 OR NO-MORE-PAYMENT-EXIST
283 033400 IF ARBAL OF CUSTOMER-MASTER-RECORD IS NEGATIVE
284 033500 SET MAKE-FIELD-BLINK
033600 DISPLAY-FIELD
033700 DO-NOT-REVERSE-FIELD-IMAGE
033800 OVER-PAYMENT-NOT-DISPLAYED
033900 DISPLAY-OVER-PAYMENT
034000 DISPLAY-EXCEPTION
034100 DO-NOT-DISPLAY-ACCEPT-PAYMENT
034200 PROTECT-INPUT-FIELD TO TRUE
285 034300 MOVE ARBAL TO OVRPMT OF SUBFILE1-0
286 034400 MOVE MESSAGE-THREE TO ECPMSG OF SUBFILE1-0
287 034500 MOVE '0' TO STSCDE OF SUBFILE1-0
288 034600 PERFORM REWRITE-DISPLAY-SUBFILE-RECORD
034700 ELSE
289 034800 SET DO-NOT-DISPLAY-FIELD
034900 DO-NOT-DISPLAY-OVER-PAYMENT
035000 DO-NOT-REVERSE-FIELD-IMAGE
035100 DO-NOT-MAKE-FIELD-BLINK
035200 DO-NOT-DISPLAY-ACCEPT-PAYMENT
035300 PROTECT-INPUT-FIELD TO TRUE
290 035400 MOVE SPACES TO ECPMSG OF SUBFILE1-0
291 035500 MOVE ZEROES TO OVRPMT OF SUBFILE1-0
292 035600 MOVE '0' TO STSCDE OF SUBFILE1-0
293 035700 PERFORM REWRITE-DISPLAY-SUBFILE-RECORD
035800 END-IF
035900 ELSE
294 036000 PERFORM NO-CUSTOMER-INVOICE-ROUTINE
036100 END-IF
036200 ELSE
295 036300 PERFORM NO-CUSTOMER-INVOICE-ROUTINE
036400 END-IF

```

Figure 152. Source Listing of a Payment Update Program Example (Part 10 of 13)

```

STMT PL SEQNBR -A 1 B.+...2...+...3...+...4...+...5...+...6...+...7..IDENTFCN S COPYNAME CHG DATE
036500 ELSE
296 036600 SET REVERSE-FIELD-IMAGE
036700 DO-NOT-PROTECT-INPUT-FIELD
036800 DISPLAY-FIELD
036900 DO-NOT-DISPLAY-OVER-PAYMENT
037000 DO-NOT-MAKE-FIELD-BLINK
037100 DISPLAY-EXCEPTION
037200 DO-NOT-DISPLAY-ACCEPT-PAYMENT
037300 DO-NOT-PROTECT-INPUT-FIELD TO TRUE
297 037400 MOVE ZEROES TO OVRPMT OF SUBFILE1-0
298 037500 MOVE MESSAGE-ONE TO ECPMSG OF SUBFILE1-0
299 037600 MOVE '1' TO STSCDE OF SUBFILE1-0
300 037700 PERFORM REWRITE-DISPLAY-SUBFILE-RECORD
037800 END-IF.
301 037900 PERFORM READ-MODIFIED-SUBFILE-RECORD.
038000
038100 START-ON-CUSTOMER-INVOICE-FILE.
302 038200 START CUSTOMER-INVOICE-FILE
038300 KEY IS GREATER THAN COMP-KEY
303 038400 INVALID KEY SET NO-MORE-INVOICES-EXIST TO TRUE
038500 END-START.
038600
038700 READ-CUSTOMER-INVOICE-RECORD.
304 038800 READ CUSTOMER-INVOICE-FILE NEXT RECORD
305 038900 AT END SET NO-MORE-INVOICES-EXIST TO TRUE
039000 END-READ.
306 039100 IF CUST OF CUSTOMER-MASTER-RECORD
039200 IS NOT EQUAL TO CUST OF CUSTOMER-INVOICE-RECORD THEN
307 039300 SET NO-MORE-INVOICES-EXIST TO TRUE
039400 END-IF.
039500
039600 CUSTOMER-MASTER-FILE-UPDATE.
308 039700 MOVE FILE-DATE TO LSTDAT OF CUSTOMER-MASTER-RECORD.
309 039800 MOVE AMPAID OF SUBFILE1-I
039900 TO LSTAMT OF CUSTOMER-MASTER-RECORD.
310 040000 SUBTRACT AMPAID OF SUBFILE1-I
040100 FROM ARBAL OF CUSTOMER-MASTER-RECORD.
311 040200 REWRITE CUSTOMER-MASTER-RECORD
040300 INVALID KEY
312 040400 DISPLAY 'ERROR IN REWRITE OF CUSTOMER MASTER'
040500 END-REWRITE.
040600
040700 REWRITE-DISPLAY-SUBFILE-RECORD.
313 040800 MOVE AMPAID OF SUBFILE1-I TO AMPAID OF SUBFILE1-0.
314 040900 MOVE CUST OF SUBFILE1-I TO CUST OF SUBFILE1-0.
315 041000 SET WRITE-DISPLAY TO TRUE.
316 041100 SET SUBFILE1-FORMAT TO TRUE.
317 041200 MOVE CORR INDICATOR-AREA TO SUBFILE1-0-INDIC.
*** CORRESPONDING items for statement 317:
*** IN51
*** IN52
*** IN53
*** IN54
*** IN55
*** IN56
    
```

Figure 152. Source Listing of a Payment Update Program Example (Part 11 of 13)

```

STMT PL SEQNBR -A 1 B..+...2...+...3...+...4...+...5...+...6...+...7..IDENTFCN S COPYNAME CHG DATE
*** End of CORRESPONDING items for statement 317
318 041300 REWRITE SUBFILE PAYMENT-UPDATE-DISPLAY-RECORD
    041400 FORMAT IS 'SUBFILE1'
    041500 END-REWRITE.
    041600
    041700 NO-CUSTOMER-INVOICE-ROUTINE.
319 041800 IF STSCDE OF SUBFILE1-I IS EQUAL TO '1' THEN
320 041900 IF ACPMPT OF SUBFILE1-I IS EQUAL TO '*NO' THEN
321 042000 SET DO-NOT-DISPLAY-FIELD
    042100 DO-NOT-DISPLAY-OVER-PAYMENT
    042200 DO-NOT-REVERSE-FIELD-IMAGE
    042300 DO-NOT-MAKE-FIELD-BLINK
    042400 DO-NOT-DISPLAY-ACCEPT-PAYMENT
    042500 PROTECT-INPUT-FIELD
    042600 TO TRUE
322 042700 MOVE SPACES TO ECPMSG OF SUBFILE1-0
323 042800 MOVE ZEROS TO OVRPMT OF SUBFILE1-0
324 042900 MOVE '0' TO STSCDE OF SUBFILE1-0
325 043000 PERFORM REWRITE-DISPLAY-SUBFILE-RECORD
    043100 ELSE
326 043200 PERFORM CUSTOMER-MASTER-FILE-UPDATE
327 043300 SET MAKE-FIELD-BLINK
    043400 DISPLAY-FIELD
    043500 DO-NOT-REVERSE-FIELD-IMAGE
    043600 OVER-PAYMENT-NOT-DISPLAYED
    043700 DISPLAY-OVER-PAYMENT
    043800 DISPLAY-EXCEPTION
    043900 DO-NOT-DISPLAY-ACCEPT-PAYMENT
    044000 PROTECT-INPUT-FIELD
    044100 TO TRUE
328 044200 MOVE ARBAL TO OVRPMT OF SUBFILE1-0
329 044300 MOVE MESSAGE-THREE TO ECPMSG OF SUBFILE1-0
330 044400 MOVE '0' TO STSCDE OF SUBFILE1-0
331 044500 PERFORM REWRITE-DISPLAY-SUBFILE-RECORD
    044600 END-IF
    044700 ELSE
332 044800 SET REVERSE-FIELD-IMAGE
    044900 DISPLAY-FIELD
    045000 DO-NOT-PROTECT-INPUT-FIELD
    045100 DO-NOT-DISPLAY-OVER-PAYMENT
    045200 DISPLAY-EXCEPTION
    045300 DISPLAY-ACCEPT-PAYMENT
    045400 DISPLAY-ACCEPT-HEADING
    045500 DISPLAY-ACCEPT-MESSAGE
    045600 DO-NOT-MAKE-FIELD-BLINK
    045700 TO TRUE
333 045800 MOVE ZEROS TO OVRPMT OF SUBFILE1-0
334 045900 MOVE MESSAGE-TWO TO ECPMSG OF SUBFILE1-0
335 046000 MOVE '1' TO STSCDE OF SUBFILE1-0
336 046100 PERFORM REWRITE-DISPLAY-SUBFILE-RECORD
    046200 END-IF.
    046300
    046400 PAYMENT-UPDATE.
337 046500 SUBTRACT AMPAID OF CUSTOMER-INVOICE-RECORD
    046600 FROM ORDAMT OF CUSTOMER-INVOICE-RECORD

```

Figure 152. Source Listing of a Payment Update Program Example (Part 12 of 13)

Customer Payment Update Prompt

Date 11/08/96

Customer	Payment
34500	2000
40500	30000
36000	2500
12500	200
22799	4500
41900	7500
10001	5000
49500	2500
13300	3500
56900	4000

Payments that would result in overpayments or that have incorrect customer numbers are left on the display and appropriate messages are added:

Customer Payment Update Prompt
Accept Payment

Date 11/08/96

Customer	Payment	Exception Message
_____ 40500	30000	NO INVOICES EXIST FOR CUSTOMER
_____ 12500	200	NO INVOICES EXIST FOR CUSTOMER
_____ 41900	7500	NO INVOICES EXIST FOR CUSTOMER
_____ 10001	5000	CUSTOMER DOES NOT EXIST
_____ 13300	3500	NO INVOICES EXIST FOR CUSTOMER

Accept payment values: (*NO *YES)

Indicate which payments to accept:

Customer Payment Update Prompt Date 11/08/96

Accept Payment	Customer	Payment	Exception Message
*NO	40500	30000	NO INVOICES EXIST FOR CUSTOMER
*YES	12500	200	NO INVOICES EXIST FOR CUSTOMER
*NO	41900 10001	7500 5000	NO INVOICES EXIST FOR CUSTOMER CUSTOMER DOES NOT EXIST
*NO	13300	3500	NO INVOICES EXIST FOR CUSTOMER

Accept payment values: (*NO *YES)

The accepted payments are processed, and overpayment information is displayed:

Customer Payment Update Prompt Date 11/08/96

Accept Payment	Customer	Payment	Exception Message	
	12500	200	CUSTOMER HAS AN OVERPAYMENT OF	58.50
	10001	5000	CUSTOMER DOES NOT EXIST	

Part 4. Appendixes

Appendix A. Level of Language Support

COBOL Standard

Standard COBOL (as defined in the “About This Guide” on page xi) consists of eleven functional processing modules, seven of which are required and five of which are optional.

The seven required modules are: Nucleus, Sequential I-O, Relative I-O, Indexed I-O, Inter-Program Communication, Sort-Merge, and Source Text Manipulation. The five optional modules are: Intrinsic Function, Report Writer, Communication, Debug, Segmentation.

Language elements within the modules may be classified as level 1 elements and level 2 elements. Elements within nine of the modules are divided into level 1 elements and level 2 elements. Three of the modules (SORT-MERGE, REPORT WRITER, and INTRINSIC FUNCTION) contain only level 1 elements. For instance, Nucleus level 1 elements perform basic internal operations. Nucleus level 2 elements provide for more extensive and sophisticated internal processing.

The three subsets of Standard COBOL are the high subset, the intermediate subset, and the minimum subset. Each subset is composed of a level of the seven required modules: Nucleus, Sequential I-O, Relative I-O, Indexed I-O, Inter-Program Communication, Sort-Merge, and Source Text Manipulation. The five optional modules (Intrinsic Function, Report Writer, Communication, Debug and Segmentation) are not required in the three subsets of Standard COBOL.

- The high subset is composed of all language elements of the highest level of all required modules. That is:
 - Level 2 elements from Nucleus, Sequential I-O, Relative I-O, Indexed I-O, Inter-Program Communication, and Source Text Manipulation
 - Level 1 elements from Sort-Merge.
- The intermediate subset is composed of all language elements of level 1 of all required modules. That is:
 - Level 1 elements from Nucleus, Sequential I-O, Relative I-O, Indexed I-O, Inter-Program Communication, Sort-Merge, and Source Text Manipulation.
- The minimum subset is composed of all language elements of level 1 of the Nucleus, Sequential I-O, and Inter-Program Communication modules.

The five optional modules are not an integral part of any of the subsets. However, none, all, or any combination of the optional modules may be associated with any of the subsets.

ILE COBOL Level of Language Support

The ILE COBOL compiler supports:

- Level 2 of the Sequential I-O and Source-Text Manipulation
- Level 1 of the Nucleus, Relative I-O, Indexed I-O, Inter-Program Communication, and Sort-Merge modules.

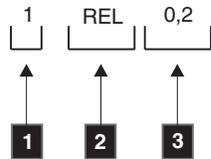
The Report Writer, Communication, Debug, and Segmentation modules of Standard COBOL are not supported by the ILE COBOL compiler.

The Intrinsic Function module of ANSI X3.23a-1989 is fully supported by the ILE COBOL compiler.

The level of support provided by the ILE COBOL compiler is represented in Table 33. The table:

- Shows the level of ILE COBOL compiler support for each functional processing module of Standard COBOL
- Describes each module.

The following is an explanation of the notation used within the table:



- 1** The level of this module supported by the ILE COBOL compiler. In this example, support is provided for Level 1 of the Relative I-O module.
- 2** A 3-character code that identifies the module. In this example, the Relative I-O module is referred.
- 3** The **range** of levels of support **defined** by Standard COBOL. A level of 0 means a **minimum** standard of COBOL does not need to support this module to conform to the standard.

Table 33. Level of ILE COBOL Compiler Support

ILE COBOL Level of Language Supported	Module Description
Nucleus 1 NUC 1,2	Contains the language elements necessary for internal processing of data within the four basic divisions of a program and the capability for defining and accessing tables.
Sequential I-O 2 SEQ 1,2	Provides access to file records by the established sequence in which they were written to the file.
Relative I-O 1 REL 0,2	Provides access to records in either a random or sequential manner. Each record is uniquely identified by an integer that represents the record's logical position in the file.
Indexed I-O 1 INX 0,2	Provides access to records in either random or sequential manner. Each record in an indexed file is uniquely identified by a record key.
Inter-program Communication 1 IPC 1,2	Allows a COBOL program to communicate with other programs through transfers of control and access to common data items.
Sort-Merge 1 SRT 0,1	Orders one or more files of records, or combines two or more identically ordered files according to user-specified keys.
Source-Text Manipulation 2 STM 0,2	Allows insertion of predefined COBOL text into a program at compile time.
Report Writer 0 RPW 0,1	Provides semiautomatic production of printed reports.

Table 33. Level of ILE COBOL Compiler Support (continued)

ILE COBOL Level of Language Supported	Module Description
Communications 0 COM 0,2	Provides the ability to access, process, and create messages or portions of messages; also allows communication through a Message Control System with local and remote communication devices.
Debug 0 DEB 0,2	Allows you to specify statements and procedures for debugging.
Intrinsic Function 1 ITR 0,1	Provides the capability to reference a data item whose value is derived automatically at the time of reference during the execution of the object program.
Segmentation 0 SEG 0,2	Provides the overlaying at object time of Procedure Division sections.

System Application Architecture® (SAA®) Common Programming Interface (CPI) Support

Source file QCBLLSRC in product library QSYSINC contains members that hold specifications for multiple SAA Common Programming Interfaces. These specifications describe parameter interfaces. This file is IBM-owned and should not be changed.

#

If you want to customize any of the specifications, you must copy any members that you want to change to a source file in one of your libraries. You can use the Copy File (CPYF) command to do this. For more information about the CPYF command, refer to the *CL and APIs* section of the *Programming* category in the **i5/OS Information Center** at this Web site -<http://www.ibm.com/systems/i/infocenter/>.

If you copy these specifications to your library, you must refresh your copies when a new product release is installed, or when any changes are made using a Program Temporary Fix (PTF). IBM provides maintenance for these specifications only in the libraries in which they are distributed.

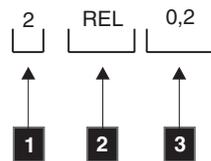
Appendix B. The Federal Information Processing Standard (FIPS) Flagger

The FIPS flagger can be specified to monitor a FIPS COBOL subset, any of the optional modules, all of the obsolete language elements, or a combination of a FIPS COBOL subset, optional modules and all obsolete elements.

The monitoring is an analysis that compares the syntax used in the source program with the syntax included in the user-selected FIPS subset and optional modules. Any syntax used in the source program that does not conform to the selected FIPS COBOL subset and optional modules is identified. Any syntax for an obsolete language element used in the source program will also be identified (depending on the compiler option chosen). See page “FLAGSTD Parameter” on page 40 for more information on the parameters for FIPS flagging.

FIPS 21-4 COBOL specifications are the language specifications contained in Standard COBOL (as described in “About This Guide” on page xi). FIPS COBOL is subdivided into three subsets and four optional modules. The three subsets are identified as Minimum, Intermediate and High. The four optional modules are Report Writer, Communication, Debug, and Segmentation. These four optional modules are not an integral part of any of the subsets; however, none, all, or any combination of the optional modules may be associated with any of the subsets. Any program written to conform to the FIPS 21-4 COBOL standard must conform to one of the subsets of FIPS 21-4 COBOL. Table 34 shows the ANSI Standard COBOL processing modules included in each of the subsets of FIPS 21-4 COBOL.

The following is an explanation of the notation used within the table:



- 1** The level of this module supported by the FIPS 21-4 COBOL Standard. In this example, support is provided for Level 2 of the Relative I-O module.
- 2** A 3-character code that identifies the module. In this example, the Relative I-O module is referred.
- 3** The **range** of levels of support **defined** by Standard COBOL. A level of 0 means a **minimum** standard of COBOL does not need to support this module to conform to the standard.

Table 34. Standard COBOL and FIPS 21-4 COBOL

ANSI Module Name	High FIPS	Intermediate FIPS	Minimum FIPS
Nucleus	2 NUC 1,2	1 NUC 1,2	1 NUC 1,2
Sequential I-O	2 SEQ 1,2	1 SEQ 1,2	1 SEQ 1,2
Relative I-O	2 REL 0,2	1 REL 0,2	0 REL 0,2
Indexed I-O	2 INX 0,2	1 INX 0,2	0 INX 0,2
Source-Text Manipulation	2 STM 0,2	1 STM 0,2	0 STM 0,2

Table 34. Standard COBOL and FIPS 21-4 COBOL (continued)

ANSI Module Name	High FIPS	Intermediate FIPS	Minimum FIPS
Sort-Merge	1 SRT 0,1	1 SRT 0,1	0 SRT 0,1
Intrinsic Function	1 ITR 0,1	0 ITR 0,1	0 ITR 0,1
Inter-Program Communication	2 IPC 1,2	1 IPC 1,2	1 IPC 1,2
Report Writer	0, or 1 RPW 0,1	0, or 1 RPW 0,1	0, or 1 RPW 0,1
Segmentation	0,1 or 2 SEG 0,2	0,1 or 2 SEG 0,2	0,1 or 2 SEG 0,2
Debug	0,1 or 2 DEB 0,2	0,1 or 2 DEB 0,2	0,1 or 2 DEB 0,2
Communications	0,1 or 2 COM 0,2	0,1 or 2 COM 0,2	0,1 or 2 COM 0,2

Elements that are specified in the ILE COBOL source program and that are not included in FIPS 21-4 COBOL are flagged as described in Appendix A, "Level of Language Support," on page 605.

Appendix C. ILE COBOL Messages

This appendix provides a general description of messages that IBM supplies with the ILE COBOL licensed program.

COBOL Message Descriptions

The messages for the ILE COBOL licensed program begin with prefixes LNC, LNM, LNP, LNR, or LNT.

- The LNC messages are issued by the COBOL syntax checker when SEU is used to enter your ILE COBOL source code. The LNC messages are also compiler-generated messages.
- The LNM messages are used as headings during a run time ILE COBOL formatted dump.
- The LNP messages are used in the ILE COBOL CL commands and menus.
- The LNR messages provide you with additional information about system operation during run time.
- The LNT messages are used as headings for various parts of the ILE COBOL compiler listing.

Message numbers are assigned as follows:

Error Message	Description
LNR7000 through LNR7199	Escape messages
LNR7200 through LNR7999	Run-time messages
LNR8000 through LNR8200	Escape messages
LNC0000 through LNC0999	Messages with severity less than 30
LNC1000 through LNC2999	Messages with severity greater than or equal to 30
LNC8000 through LNC8799	FIPS Flagger messages
LNC9001 through LNC9099	Compiler messages

Severity Levels

The ILE COBOL licensed program provides the following message severity levels:

Severity	Meaning
00	Informational: This level is used to convey information to the user. No error has occurred. Informational messages are listed only when the FLAG (00) option is specified.
10	Warning: This level indicates that an error was detected but is not serious enough to interfere with the running of the program.
20	Error: This level indicates that an error was made, but the compiler is taking a recovery that might yield the desired code.
30	Severe Error: This level indicates that a serious error was detected. Compilation is completed, but the module object is not created and running of the program cannot be attempted.
40	Unrecoverable: This level usually indicates a user error that forces termination of processing.

- 50 Unrecoverable: This level usually indicates a compiler error that forces termination of processing.
- 99 Action: Some manual action is required, such as entering a reply, changing printer forms, or replacing diskettes.

Note: 00, 10, and 20 messages are suppressed when the FLAG(30) option of the PROCESS statement is used or the CRTCLMOD/CRTBNDCBL command specifies FLAG(30) and is not overridden by the PROCESS statement. See "Using the PROCESS Statement to Specify Compiler Options" on page 51 for further information.

The compiler always attempts to provide full diagnostics of all source text in the program, even when errors have been detected. If the compiler cannot continue on a given statement, the message states that the compiler cannot continue and that it will ignore the rest of the statement. When this error occurs, the programmer should examine the entire statement.

The IBM i message facility is used to produce all messages. The ILE COBOL compiler messages reside in the message file, QLNCMSG, and the run-time messages reside in the message file, QLNRMSG.

Substitution variables and valid reply values are determined by the program sending the message, *not* by the message description stored in the message file. However, certain elements of a message description can be changed: for example, the text, severity level, default response, or dump list. To effect such changes, you need to define another message description using an Add Message Description (ADDMSGD) command, place the modified description in a user-created message file, and specify that file in the Override Message File (OVRMSGF) command. Using the OVRMSGF command allows the compiler to retrieve messages from the specified file. For additional information, see the ADDMSGD and OVRMSGF commands in the *CL and APIs* section of the *Programming* category in the **i5/OS Information Center** at this Web site -<http://www.ibm.com/systems/i/infocenter/>.

Note: If an IBM-supplied message must be changed and replaced in its message file, call your service representative.

CAUTION

Overriding an IBM-supplied message with a user-created message can produce results you do not anticipate. If reply values are not retained, the program might not respond to any replies. Changing default replies on *NOTIFY type messages could affect the ability of the program to run in unattended mode. Changing the severity could cancel a job not previously canceled. Be cautious when overriding IBM-supplied messages with user-created messages.

Compilation Messages

LNK messages are printed in the program listing when errors are found during program compilation. The LNK messages include the message issued when Federal Information Processing Standard (FIPS) flagging is requested; for more information on the FIPS messages, refer to page "The FIPS Flagger" on page 609 in this appendix.

Program Listings

In the compiler output, the ILE COBOL messages listing follows the source listing. The ILE COBOL messages listing gives the message identifier, severity, text, usually the location of the error, and the messages summary.

When the *IMBEDERR value is specified with the OPTION parameter of the CRTCBMOD or CRTBNDCBL commands, first level message text is also provided in the source listing immediately following the line on which the error was detected.

For more information about Program Listings, see “Source Listing” on page 66.

Interactive Messages

In an interactive environment, messages are displayed on the workstation display. They can appear on the current display as a result of the running of the program or in response to your keyed input to prompts, menus, command entry displays, or IBM Rational Development Studio for System i tools. The messages can also appear on request, as a result of a display command or an option on a menu.

The messages for the ILE COBOL licensed program begin with an LNC or LNR prefix.

The LNC messages are issued by the ILE COBOL syntax checker when the Source Entry Utility (SEU) is used to enter your ILE COBOL source code. For example, you see the following display after incorrectly entering the program name in the PROGRAM-ID paragraph.

```
Columns . . . : 1 71          Edit          XMPLIB/QCBLLESRC
SEU==>
FMT CB .....-A+++B+++++
***** Beginning of data *****
0000.10      IDENTIFICATION DIVISION.
0000.20      PROGRAM-ID. #TESTPR.
0000.70      ENVIRONMENT DIVISION.
0000.90      SOURCE-COMPUTER. IBM-ISERIES.
***** End of data *****
F3=Exit  F4=Prompt  F5=Refresh  F9=Retrieve  F10=Cursor
F16=Repeat find  F17=Repeat change  F24=More keys
# not in COBOL character set. Line rejected.
```

Figure 153. Example of a ILE COBOL Syntax Checker Message

The LNC messages are also issued during program compilation. See “Compilation Messages” on page 612 for a description.

LNR messages provide you with additional information about system operation during run time. For example, you might see the following display if you have a runtime error:

```

Display Program Messages
Job 008529/TESTLIB/QPADEV0003 started on 94/04/08 at 15:32:58 in subsystem Q
Message 'MCH1202' in program object 'SAMPDUMP' in library 'TESTLIB' (C D F G)

Type reply, press Enter.
Reply . . . _____

F3=Exit  F12=Cancel

```

Figure 154. Run-Time Error Message

If you move the cursor to the line on which message number MCH1202 is indicated and press either the HELP key or F1, the LNR message information is displayed as shown:

```

Additional Message Information
Message ID . . . . . : LNR7200      Severity . . . . . : 50
Message type . . . . . : Inquiry
Date sent . . . . . : 96/11/08     Time sent . . . . . : 15:33:31
Message . . . . . : Message 'MCH1202' in program object 'SAMPDUMP' in library
'TESTLIB' (C D F G).
Cause . . . . . : Message 'MCH1202' was detected in COBOL statement 42 of
COBOL program 'SAMPDUMP' in program object 'SAMPDUMP' in library 'TESTLIB'.
Recovery . . . . . : Enter a G to continue the program at the next MI
instruction, or a C if no dump is wanted, a D if a dump of the COBOL
identifiers is wanted, or an F to dump both the COBOL identifiers and the
file information. The message text for 'MCH1202' follows: 'Decimal data
error.'
Possible choices for replying to message . . . . . :
C -- No formatted dump is given
D -- A dump of the COBOL identifiers is given
More...

Press Enter to continue.
F3=Exit  F6=Print  F10=Display messages in job log
F11=Display message details  F12=Cancel  F21=Select assistance level

```

```

Additional Message Information
Message ID . . . . . : LNR7200      Severity . . . . . : 50
Message type . . . . . : Inquiry
F -- A dump of the COBOL identifiers and file information
G -- To continue the program at the next MI instruction.
Bottom

Press Enter to continue.

F3=Exit  F6=Print  F10=Display messages in job log
F11=Display message details  F12=Cancel  F21=Select assistance level

```

Figure 155. Runtime Error Message—Second-Level Text

“Responding to Messages” explains how to display second-level message text and how to reply to messages.

LNM messages 0001 to 0050 are used as headings for information printed during a ILE COBOL formatted dump.

Responding to Messages

In an interactive environment, a message is indicated by one or several of these conditions:

- A brief message (called first-level text) on the message line
- Reverse image highlighting of the input field in error

- A locked keyboard
- The sound of an alarm (if the alarm option is installed).

The following paragraphs briefly describe some methods of responding to error messages; more information is available in the IBM Rational Development Studio for System i publications.

If the necessary correction is obvious from the initial display, you can press the Error Reset key (if the keyboard is locked), enter the correct information, and continue your work.

If the message requires that you choose a reply (such as **C** to cancel, **D** to dump COBOL identifiers, **F** to dump COBOL identifiers and file information, or **G** to resume processing at the next COBOL statement), the reply options are shown in parentheses in the first-level message text. For an example, see Figure 154 on page 614.

If the information on the initial information display does not provide sufficient data for you to handle the error, you can press the HELP key (after positioning the cursor to the message line, if required) to get a second-level display with additional information about how to correct this error. To return to the initial display, press the Enter key; then press the Error Reset key (if the keyboard is locked), and make your correction or response.

If the error occurs when you are compiling or running a program, you might need to modify your ILE COBOL source statements or control language (CL) commands. Refer to the *ADTS for AS/400: Source Entry Utility* for information on how to change the statements.

Appendix D. Supporting International Languages with Double-Byte Character Sets

This appendix describes only those enhancements made to the COBOL programming language for writing programs that process double-byte characters.

Specifically, this appendix describes where you can use Double-Byte Character Set (DBCS) characters in each portion of a COBOL program, and considerations for working with DBCS data in the ILE COBOL language.

There are two ways to specify DBCS characters:

- Bracketed-DBCS
- DBCS-graphic data.

In general, COBOL handles bracketed-DBCS characters in the same way it handles alphanumeric characters. **Bracketed-DBCS** is a character string in which each character is represented by two bytes. The character string starts with a shift-out (SO) character, and ends with a shift-in (SI) character. It is up to you to know (or have the COBOL program check) which data items contain DBCS characters, and to make sure the program receives and processes this information correctly.

You can use DDS descriptions that define DBCS-graphic data fields with your ILE COBOL programs. **DBCS-graphic** pertains to a character string where each character is represented by two bytes. The character string does not contain shift-out or shift-in characters. For information on handling graphic data items specified in externally-described files in your ILE COBOL programs, refer to “DBCS-Graphic Fields” on page 451.

Using DBCS Characters in Literals

A mixed literal consists of Double-Byte Character Set (DBCS) and Single-Byte Character Set (SBCS) characters.

The GRAPHIC option of the PROCESS statement is available for processing DBCS characters in mixed literals. When the GRAPHIC option is specified, mixed literals will be handled with the assumption the hex 0E and hex 0F are shift-in and shift-out characters respectively, and they enclose the DBCS characters in the mixed literal. When NOGRAPHIC is specified or implied, the ILE COBOL compiler will treat nonnumeric literals containing hex 0E and hex 0F as if they only contains SBCS characters. Hex 0E and hex 0F are not treated as shift-in and shift-out characters, they are considered to be part of the SBCS character string.

A DBCS literal consists only of Double-Byte Character Set characters and is always treated as a DBCS character string.

Note: The GRAPHIC option on the PROCESS statement is not to be confused with the *PICXGRAPHIC or *PICGGGRAPHIC values in the CVTOPT parameter of the CRTCBMOD or CRTBND CBL command and the CVTPICXGRAPHIC and CVTPICGGGRAPHIC options on the PROCESS statement, which are used to specify double-byte graphic data from a DDS description. For more information on specifying graphic data, refer to “DBCS-Graphic Fields” on page 451.

How to Specify Literals Containing DBCS Characters

When you specify any literal that contains DBCS characters, follow the same rules that apply in specifying alphanumeric literals, as well as the following rules specific to mixed and DBCS literals:

- Mixed literals can take many different forms. The following are only two possible examples:

```
"SINGLE0E K1K2K30F BYTES"
```

```
"0E K1K20F"
```

- DBCS literals start with

```
G"0E or N"0E
```

followed by one or more Double-Byte characters and ended with

```
0F"
```

An example of this is as follows:

```
G"0E K1K20F"
```

```
N"0E 0F"
```

- Mixed literals have an implicit USAGE DISPLAY. DBCS literals have an implicit USAGE DISPLAY-1.
- EBCDIC characters can appear before or after any DBCS string in the mixed literal.
- All DBCS strings appear between shift-out and shift-in characters. A **shift-out** character is a control character (hex 0E) that indicates the start of a string of double-byte characters. A shift-out character occupies 1 byte. A **shift-in** character is a control character (hex 0F) that indicates the end of a string of double-byte characters. A shift-in character occupies 1 byte.
- Double all SBCS quotation marks that occur within the mixed literal. DBCS quotation marks within G" literals do not require doubling but DBCS quotation marks within N" literals must be doubled. For example:

```
"Mixed "0E K1K2K30F" literal"
```

```
G"0E K1K2K3"K4"K5K60F"
```

```
N"0E K1K2K3""K4""K5K60F"
```

- You can use null DBCS strings (shift-out and shift-in characters without any DBCS characters) in a mixed literal *only* when the literal contains at least one SBCS character.

The shift-out and shift-in characters cannot be nested.

The shift control characters are part of a mixed literal (not a pure DBCS literal), and take part in all operations.

Other Considerations

Quotation Marks: Although the preceding discussion uses the term *a quotation mark* to describe the character that identifies a literal, the character actually used can vary depending upon the option specified on the CRTCBMOD or CRTBNDCBL commands, or on the PROCESS statement. If you specify the APOST option, an apostrophe (') is used. Otherwise, a quotation mark (") is used. In this appendix, *a quotation mark* refers to both an apostrophe and a quotation mark. The character that you choose does not affect the rules for specifying a literal.

Shift Characters: The shift-out and shift-in characters separate EBCDIC characters from DBCS characters. They are part of the mixed literal. Therefore, the shift code

characters participate in all operations when they appear in mixed literals. Shift code characters do not participate in any operations when they appear in DBCS literals.

How the COBOL Compiler Checks DBCS Characters

When the COBOL compiler finds a DBCS string, it checks the DBCS string by scanning it one DBCS character at a time.

The following conditions cause the COBOL compiler to diagnose a literal containing DBCS characters as *not valid*:

- The syntax for the literal is incorrect.
- The mixed literal is longer than one line and does *not* follow the rules for continuing nonnumeric literals. (See “How to Continue Mixed Literals on a New Line” for more information.)
- The DBCS literal is longer than one line.

For each DBCS, mixed, or SBCS literal that is not valid, the compiler generates an error message and accepts or ignores the literal.

How to Continue Mixed Literals on a New Line

To continue a mixed literal onto another line of source code, do *all* of the following:

- Place a shift-in character in either column 71 or column 72 of the line to be continued. (If you put the shift-in character in column 71, the blank in column 72 is ignored.)
- Place a hyphen (-) in column 7 (the continuation area) of the new line.
- Place a quotation mark, then a shift-out character, and then the rest of the literal in Area B of the new line.

For example:

```
-A 1 B
  :
 01 DBCS1          PIC X(12)          VALUE "0_EK1K2K30_F
-   "0_EK4K50_F".
  :
```

The value of DBCS1 is "0_EK1K2K3K4K50_F".

The shift-in character, quotation mark, and shift-out character used to continue a line are not counted in the length of the mixed literal. The first shift-out and final shift-in characters are counted.

Syntax-Checker Considerations

When the syntax-checker is working with a line containing a literal, it has no way of knowing whether or not the user intends to specify the GRAPHIC option when the program is compiled. It, therefore, assumes that the default option, NOGRAPHIC, is in effect. This means that certain mixed literals that are valid if compiled with the GRAPHIC option will cause syntax errors to be flagged. For example:

```
"ABC0_EK1K"0_FDEF"
```

is valid when the GRAPHIC option is specified, since the double quotation mark appearing between the shift-out and shift-in characters is treated as one element of

a DBCS character. The syntax-checker, however, will mistake this double quotation mark as the termination character for the literal, and the remaining characters (starting with the shift-in character) will be flagged as an error. This may be avoided by replacing the mixed literal with a combination of SBCS nonnumeric literals and pure DBCS literals.

Where You Can Use DBCS Characters in a COBOL Program

In general, you can use mixed literals wherever nonnumeric literals are allowed. Literals for the following, however, cannot include double-byte characters:

- ALPHABET-name clause
- CURRENCY SIGN clause
- ASSIGN clause
- CLASS-name clause
- CALL statement
- CANCEL statement.

You can use DBCS literals whenever nonnumeric literals are allowed except as a literal in the following:

- ALPHABET clause
- ASSIGN clause
- CLASS clause
- CURRENCY SIGN clause
- LINKAGE clause
- CALL statement program-id
- CANCEL statement
- END PROGRAM statement
- PADDING CHARACTER clause
- PROGRAM-ID paragraph
- ACQUIRE statement
- DROP statement
- As the text-name in a COPY statement
- As the library-name in a COPY statement.

Note: You can use DBCS characters for COBOL words or names. See the *IBM Rational Development Studio for i: ILE COBOL Reference* for information on rules for formatting COBOL system-names, reserved words, and user-defined words such as data names and file names.

How to Write Comments

You can write comments containing DBCS characters in a COBOL program by putting an asterisk (*) or slash (/) in column seven of the program line. Either symbol causes the compiler to treat any information following column seven as documentation. The slash also causes a page eject. Because the COBOL compiler does not check the contents of comment lines, DBCS characters in comments are not detected. DBCS characters that are not valid can cause the compiler listing to print improperly.

Identification Division

You can put comment entries that contain DBCS characters in any portion of the Identification Division except the PROGRAM-ID paragraph. The program name specified in the PROGRAM-ID paragraph must be alphanumeric.

Environment Division

Configuration Section

You can use DBCS characters in comment entries only in the Configuration Section paragraph. All function-names, mnemonic-names, condition-names, and alphabet-names must be specified with alphanumeric characters. For the SOURCE-COMPUTER and the OBJECT-COMPUTER entry, use the alphanumeric computer name:

IBM-ISERIES

You cannot use mixed literals in the Configuration Section. Instead, use alphanumeric literals to define an alphabet-name and the literal in the CURRENCY SIGN clause of the SPECIAL-NAMES paragraph. There is no DBCS alphabet or class. Use the EBCDIC character set instead.

Input-Output Section

Specify all data names, file names, and assignment names using alphanumeric characters. You can use DBCS characters in comments.

For indexed files, the data name in the RECORD KEY clause can refer to a DBCS data item within a record.

You cannot use DBCS mixed data as the RELATIVE KEY in relative files.

File Control Paragraph

ASSIGN Clause

You cannot use literals containing DBCS characters in the ASSIGN clause to specify an external medium such as a printer or a database.

Data Division

File Section

For the FD (File Description) Entry, you can use DBCS data items or literals in the VALUE OF clause. The DATA RECORDS clause can refer to data items only. Because the ILE COBOL compiler treats both the VALUE OF clause and the DATA RECORDS clause in the File Section as documentation, neither clause has any effect when you run the program. However, the COBOL compiler checks all literals in the VALUE OF clause to make sure they are valid.

For magnetic tapes, the system can only read DBCS characters from or write DBCS characters to the tape in the EBCDIC format. The system cannot perform tape functions involving a tape in the ASCII format. Define the alphabet-name in the CODE-SET clause as NATIVE or EBCDIC.

Working-Storage Section

REDEFINES Clause

The existing rules for redefining data also apply to data that contains DBCS characters. When you determine the length of a redefining or redefined data item, remember that each DBCS character is twice as long as an alphanumeric character.

Also, ensure that redefined data items contain the shift control characters when and where necessary.

OCCURS Clause

Use this clause to define tables for storing DBCS data. If you specify the `ASCENDING/DESCENDING KEY` phrase, COBOL assumes the contents of the table are in the EBCDIC program collating sequence. The shift control characters in mixed data take part in the collating sequence.

For more information about handling tables that contain DBCS characters, see “Table Handling—SEARCH Statement” on page 628.

JUSTIFIED RIGHT Clause

Use the `JUSTIFIED RIGHT` clause to align DBCS data at the rightmost position of an elementary receiving field. If the receiving field is shorter than the sending field, COBOL truncates the rightmost characters. If the receiving field is longer than the sending field, COBOL pads (fills) the unused space on the left of the receiving field with blanks.

The `JUSTIFIED` clause does not affect the initial setting in the `VALUE` clause.

VALUE Clause

You can use mixed literals to specify an initial value for a data item that is not numeric, or to define values for level-88 condition-name entries. DBCS literals should be used to specify initial values for DBCS or DBCS-edited data items.

Any shift control characters in the literal are considered part of the literal’s picture string, except when used to continue a new line. When you continue a mixed literal, the compiler does *not* include the shift-in character in column 71 or 72, or the initial quotation mark (") and shift-out character on the continued line as part of the mixed literal. Make certain, however, that the mixed literal does not exceed the size of the data item specified in the `PICTURE` clause, otherwise truncation occurs.

DBCS literals can be used to initialize DBCS data items.

When you use literals that contain DBCS characters in the `VALUE` clause for level-88 condition-name entries, COBOL treats the DBCS characters as alphanumeric. Therefore, follow the rules for specifying alphanumeric data, including allowing a `THROUGH` option. This option uses the normal EBCDIC collating sequence, but remember that shift control characters in DBCS data take part in the collating sequence.

PICTURE Clause

Use the `PICTURE` symbol `X` to define mixed data items and either `G` or `N` for DBCS data items. You would define a DBCS data item containing n DBCS characters as

```
PICTURE G(n) or PICTURE N(n)
```

A mixed data item containing m SBCS characters, and one string of n DBCS characters would be defined as

```
PICTURE X(m+2n+2)
```

You can use all edited alphanumeric PICTURE symbols for mixed data items. The editing symbols have the same effect on the DBCS data in these items as they do on alphanumeric data items. Check that you have obtained the desired results. Pure DBCS data items can only use the B-editing symbol.

RENAMES Clause

Use this clause to specify alternative groupings of elementary data items. The existing rules for renaming alphanumeric data items also apply to DBCS data items.

Procedure Division

Intrinsic Functions

You can use DBCS data items, DBCS literals, and mixed literals as arguments to some intrinsic functions.

Intrinsic functions may also return a DBCS data item if one of the arguments of the intrinsic function is a DBCS data item or a DBCS literal.

For more information on the intrinsic functions that support DBCS items see the chapter on Intrinsic Functions in the *IBM Rational Development Studio for i: ILE COBOL Reference*.

Conditional Expressions

Because condition-names (level-88 entries) can refer to data items that contain DBCS characters, you can use the condition-name condition to test this data. (See “VALUE Clause” on page 622.) Follow the rules listed in the *IBM Rational Development Studio for i: ILE COBOL Reference* for using conditional variables and condition-names.

You can use DBCS data items or mixed literals as the operands in a relation condition. Because COBOL treats mixed data as alphanumeric, all comparisons occur according to the rules for alphanumeric operands. DBCS data items can only be compared to other DBCS data items. Keep the following in mind:

- The system does not recognize the mixed content
- The system uses the shift codes in comparisons of mixed data
- The system compares the data using either the EBCDIC collating sequence, or a user-defined sequence
- In a comparison of DBCS items with similar items of unequal size, the smaller item is padded on the right with spaces.

See “SPECIAL-NAMES” paragraph in the *IBM Rational Development Studio for i: ILE COBOL Reference* for more information.

You can use class conditions and switch status conditions as described in the *IBM Rational Development Studio for i: ILE COBOL Reference*.

Input/Output Statements

ACCEPT Statement

The input data received from a device by using a Format 1 ACCEPT statement can include DBCS. All DBCS data must be identified by the proper syntax. The input

data, excluding shift control characters, replaces the existing contents of a DBCS data item. The shift control characters are included in the contents of the mixed data items. COBOL does not perform special editing or error checking on the data.

If you use the Format 3 ACCEPT statement to get OPEN-FEEDBACK information about a file, that information includes a field showing whether the file has DBCS or mixed data.

Information received from the local data area by a Format 4 ACCEPT statement can include DBCS or mixed character strings. Information received replaces the existing contents. COBOL does not perform any editing or checking for errors. This also applies to information received from the PIP data area by a Format 5 ACCEPT statement, and from a user defined data area by a Format 9 ACCEPT statement.

Using the Format 6 ACCEPT statement, you can get the attributes of a workstation display and its keyboard. For display stations that can display DBCS characters, the system sets the appropriate value in the ATTRIBUTE-DATA data item. You cannot use DBCS characters to name a device.

If you use an extended (Format 7) ACCEPT statement for field-level workstation input, you must ensure that DBCS data is not split across lines. COBOL does not perform any checking for errors or editing, except for the removal of shift in and shift out characters when necessary.

DISPLAY Statement

You can specify DBCS or mixed data items or literals in the DISPLAY statement. You can mix the types of data. DBCS and mixed data, from either data items or literals, is sent as it appears to the program device or local data area or user-defined data area that is the target named on the DISPLAY statement.

Because COBOL does not know the characteristics of the device on which data is being displayed, you must make sure that the DBCS and mixed data is correct.

Note: ALL is a valid option for mixed literals.

If you use a Format 3 DISPLAY statement or a Format 4 DISPLAY statement for field-level workstation output, you must ensure that DBCS data is not split across lines.

READ Statement

You can use DBCS data items as the RECORD KEY for an indexed file. See "Input-Output Section" on page 621 for more information.

INTO Phrase: You can read a record into a DBCS data item using the INTO phrase. This phrase causes a MOVE statement (without the CORRESPONDING option) to be performed. The compiler moves DBCS data in the same manner that it moves alphanumeric data. It does not make sure that this data is valid.

REWRITE Statement

Use the FROM phrase of this statement to transfer DBCS data from a DBCS data item to an existing record. The FROM phrase causes both types of data to be moved in the same manner as the INTO phrase with the READ statement. (See "READ Statement.")

START Statement

If you use DBCS characters in the key of an indexed file, specify a corresponding data item in the KEY phrase of the START statement.

One of the following must be true:

- The data item must be the same as the data item specified in the RECORD KEY clause of the FILE-CONTROL paragraph.
- The data item has the same first character as the record key and is not longer than the record key.

You can specify valid operators (such as EQUAL, GREATER THAN, NOT LESS THAN) in the KEY phrase. The system can follow either the EBCDIC or a user-defined collating sequence.

WRITE Statement

Use the FROM phrase of this statement to write DBCS data to a record. This phrase moves the data in the same manner as the REWRITE statement. (See "REWRITE Statement" on page 624.)

You must include the shift control characters when you write the data into a device file.

Data Manipulation Statements

Arithmetic Statements

Because COBOL treats DBCS characters in the same manner that it treats SBCS characters, do not use DBCS characters in numeric operations, nor manipulate them with arithmetic statements.

INSPECT Statement

You can use any DBCS data item as an operand for the INSPECT statement. The system tallies and replaces on each half of a DBCS character, including the shift control characters in these operations. Therefore, the data may not be matched properly.

You can only use DBCS character operands with other DBCS character literals or data items. Mixed operands are treated as alphanumeric. If you use the REPLACING phrase, you might cause parts of an inspected mixed data item to be replaced by alphanumeric data, or parts of an inspected alphanumeric data item to be replaced by mixed data.

You cannot replace a character string with a string of a different length. Consider this when replacing SBCS characters with DBCS characters in a mixed data item, or replacing DBCS characters with SBCS characters in a mixed data item.

If you want to control the use of the INSPECT statement with mixed items containing DBCS characters, define data items containing shift control characters. Use the shift-out and shift-in characters as BEFORE/AFTER operands in the INSPECT statement.

The following example shows how you can use the INSPECT statement to replace one DBCS character with another in a mixed data item.

```

01 SUBJECT-ITEM          PICTURE X(50).
01 DBCS-CHARACTERS      VALUE "0_EK1K20_F".
   05 SHIFT-OUT         PICTURE X.
   05 DBCS-CHARACTER-1  PICTURE XX.
   05 DBCS-CHARACTER-2  PICTURE XX.
   05 SHIFT-IN          PICTURE X.

```

The INSPECT statement would be coded as follows:

```

INSPECT SUBJECT-ITEM
  REPLACING ALL DBCS-CHARACTER-1
            BY DBCS-CHARACTER-2
  AFTER INITIAL SHIFT-OUT.

```

Note: Using the AFTER INITIAL SHIFT-OUT phrase helps you to avoid the risk of accidentally replacing two consecutive alphanumeric characters that have the same EBCDIC values as DBCS-CHARACTER-1 (in cases where SUBJECT-ITEM contains mixed data).

You can also use the INSPECT statement to determine if a data item contains DBCS characters, so that appropriate processing can occur. For example:

```

01 SUBJECT-FIELD        PICTURE X(50).
01 TALLY-FIELD          PICTURE 9(3) COMP.
01 SHIFTS               VALUE "0_E0_F".
   05 SHIFT-OUT         PICTURE X.
   05 SHIFT-IN          PICTURE X.

```

In the Procedure Division you might enter the following:

```

MOVE ZERO TO TALLY-FIELD.
INSPECT SUBJECT-FIELD TALLYING TALLY-FIELD
                        FOR ALL SHIFT-OUT.
IF TALLY-FIELD IS GREATER THAN ZERO THEN
  PERFORM DBCS-PROCESSING
ELSE
  PERFORM A-N-K-PROCESSING.

```

MOVE Statement

All DBCS characters are moved as alphanumeric character strings. The system does not convert the data or examine it.

You can move mixed literals to group items and alphanumeric items. You can only move DBCS data items or DBCS literals to DBCS data items.

If the length of the receiving field is different from that of the sending field, COBOL does one of the following:

- Truncates characters from the sending item if it is longer than the receiving item. This operation can reduce data integrity.
- Pads the sending item with blanks if it is shorter than the receiving item.

To understand more about the effect of editing symbols in the PICTURE clause of the receiving data item, see the *IBM Rational Development Studio for i: ILE COBOL Reference*.

SET Statement (Condition-Name Format)

When you set the condition name to TRUE on this statement, COBOL moves the literal from the VALUE clause to the associated data item. You can move a literal with DBCS characters.

STRING Statement

You can use the STRING statement to construct a data item that contains DBCS subfields. All data in the source data items or literals, including shift control characters, is moved to the receiving data item, one-half of a DBCS character at a time.

UNSTRING Statement

The UNSTRING statement treats DBCS data and mixed data the same as alphanumeric data. The UNSTRING operation is performed on one-half of a DBCS character at a time.

Data items can contain both alphanumeric and DBCS characters within the same field.

Use the DELIMITED BY phrase to locate double-byte and alphanumeric subfields within a data field. Identify the data items containing shift control characters, and use those data items as identifiers on the DELIMITED BY phrase. See the following examples for more information on how to do this. Use the POINTER variable to continue scanning through subfields of the sending field.

After the system performs the UNSTRING operation, you can check the delimiters stored by the DELIMITER IN phrases against the shift control character values to see which subfields contain DBCS and which contain alphanumeric characters.

The following example shows how you might set up fields to prepare for the unstring operation on a character string that contain mixed data:

```
01 SUBJECT-FIELD      PICTURE X(40)
01 FILLER.
   05 UNSTRING-TABLE OCCURS 4 TIMES.
     10 RECEIVER      PICTURE X(40).
     10 DELIMTR       PICTURE X.
     10 COUNTS        PICTURE 99 COMP.
01 SHIFTS             VALUE "0_0_F".
   05 SHIFT-OUT       PICTURE X.
   05 SHIFT-IN        PICTURE X.
```

Code the UNSTRING statement as follows:

```
UNSTRING SUBJECT-FIELD DELIMITED BY SHIFT-OUT
                                OR SHIFT-IN
INTO RECEIVER (1) DELIMITER IN DELIMTR (1)
                                COUNT    IN COUNTS (1)
INTO RECEIVER (2) DELIMITER IN DELIMTR (2)
                                COUNT    IN COUNTS (2)
INTO RECEIVER (3) DELIMITER IN DELIMTR (3)
                                COUNT    IN COUNTS (3)
INTO RECEIVER (4) DELIMITER IN DELIMTR (4)
                                COUNT    IN COUNTS (4)
ON OVERFLOW PERFORM UNSTRING-OVERFLOW-MESSAGE.
```

This UNSTRING statement divides a character string into its alphanumeric and DBCS parts. Assuming that the data in the character string is valid, a delimiter value of shift-out indicates that the corresponding receiving field contains alphanumeric data, while a value of shift-in indicates that corresponding receiving field has DBCS data. You can check the COUNT data items to determine whether each receiving field received any characters. The following figure is an example that shows the results of the UNSTRING operation just described:

```

SUBJECT-FIELD = ABC0EK1K2K30FD0EK4K5K60F
RECEIVER (1) = ABC          DELIMTR (1) = 0E    COUNTS (1) = 3
RECEIVER (2) = K1K2K3      DELIMTR (2) = 0F    COUNTS (2) = 6
RECEIVER (3) = D           DELIMTR (3) = 0E    COUNTS (3) = 1
RECEIVER (4) = K4K5K6      DELIMTR (4) = 0F    COUNTS (4) = 6
SUBJECT-FIELD = 0EK1K2K30FABC0EK40F
RECEIVER (1) = (b1anks)    DELIMTR (1) = 0E    COUNTS (1) = 0
RECEIVER (2) = K1K2K3      DELIMTR (2) = 0F    COUNTS (2) = 6
RECEIVER (3) = ABC         DELIMTR (3) = 0E    COUNTS (3) = 3
RECEIVER (4) = K4          DELIMTR (4) = 0F    COUNTS (4) = 2

```

Procedure Branching Statements

You can use a mixed literal as the operand for the STOP statement. When you do, the system displays the literal as you entered it at your workstation for interactive jobs. For batch jobs, the system displays underscores where the literal would normally appear on the system operator's message queue. The system does not edit or check the contents of the literal.

Table Handling—SEARCH Statement

You can perform a Format 1 SEARCH statement (sequential search of a table) on a table that contains DBCS data half a DBCS character at a time.

You can also perform a Format 2 SEARCH statement (SEARCH ALL) against a DBCS table as well. Order the table according to the chosen collating sequence.

Note: The shift control characters in DBCS data participate in the comparison.

SORT/MERGE

You can use DBCS data items as keys in a SORT or MERGE statement. The sort operation orders data according to the collating sequence specified in the SORT, MERGE, or SPECIAL NAMES paragraph. The system orders any shift control characters contained in DBCS and mixed keys.

Use the RELEASE statement to transfer records containing DBCS characters from an input/output area to the initial phase of a sort operation. The system performs the FROM phrase with the RELEASE statement in the same way it performs the FROM phrase with the WRITE statement. (See "WRITE Statement" on page 625.)

You can also use the RETURN statement to transfer records containing DBCS characters from the final phase of a sort or merge operation to an input/output area. The system performs the INTO phrase with the RETURN statement in the same manner that it performs the INTO phrase with the READ statement. (See "READ Statement" on page 624.)

Compiler-Directing Statements

COPY Statement

You can use the COPY statement to copy source text that contains DBCS characters into a COBOL program. When you do, make sure that you specify the name of the copy book using alphanumeric data, and that you specify these names according to the rules stated in the *IBM Rational Development Studio for i: ILE COBOL Reference*.

Use the Format 2 COPY statement to copy fields defined in the data description specifications (DDS). DBCS (value in column 35 of the DDS form is G) and mixed

data items (the value in column 35 of the DDS form is O) are copied into a COBOL program in the PICTURE X(*n*) format. If *PICGGRAPHIC is selected, DBCS data items (format G) are copied in the PICTURE G(*n*) format. The compiler listing does not indicate that these fields contain DBCS characters, unless a field is a key field. In those cases, the system prints an O in the comment table for keys.

DBCS-graphic data items are copied into a COBOL program in the PICTURE X(*n*) format. The compiler listing indicates that these fields contain graphic data. See “DBCS-Graphic Fields” on page 451 for a description of the DBCS-graphic data type.

You can put DBCS characters in text comments that are copied from DDS if the associated DDS field has comments.

If you specify the REPLACING phrase of the COPY statement, consider the following:

- Pseudo-text can contain any combination of DBCS and alphanumeric characters
- You can use literals with DBCS content
- Identifiers can refer to data items that contain DBCS characters.

REPLACE Statement

The REPLACE statement resembles the REPLACING phrase of the COPY statement, except that it acts on the entire source program, not just the text in COPY libraries.

If you specify the REPLACE statement, consider the following:

- Pseudo-text can contain any combination of DBCS and alphanumeric characters
- You can use literals with DBCS content
- Identifiers can refer to data items that contain DBCS characters.

TITLE Statement

You can use DBCS literals as the literal in the TITLE statement.

Communications between Programs

You can specify entries for alphanumeric data items that contain DBCS or mixed characters, in the Linkage Section of the Data Division. If DBCS data items or DBCS literals are being passed to a program you can also define the receiving linkage section items as DBCS data items.

You can pass DBCS characters from one program to another program by specifying those data items in the USING phrase. USING BY CONTENT and USING BY VALUE, allows mixed and DBCS literals to be passed.

You cannot use DBCS characters in the CALL statement for the program-name of the called program. You cannot use DBCS characters in the CANCEL statement because they specify program-names.

FIPS Flagger

Enhancements to the COBOL language that let you use DBCS characters are flagged (identified) by the FIPS (Federal Information Processing Standard) flagger provided by the compiler as IBM extensions.

COBOL Program Listings

DBCS characters can appear in listings that originate from DBCS-capable source files, and that are produced on DBCS-capable systems.

DBCS characters that appear in a program listing originate from the source file, from source text generated by the COPY statement, or from COBOL compiler messages.

A listing containing DBCS characters should be output to a printer file that is capable of processing DBCS data. Listings containing DBCS characters are handled correctly if one of the following conditions is true:

- The source file is defined as capable of containing DBCS data using the IGCDTA parameter of the CRTSRCPF command. In this case, the program overrides the existing value of the attribute for the output printer file.
- The user has specified the required attribute for the output printer, using the IGCDTA parameter of the OVRPRTF command, before compiling the program.

Note: The IGCDTA parameter is only available on DBCS systems, and it cannot be defined or displayed on non-DBCS systems. You can, however, create objects with DBCS attributes on a non-DBCS system by copying them from a DBCS system. You should check for possible incompatibilities if you do this.

The compiler may use characters from your source program as substitution parameters in compiler and syntax checker messages. The system does not check or edit the substitution parameters. If you do not specify DBCS characters properly, the system may print or display parts of messages incorrectly.

Intrinsic Functions with Collating Sequence Sensitivity

The intrinsic functions CHAR and ORD are dependent on the ordinal positions of characters. These intrinsic functions are not supported for the DBCS data type (for example, supported for single-byte characters, alphabetic or numeric). The results of these functions are all based on the collating sequence in effect. The current CCSID does not affect the result of these intrinsic functions.

Appendix E. Example of a COBOL Formatted Dump

Figure 156 on page 632 shows an example of a COBOL formatted dump. A dump is usually available if something goes wrong when you try to run your program.

Defining a data item in the Data Division as a user-defined data type does not change how the data is represented in a dump. Data items defined using the TYPE clause behave exactly as if they had been defined without using the TYPE clause.

You can request two types of dumps, a data dump and an extended dump. The example in Figure 156 on page 632 is an extended dump.

The **data dump** contains the following information. The labels identify where on the formatted dump you will find the information.

#	A	The name of each variable
#	B	The data type
#	C	The value
#	D	The hexadecimal value

Note: Only the first 250 characters will be shown in the dump.

The **extended dump** also contains the following additional information. The labels identify where on the formatted dump you will find the information.

	E	The name of each file
	F	The system name of each file
	G	External/internal flag
	H	Last I/O operation attempted
	I	Last file status
	J	Last extended status
	K	Open/close status
	L	Blocking information
	M	Blocking factor
	N	I/O feedback area information
	O	Open feedback area information
	P	Offset in bytes of the array element

If you do not want a user to be able to see the values of your program's variables in a formatted dump, do one of the following:

- Ensure that debug data is not present in the program by removing observability.
- Give the user sufficient authority to run the program, but not to perform the formatted dump. This can be done by giving *OBJOPR plus *EXECUTE authority.

```

Source
STMT PL SEQNBR -A 1 B.+....2....+....3....+....4....+....5....+....6....+....7..IDENTFCN S COPYNAME CHG DATE
1 000100 IDENTIFICATION DIVISION.
2 000200 PROGRAM-ID. SAMPDUMP.
000300
3 000400 ENVIRONMENT DIVISION.
4 000500 CONFIGURATION SECTION.
5 000600 SOURCE-COMPUTER. IBM-ISERIES.
6 000700 OBJECT-COMPUTER. IBM-ISERIES.
7 000800 INPUT-OUTPUT SECTION.
8 000900 FILE-CONTROL.
9 001000 SELECT FILE-1 ASSIGN TO DISK-DBSRC.
11 001100 DATA DIVISION.
12 001200 FILE SECTION.
13 001300 FD FILE-1.
14 001400 01 RECORD-1.
15 001500 05 R-TYPE PIC X(1).
16 001600 05 R-AREA-CODE PIC 9(2).
17 001700 88 R-NORTH-EAST VALUES 15 THROUGH 30.
18 001800 05 R-SALES-CAT-1 PIC S9(5)V9(2) COMP-3.
19 001900 05 R-SALES-CAT-2 PIC S9(5)V9(2) COMP-3.
20 002000 05 FILLER PIC X(1).
002100
21 002200 WORKING-STORAGE SECTION.
22 002300 01 W-SALES-VALUES.
23 002400 05 W-CAT-1 PIC S9(8)V9(2).
24 002500 05 W-CAT-2 PIC S9(8)V9(2).
25 002600 05 W-TOTAL PIC S9(8)V9(2).
24 002500 01 ALPHACODE.
25 002600 05 STORECODE PIC XX OCCURS 20 TIMES indexed by PMIND.
002700
26 002800 01 W-EDIT-VALUES.
27 002900 05 FILLER PIC X(8) VALUE "TOTALS: ".
28 003000 05 W-EDIT-1 PIC Z(7)9.9(2)-.
29 003100 05 FILLER PIC X(3) VALUE SPACES.
30 003200 05 W-EDIT-2 PIC Z(7)9.9(2)-.
31 003300 05 FILLER PIC X(3) VALUE SPACES.
32 003400 05 W-EDIT-TOTAL PIC Z(7)9.9(2)-.
003500
33 003600 01 END-FLAG PIC X(1) VALUE SPACE.
34 003700 88 END-OF-INPUT VALUE "Y".
003800
35 003900 PROCEDURE DIVISION.
004000 MAIN-PROGRAM SECTION.
004100 MAINLINE.
004200*****
004300* OPEN THE INPUT FILE AND CLEAR TOTALS. *
004400*****
36 004500 OPEN INPUT FILE-1.
37 004600 MOVE ZEROS TO W-SALES-VALUES.
004700
004800*****
004900* READ THE INPUT FILE PROCESSING ONLY THOSE RECORDS FOR THE *
005000* NORTH EAST AREA. WHEN END-OF-INPUT REACHED, SET THE FLAG *
005100*****
38 005200 PERFORM UNTIL END-OF-INPUT
39 005300 READ FILE-1
40 005400 AT END SET END-OF-INPUT TO TRUE
005500 END-READ
41 005600 IF R-NORTH-EAST AND NOT END-OF-INPUT THEN
42 005700 ADD R-SALES-CAT-1 TO W-CAT-1, W-TOTAL
43 005800 ADD R-SALES-CAT-2 TO W-CAT-2, W-TOTAL
005900 END-IF
006000 END-PERFORM.
44 006100 SET PMIND to 5.
45 006200 MOVE 'Z1' TO STORECODE(PMIND).
006100
006200*****
006300* DISPLAY THE RESULTS AND END THE PROGRAM. *
006400*****
44 006500 MOVE W-CAT-1 TO W-EDIT-1.
45 006600 MOVE W-CAT-2 TO W-EDIT-2.
46 006700 MOVE W-TOTAL TO W-EDIT-TOTAL.
47 006800 DISPLAY W-EDIT-VALUES.
48 006900 STOP RUN.
***** END OF SOURCE *****

```

Figure 156. COBOL Program Used to Generate a COBOL Formatted Dump

LNR7200 exception in module 'SAMPDUMP', program 'SAMPDUMP' in library 'TESTLIB' at statement number 42.

Formatted data dump for module 'SAMPDUMP', program 'SAMPDUMP' in library 'TESTLIB'.

NAME	ATTRIBUTE	VALUE
DB-FORMAT-NAME	A	
CHAR(10)	B	"DBSRC" C
		"C4C2E2D9C34040404040"X D
END-FLAG		
CHAR(1)		" "
		"40"X
PMIND		
IX(4)		5
		"00000008"X P
R-AREA-CODE	OF RECORD-1 OF FILE-1	
ZONED(2 0)		0.
		"0000"X
R-SALES-CAT-1	OF RECORD-1 OF FILE-1	
PACKED(7 2)		00000.00
		"00000000"X
R-SALES-CAT-2	OF RECORD-1 OF FILE-1	
PACKED(7 2)		00000.72
		"0000B7A0"X
RETURN-CODE		
BIN(2)		0000.
		"0000"X
STORECODE	OF ALPHACODE	
DIM(1) (1 20)		
STORECODE	OF ALPHACODE	
CHAR(2)		
(1)		" "
		"4040"X
...		...
(5)		"Z1"
		"E9F1"X
(6)		" "
		"4040"X
...		...
W-CAT-1	OF W-SALES-VALUES	
ZONED(10 2)		00311111.08
		"F0F0F3F1F1F1F1F0F8"X
W-CAT-2	OF W-SALES-VALUES	
ZONED(10 2)		00622222.16
		"F0F0F6F2F2F2F2F1F6"X
W-EDIT-TOTAL	OF W-EDIT-VALUES	
CHAR(12)		" "
		"40404040404040404040"X
W-EDIT-1	OF W-EDIT-VALUES	
CHAR(12)		" "
		"40404040404040404040"X
W-EDIT-2	OF W-EDIT-VALUES	
CHAR(12)		" "
		"40404040404040404040"X
W-TOTAL	OF W-SALES-VALUES	
ZONED(10 2)		00933333.24
		"F0F0F9F3F3F3F3F2F4"X

E **F**
Current active file: FILE-1 (DISK-DBSRC).
Information pertaining to file FILE-1 (DISK-DBSRC).
File is internal. **G**
Last I-O operation attempted for file: READ. **H**
Last file status: '00'. **I**
Last extended file status: ' '. **J**
File is open. **K**
Blocking is in effect. **L**
Blocking factor: 17. **M**
I-O Feedback Area. **N**
Number of successful PUT operations: 0.

Figure 157. Example of a COBOL Formatted Dump (Part 1 of 2)

Appendix F. XML reference material

This appendix describes the XML exception codes that the XML parser returns in
special register XML-CODE. It also documents which well-formedness constraints
from the XML specification that the parser checks. Note that the XML parser was
ported from IBM's Enterprise COBOL and some of the error codes may not be
applicable on the i5/OS server, but they are included in the table for completeness.

RELATED REFERENCES

"XML exceptions that allow continuation"

"XML exceptions that do not allow continuation" on page 639

"XML conformance" on page 643

XML specification (www.w3c.org/XML/)

XML exceptions that allow continuation

The following table provides the exception codes that are associated with XML event EXCEPTION and that the XML parser returns in special register XML-CODE when the parser can continue processing the XML data. That is, the code is within one of the following ranges:

- 1-99
- 100,001-165,535
- 200,001-265,535

The table describes the exception and the actions that the parser takes when you request it to continue after the exception. In these descriptions, the term "XML text" means either XML-TEXT or XML-NTEXT, depending on whether the XML document that you are parsing is in an alphanumeric or national data item, respectively.

Table 35. XML exceptions that allow continuation

Code	Description	Parser action on continuation
1	The parser found an invalid character while scanning white space outside element content.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the END-OF-DOCUMENT event.
2	The parser found an invalid start of a processing instruction, element, comment, or document type declaration outside element content.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the END-OF-DOCUMENT event.
3	The parser found a duplicate attribute name.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the END-OF-DOCUMENT event.

Table 35. XML exceptions that allow continuation (continued)

Code	Description	Parser action on continuation
4	The parser found the markup character '<' in an attribute value.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the END-OF-DOCUMENT event.
5	The start and end tag names of an element did not match.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the END-OF-DOCUMENT event.
6	The parser found an invalid character in element content.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the END-OF-DOCUMENT event.
7	The parser found an invalid start of an element, comment, processing instruction, or CDATA section in element content.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the END-OF-DOCUMENT event.
8	The parser found in element content the CDATA closing character sequence ']]>' without the matching opening character sequence '<![CDATA['.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the END-OF-DOCUMENT event.
9	The parser found an invalid character in a comment.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the END-OF-DOCUMENT event.
10	The parser found in a comment the character sequence '—' (two hyphens) not followed by '>'.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the END-OF-DOCUMENT event.
11	The parser found an invalid character in a processing instruction data segment.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the END-OF-DOCUMENT event.
12	A processing instruction target name was 'xml' in lowercase, uppercase or mixed case.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the END-OF-DOCUMENT event.

Table 35. XML exceptions that allow continuation (continued)

Code	Description	Parser action on continuation
13	The parser found an invalid digit in a hexadecimal character reference (of the form ෝ).	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the END-OF-DOCUMENT event.
14	The parser found an invalid digit in a decimal character reference (of the form &#ddd;).	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the END-OF-DOCUMENT event.
15	The encoding declaration value in the XML declaration did not begin with lowercase or uppercase A through Z.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the END-OF-DOCUMENT event.
16	A character reference did not refer to a legal XML character.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the END-OF-DOCUMENT event.
17	The parser found an invalid character in an entity reference name.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the END-OF-DOCUMENT event.
18	The parser found an invalid character in an attribute value.	The parser continues detecting errors until it reaches the end of the document or encounters an error that does not allow continuation. The parser does not signal any further normal events, except for the END-OF-DOCUMENT event.
50	The document was encoded in EBCDIC, and the CCSID of the COBOL source member is a supported EBCDIC CCSID, but the document encoding declaration did not specify a recognizable encoding.	The parser uses the encoding specified by the CCSID of the COBOL source member.
51	The document was encoded in EBCDIC, and the document encoding declaration specified a supported EBCDIC encoding, but the parser does not support the CCSID of the COBOL source member.	The parser uses the encoding specified by the document encoding declaration.
52	The document was encoded in EBCDIC, and the CCSID of the COBOL source member is a supported EBCDIC CCSID, but the document encoding declaration specified an ASCII encoding.	The parser uses the encoding specified by the CCSID of the COBOL source member

Table 35. XML exceptions that allow continuation (continued)

Code	Description	Parser action on continuation
53	The document was encoded in EBCDIC, and the CCSID of the COBOL source member is a supported EBCDIC CCSID, but the document encoding declaration specified a supported Unicode encoding.	The parser uses the encoding specified by the CCSID of the COBOL source member.
54	The document was encoded in EBCDIC, and the CCSID of the COBOL source member is a supported EBCDIC CCSID, but the document encoding declaration specified a Unicode encoding that the parser does not support.	The parser uses the encoding specified by the CCSID of the COBOL source member.
55	The document was encoded in EBCDIC, and the CCSID of the COBOL source member is a supported EBCDIC CCSID, but the document encoding declaration specified an encoding that the parser does not support.	The parser uses the encoding specified by the CCSID of the COBOL source member.
56	The document was encoded in ASCII, and the CCSID of the COBOL source member is a supported ASCII CCSID, but the document encoding declaration did not specify a recognizable encoding.	The parser uses the encoding specified by the CCSID of the COBOL source member.
57	The document was encoded in ASCII, and the document encoding declaration specified a supported ASCII encoding, but the parser does not support the CCSID specified by the CCSID of the COBOL source member.	The parser uses the encoding specified by the document encoding declaration.
58	The document was encoded in ASCII, and the CCSID of the COBOL source member is a supported ASCII CCSID, but the document encoding declaration specified a supported EBCDIC encoding.	The parser uses the encoding specified by the CCSID of the COBOL source member.
59	The document was encoded in ASCII, and the CCSID of the COBOL source member is a supported ASCII CCSID, but the document encoding declaration specified a supported Unicode encoding.	The parser uses the encoding specified by the CCSID of the COBOL source member.
60	The document was encoded in ASCII, and the CCSID of the COBOL source member is a supported ASCII CCSID, but the document encoding declaration specified a Unicode encoding that the parser does not support.	The parser uses the encoding specified by the CCSID of the COBOL source member.

Table 35. XML exceptions that allow continuation (continued)

Code	Description	Parser action on continuation
61	The document was encoded in ASCII, and the CCSID of the COBOL source member is a supported ASCII CCSID, but the document encoding declaration specified an encoding that the parser does not support.	The parser uses the encoding specified by the CCSID of the COBOL source member.
100,001 - 165,535	The document was encoded in EBCDIC, and the encodings specified by the CCSID of the COBOL source member and the document encoding declaration are both supported EBCDIC CCSIDs, but are not the same. XML-CODE contains the CCSID for the encoding declaration plus 100,000.	If you set XML-CODE to zero before returning from the EXCEPTION event, the parser uses the encoding specified by the CCSID of the COBOL source member. If you set XML-CODE to the CCSID for the document encoding declaration (by subtracting 100,000), the parser uses this encoding.
200,001 - 265,535	The document was encoded in ASCII, and the encodings specified by the CCSID of the COBOL source member and the document encoding declaration are both supported ASCII CCSIDs, but are not the same. XML-CODE contains the CCSID for the encoding declaration plus 200,000.	If you set XML-CODE to zero before returning from the EXCEPTION event, the parser uses the encoding specified by the CCSID of the COBOL source member. If you set XML-CODE to the CCSID for the document encoding declaration (by subtracting 200,000), the parser uses this encoding.

RELATED TASKS

“Handling errors in XML documents” on page 294

XML exceptions that do not allow continuation

With these XML exceptions, no further events are returned from the parser, even if you set XML-CODE to zero and return control to the parser after processing the exception. Control is passed to the statement that you specify on your NOT ON EXCEPTION phrase or to the end of the parse statement if you have not coded a NOT ON EXCEPTION phrase.

Table 36. XML exceptions that do not allow continuation

Code	Description
100	The parser reached the end of the document while scanning the start of the XML declaration.
101	The parser reached the end of the document while looking for the end of the XML declaration.
102	The parser reached the end of the document while looking for the root element.
103	The parser reached the end of the document while looking for the version information in the XML declaration.
104	The parser reached the end of the document while looking for the version information value in the XML declaration.
106	The parser reached the end of the document while looking for the encoding declaration value in the XML declaration.
108	The parser reached the end of the document while looking for the standalone declaration value in the XML declaration.

Table 36. XML exceptions that do not allow continuation (continued)

Code	Description
109	The parser reached the end of the document while scanning an attribute name.
110	The parser reached the end of the document while scanning an attribute value.
111	The parser reached the end of the document while scanning a character reference or entity reference in an attribute value.
112	The parser reached the end of the document while scanning an empty element tag.
113	The parser reached the end of the document while scanning the root element name.
114	The parser reached the end of the document while scanning an element name.
115	The parser reached the end of the document while scanning character data in element content.
116	The parser reached the end of the document while scanning a processing instruction in element content.
117	The parser reached the end of the document while scanning a comment or CDATA section in element content.
118	The parser reached the end of the document while scanning a comment in element content.
119	The parser reached the end of the document while scanning a CDATA section in element content.
120	The parser reached the end of the document while scanning a character reference or entity reference in element content.
121	The parser reached the end of the document while scanning after the close of the root element.
122	The parser found a possible invalid start of a document type declaration.
123	The parser found a second document type declaration.
124	The first character of the root element name was not a letter, '_', or ':'.
125	The first character of the first attribute name of an element was not a letter, '_', or ':'.
126	The parser found an invalid character either in or following an element name.
127	The parser found a character other than '=' following an attribute name.
128	The parser found an invalid attribute value delimiter.
130	The first character of an attribute name was not a letter, '_', or ':'.
131	The parser found an invalid character either in or following an attribute name.
132	An empty element tag was not terminated by a '>' following the '/'.
133	The first character of an element end tag name was not a letter, '_', or ':'.
134	An element end tag name was not terminated by a '>'.
135	The first character of an element name was not a letter, '_', or ':'.
136	The parser found an invalid start of a comment or CDATA section in element content.
137	The parser found an invalid start of a comment.

Table 36. XML exceptions that do not allow continuation (continued)

Code	Description
138	The first character of a processing instruction target name was not a letter, '_', or '!'.
139	The parser found an invalid character in or following a processing instruction target name.
140	A processing instruction was not terminated by the closing character sequence '?>'.
141	The parser found an invalid character following '&' in a character reference or entity reference.
142	The version information was not present in the XML declaration.
143	'version' in the XML declaration was not followed by '='.
144	The version declaration value in the XML declaration is either missing or improperly delimited.
145	The version information value in the XML declaration specified a bad character, or the start and end delimiters did not match.
146	The parser found an invalid character following the version information value closing delimiter in the XML declaration.
147	The parser found an invalid attribute instead of the optional encoding declaration in the XML declaration.
148	'encoding' in the XML declaration was not followed by '='.
149	The encoding declaration value in the XML declaration is either missing or improperly delimited.
150	The encoding declaration value in the XML declaration specified a bad character, or the start and end delimiters did not match.
151	The parser found an invalid character following the encoding declaration value closing delimiter in the XML declaration.
152	The parser found an invalid attribute instead of the optional standalone declaration in the XML declaration.
153	'standalone' in the XML declaration was not followed by a '='.
154	The standalone declaration value in the XML declaration is either missing or improperly delimited.
155	The standalone declaration value was neither 'yes' nor 'no' only.
156	The standalone declaration value in the XML declaration specified a bad character, or the start and end delimiters did not match.
157	The parser found an invalid character following the standalone declaration value closing delimiter in the XML declaration.
158	The XML declaration was not terminated by the proper character sequence '?>', or contained an invalid attribute.
159	The parser found the start of a document type declaration after the end of the root element.
160	The parser found the start of an element after the end of the root element.
300	The document was encoded in EBCDIC, but the CCSID of the COBOL source member is a supported ASCII CCSID.
301	The document was encoded in EBCDIC, but the CCSID of the COBOL source member is Unicode.
302	The document was encoded in EBCDIC, but the CCSID of the COBOL source member is an unsupported CCSID.

Table 36. XML exceptions that do not allow continuation *(continued)*

Code	Description
303	The document was encoded in EBCDIC, but CCSID of the COBOL source member is unsupported and the document encoding declaration was either empty or contained an unsupported alphabetic encoding alias.
304	The document was encoded in EBCDIC, but the CCSID of the COBOL source member is unsupported and the document did not contain an encoding declaration.
305	The document was encoded in EBCDIC, but the CCSID of the COBOL source member is unsupported and the document encoding declaration did not specify a supported EBCDIC encoding.
306	The document was encoded in ASCII, but the CCSID of the COBOL source member is a supported EBCDIC CCSID.
307	The document was encoded in ASCII, but the CCSID of the COBOL source member is Unicode.
308	The document was encoded in ASCII, but the CCSID of the COBOL source member is unsupported and the document did not contain an encoding declaration.
309	The CCSID of the COBOL source member is a supported ASCII CCSID, but the document was encoded in Unicode.
310	The CCSID of the COBOL source member specified a supported EBCDIC CCSID, but the document was encoded in Unicode.
311	The CCSID of the COBOL source member specified an unsupported CCSID and the document was encoded in Unicode.
312	The document was encoded in ASCII, but the CCSID of the COBOL source member is unsupported and the document encoding declaration was either empty or contained an unsupported alphabetic encoding alias.
313	The document was encoded in ASCII, but the CCSID of the COBOL source member is unsupported and the document did not contain an encoding declaration.
314	The document was encoded in ASCII, but the CCSID of the COBOL source member is unsupported and the document encoding declaration did not specify a supported ASCII encoding.
315	The document was encoded in UTF-16 Little Endian, which the parser does not support on this platform.
316	The document was encoded in UCS4, which the parser does not support.
317	The parser cannot determine the document encoding. The document may be damaged.
318	The document was encoded in UTF-8, which the parser does not support.
319	The document was encoded in UTF-16 Big Endian, which the parser does not support on this platform.
500-999	Internal error. Please report the error to your service representative.

RELATED TASKS

“Handling errors in XML documents” on page 294

XML conformance

The XML parser included in ILE COBOL is not a conforming XML processor according to the definition in the XML specification. It does not validate the XML documents that you parse. While it does check for many well-formedness errors, it does not perform all of the actions required of a nonvalidating XML processor.

In particular, it does not process the internal document type definition (DTD internal subset). Thus it does not supply default attribute values, does not normalize attribute values, and does not include the replacement text of internal entities except for the predefined entities. Instead, it passes the entire document type declaration as the contents of XML-TEXT or XML-NTEXT for the DOCUMENT-TYPE-DESCRIPTOR XML event, which allows the application to perform these actions if required.

The parser optionally allows programs to continue processing an XML document after some errors. The purpose of this is to facilitate debugging of XML documents and processing programs.

Recapitulating the definition in the XML specification, a textual object is a well-formed XML document if:

- Taken as a whole, it conforms to the grammar for XML documents.
- It meets all the explicit well-formedness constraints given in the XML specification.
- Each parsed entity (piece of text) that is referenced directly or indirectly within the document is well formed.

The COBOL XML parser does check that documents conform to the XML grammar, except for any document type declaration. The declaration is supplied in its entirety, unchecked, to your application.

The following material is an annotation from the XML specification. The W3C is not responsible for any content not found at the original URL (www.w3.org/TR/REC-xml). All the annotations are non-normative and are shown in *italic*.

Copyright (C) 1994-2001 W3C (R) (Massachusetts Institute of Technology, Institut National de Recherche en Informatique et en Automatique, Keio University), All Rights Reserved. W3C liability, trademark, document use, and software licensing rules apply. (www.w3.org/Consortium/Legal/ipr-notice-20000612)

The XML specification also contains twelve explicit well-formedness constraints. The constraints that the COBOL XML parser checks partly or completely are shown in **bold** type:

1. **Parameter Entities (PEs) in Internal Subset:**

“In the internal DTD subset, parameter-entity references can occur only where markup declarations can occur, not within markup declarations. (This does not apply to references that occur in external parameter entities or to the external subset.)”

The parser does not process the internal DTD subset, so it does not enforce this constraint.

2. **External Subset:**

“The external subset, if any, must match the production for extSubset.”

The parser does not process the external subset, so it does not enforce this constraint.

3. **Parameter Entity Between Declarations:**
 "The replacement text of a parameter entity reference in a DeclSep must match the production extSubsetDecl."
The parser does not process the internal DTD subset, so it does not enforce this constraint.
4. **Element Type Match:**
 "The Name in an element's end-tag must match the element type in the start-tag."
The parser enforces this constraint.
5. **Unique Attribute Specification:**
 "No attribute name may appear more than once in the same start-tag or empty-element tag."
The parser partly supports this constraint by checking up to 10 attribute names in a given element for uniqueness. The application can check any attribute names beyond this limit.
6. **No External Entity References:**
 "Attribute values cannot contain direct or indirect entity references to external entities."
The parser does not enforce this constraint.
7. **No '<' in Attribute Values:**
 "The replacement text of any entity referred to directly or indirectly in an attribute value must not contain a '<'.
The parser does not enforce this constraint.
8. **Legal Character:**
 "Characters referred to using character references must match the production for Char."
The parser enforces this constraint.
9. **Entity Declared:**
 "In a document without any DTD, a document with only an internal DTD subset which contains no parameter entity references, or a document with standalone='yes', for an entity reference that does not occur within the external subset or a parameter entity, the Name given in the entity reference must match that in an entity declaration that does not occur within the external subset or a parameter entity, except that well-formed documents need not declare any of the following entities: amp, lt, gt, apos, quot. The declaration of a general entity must precede any reference to it which appears in a default value in an attribute-list declaration.
 Note that if entities are declared in the external subset or in external parameter entities, a non-validating processor is not obligated to read and process their declarations; for such documents, the rule that an entity must be declared is a well-formedness constraint only if standalone='yes'."
The parser does not enforce this constraint.
10. **Parsed Entity:**
 "An entity reference must not contain the name of an unparsed entity. Unparsed entities may be referred to only in attribute values declared to be of type ENTITY or ENTITIES."
The parser does not enforce this constraint.
11. **No Recursion:**
 "A parsed entity must not contain a recursive reference to itself, either directly or indirectly."
The parser does not enforce this constraint.

12. In DTD:

“Parameter-entity references may only appear in the DTD.”

The parser does not enforce this constraint, because the error cannot occur.

The preceding material is an annotation from the XML specification. The W3C is not responsible for any content not found at the original URL (www.w3.org/TR/REC-xml); all these annotations are non-normative. This document has been reviewed by W3C Members and other interested parties and has been endorsed by the Director as a W3C Recommendation. It is a stable document and may be used as reference material or cited as a normative reference from another document. The normative version of the specification is the English version found at the W3C site; any translated document may contain errors from the translation.

RELATED CONCEPTS

“XML parser in COBOL” on page 277

RELATED REFERENCES

XML specification (www.w3c.org/XML/)

2.8 Prolog and document type declaration (*XML specification* at www.w3.org/TR/REC-xml#sec-prolog-dtd)

XML generate exceptions

The following table shows the exception codes that can occur during XML generation. The exception codes are returned in special register XML-CODE. If one of these exceptions occurs, control is passed to the statement in the ON EXCEPTION phrase, or to the end of the XML GENERATE statement if you did not code an ON EXCEPTION phrase.

Table 37.

Code	Description
400	The receiver was too small to contain the generated XML Document. The COUNT IN data item, if specified, contains the count of character positions that were actually generated.
401	A data-name contained a character that, when converted to Unicode, was not valid in an XML element name.
411	The CCSID specified by PROCESS statement CCSID option d is not a supported single-byte CCSID.
450	The XML file already exists.
451	The existing XML file has incorrect CCSID.
600–699	Internal error. Please report the error to your service representative.
650	Internal error. OPEN, WRITE, CLOSE, or REPLACE of the stream file failed. Please report the error to your service representative.
3000–3600	Internal error with the stream file. Please report the error to your service representative.

RELATED TASKS

“Handling errors in generating XML output” on page 311

Appendix G. Migration and Compatibility Considerations between OPM COBOL/400 and ILE COBOL

This appendix describes the differences between ILE COBOL and OPM COBOL/400.

If you are moving your existing OPM COBOL/400 programs and applications to ILE COBOL, you must be aware of the following differences between the OPM COBOL/400 compiler and the ILE COBOL compiler. In some cases, changes to your programs may be required.

Migration Strategy

When migrating your existing OPM COBOL/400 programs and applications to ILE COBOL, the following migration strategy is recommended:

- Migrate an entire application (or COBOL run unit) at one time to a pure ILE environment instead of migrating one program at a time.
- Map a COBOL run unit to an ILE activation group. For example, for a COBOL run unit that contains a number of COBOL programs, you can do one of the following to preserve the COBOL run unit semantics:
 - Create all of the COBOL programs using the CRTBNDCBL command. In this case, all of the programs will run in the QILE activation group.
 - Create all of the COBOL programs using the CRTCBMOD command followed by CRTPGM with ACTGRP(anyname). In this case, all of the programs will run in the activation group named "anyname".
 - Create the first COBOL program with ACTGRP(*NEW) using the CRTPGM command and create the rest of the programs in the application with ACTGRP(*CALLER). In this case, all of the programs will run in the *NEW activation group of the first COBOL program.
- Ensure that the caller of programs created with the ACTGRP(*CALLER) option on the CRTPGM command are not OPM programs.

Note: Mixing OPM COBOL/400 and ILE COBOL programs in the same run unit is *not* recommended.

- Pay special attention to system functions that allow different scoping options. For example, default scoping of the following system functions is changed to *ACTGRPDFN (the activation group level) when used in an ILE activation group whereas they have other defaults, such as *CALLLVL (the call level), when used in OPM programs.
 - For OPNDBF and OPNQRYF, you may need to change OPNSCOPE depending on the application. For example, if the application is running in different activation groups and need to share files, you will need to change the scope to *JOB.
 - Overrides.
 - Commitment Control.
- RCLRSRC has no effect on ILE activation groups. Instead, use RCLACTGRP to clean up ILE activation groups.

Compatibility Considerations

This section describes compatibility considerations between ILE COBOL and OPM COBOL/400.

General Considerations

Area Checking

In ILE COBOL, area checking is only active for the first token on a line. Subsequent tokens are not checked to see if they are in the correct area.

The OPM COBOL/400 compiler checks all tokens.

Attributes Field in the Data Division Map Section of the Compiler Listing

In ILE COBOL, syntax checked only attributes (for example, SAME SORT AREA, SAME SORT-MERGE AREA, SAME AREA, LABEL information) are not reported in the Data Division Map section of the compiler listing.

In ILE COBOL, condition names are not listed in Data Division Map section of the compiler listing.

OPM COBOL/400 lists condition names but does not specify any attribute information.

MIXED, COMMUNICATIONS, and BSC files

MIXED, COMMUNICATIONS, and BSC files are not supported in ILE COBOL. These file types are valid in the System/38 environment and are not supported by the ILE COBOL compiler at compile time (for COPY DDS) or at run time.

Reserved Words

ILE COBOL supports a number of reserved words that are not currently supported by OPM COBOL/400. For example, SORT-RETURN and RETURN-CODE are special registers. An occurrence of SORT-RETURN or RETURN-CODE in an OPM COBOL/400 program would generate a severity 10 message which indicates that these are reserved words in other implementations of COBOL.

ILE COBOL recognizes these words as reserved words and, in similar situations, ILE COBOL issues a severity 30 message indicating that a reserved word was found where a user-defined word would be required.

Source files for SAA CPI Data Structures

In ILE COBOL, the source files for SAA CPI data structures are found in file QCBLLSRC of library QSYSINC.

In OPM COBOL/400, the source files for SAA CPI data structures are found in file QILBINC of libraries QLBL and QLBLP.

CL Commands

CRTCBLPGM Command Replaced By CRTCBMOD and CRTBNDCBL Commands

The OPM COBOL/400 compiler is invoked by the CRTCBLPGM CL command. The CRTCBLPGM CL command creates a *PGM object.

The ILE COBOL compiler is invoked by the CRTCBLMOD or CRTBNDCBL CL commands. The CRTCBLMOD CL command creates a *MODULE object and the CRTBNDCBL CL command creates a *PGM object.

The following CRTCBLPGM parameters and options (and their associated PROCESS statement options) are not found on the CRTCBLMOD and CRTBNDCBL commands:

- GENOPT parameter (all remaining GENOPT details have been moved to OPTION details)
- PRTFILE parameter
- SAAFLAG parameter
- DUMP parameter
- ITDUMP parameter
- NOSRCDBG/SRCDBG option in the OPTION parameter
- NOLSTDBG/LSTDBG option in the OPTION parameter
- PRINT/NOPRINT option in the OPTION parameter
- LIST/NOLIST option in the GENOPT parameter
- NOPATCH/PATCH option in the GENOPT parameter
- NODUMP/DUMP option in the GENOPT parameter
- NOATR/ATR option in the GENOPT parameter
- NOOPTIMIZE/OPTIMIZE option in the GENOPT parameter
- STDERR/NOSTDERR option in the GENOPT parameter
- NOEXTACCDSP/EXTACCDSP option in the GENOPT parameter
- FS21DUPKY/NOFS21DUPKY option in the GENOPT parameter.

The following parameters and options have changed:

- For the SRCFILE parameter, the default source file name is QCBLLSRC
- For the CVTOPT parameter, the GRAPHIC/NOGRAPHIC keyword in CRTCBLPGM is changed to PICXGRAPHIC/NOPIXGRAPHIC in CRTCBLMOD and CRTBNDCBL
- For the MSGLMT parameter, the default maximum severity level is 30
- For the GENLVL parameter, the default severity level is 30
- For the FLAGSTD parameter, the NOSEG/SEG1/SEG2 and NODEB/DEB1/DEB2 options in CRTCBLPGM no longer exist in CRTCBLMOD or CRTBNDCBL
- For the OPTION parameter, the default for the NOUNREF/UNREF option is changed to NOUNREF
- For the OPTION parameter, the default for the NOSECLVL/SECLVL option is changed to NOSECLVL.

The following parameters and options are new in the CRTCBLMOD and CRTBNDCBL commands:

- MODULE parameter for CRTCBLMOD only
- PGM parameter for CRTBNDCBL only
- OUTPUT parameter
- DBGVIEW parameter
- OPTIMIZE parameter
- LINKLIT parameter
- SIMPLEPGM parameter for CRTBNDCBL only
- MONOPRC/NOMONOPRC option in the OPTION parameter
- NOSTDTRUNC/STDTRUNC option in the OPTION parameter
- NOIMBEDERR/IMBEDERR option in the OPTION parameter
- NOCHGPOSSGN/CHGPOSSGN option in the OPTION parameter
- NOEVENTF/EVENTF option in the OPTION parameter
- MONOPIC/NOMONOPIC option in the OPTION parameter
- NOPICGGRAPHIC/PICGGRAPHIC option in the CVTOPT parameter

- NOPICNGRAPHIC/PICNGRAPHIC option in the CVTOPT parameter
- NOFLOAT/FLOAT option in the CVTOPT parameter
- NODATE/DATE option in the CVTOPT parameter
- NOTIME/TIME option in the CVTOPT parameter
- NOTIMESTAMP/TIMESTAMP option in the CVTOPT parameter
- NOCVTTODATE/CVTTODATE option in the CVTOPT parameter
- ENBPFCOL parameter
- PRFDTA parameter
- CCSID parameter
- ARITHMETIC parameter
- NTLPADCHAR parameter
- LICOPT parameter
- STGMDL parameter
- DBGENCKEY parameter
- BNDDIR parameter for CRTBNDCBL only
- ACTGRP parameter for CRTBNDCBL only.

All of the deletions, changes, and additions to parameters and options are also reflected in associated changes to the PROCESS statement options.

The NOGRAPHIC PROCESS statement option has been added to ILE COBOL as the default value for the GRAPHIC option on the PROCESS statement.

The following OPM COBOL/400 PROCESS statement options are not found in ILE COBOL:

- FS9MTO0M/NOFS9MTO0M
- FS9ATO0A/NOFS9ATO0A.

Coded Character Set Identifiers (CCSID)

In ILE COBOL, CCSID normalization of the source members in a compilation is to the CCSID of the primary source file. In OPM COBOL/400, it is to the CCSID of the compile time job.

Default Source Member Type

In ILE COBOL, the default source member type is CBLLE. In OPM COBOL/400, the default source member type is CBL.

Error Messages

In ILE COBOL, compile time error messages are prefixed with LNC. Also, some of the message numbers are not always the same as in OPM COBOL/400.

GENLVL Parameter

ILE COBOL will *not* generate code when an error with a severity level *greater than or equal* to the severity specified for GENLVL occurs.

OPM COBOL/400 will *not* generate code when an error with a severity level *greater than* the severity specified for GENLVL occurs.

SAA Flagging

SAA Flagging is not supported in ILE COBOL.

STRCBLDBG and ENDCBLDBG CL Commands

The STRCBLDBG and ENDCBLDBG commands are not support in ILE COBOL.

Compiler-Directing Statements

COPY Statement

Comment after Variable Length Field: In OPM COBOL/400, a DDS source with data type G and VARLEN will produce the following:

```
06 FILLER PIC X(10)
   (Variable length field)
```

ILE COBOL adds a comment after the variable length field comment, which is more accurate:

```
06 FILLER PIC X(10)
   (Variable length field)
   (Graphic field)
```

Default Source File Name: In ILE COBOL, if a source file member is being compiled, the default source file name is QCBLLSRC. If a stream file is being compiled, the stream file must be specified. In ILE COBOL, if a source file member is being compiled, a COPY statement without the source qualifier will use QCBLLSRC. If the default file name is used and the source member is not found in file QCBLLSRC then file QLBSRC will also be checked. If a stream file is being compiled, the compiler follows a different search order to resolve copy books. See the *ILE COBOL Reference* for details.

In OPM COBOL/400 the default source file name is QLBSRC.

PROCESS Statement

***CBL/*CONTROL Statement:** If *CONTROL is encountered on the PROCESS statement, then it is not handled as a directive but as an invalid PROCESS option. The *CBL/*CONTROL directive should be the only statement on a given line.

INTERMEDIATE and MINIMUM Options (FIPS Flagging): In ILE COBOL, if FIPS flagging is not requested on the CRTCBMOD or CRTBNDCBL commands, and there is a COPY statement within the PROCESS statement, no FIPS flagging will be performed against the copy member when INTERMEDIATE or MINIMUM is specified after the COPY statement. However, if INTERMEDIATE or MINIMUM is specified before the COPY statement, then FIPS flagging will be performed against the copy member.

In OPM COBOL/400, regardless of whether or not INTERMEDIATE or MINIMUM is specified before or after the COPY STATEMENT, FIPS flagging is performed against the copy member.

NOSOURCE Option: In OPM COBOL/400, when the NOSOURCE option is specified on the PROCESS statement, the Options in Effect values are printed on the compiler listing.

In ILE COBOL, when the NOSOURCE option is specified on the PROCESS statement, the Options in Effect values are not printed on the compiler listing.

USE FOR DEBUGGING

OPM COBOL/400 accepts USE FOR DEBUGGING when WITH DEBUGGING MODE is specified.

ILE COBOL does not support USE FOR DEBUGGING. Text is treated as comments until the start of the next section or the end of the DECLARATIVES. A severity 0 error message and a severity 20 error message are issued.

Environment Division

Order of DATA DIVISION and ENVIRONMENT DIVISION

OPM COBOL/400 is fairly relaxed about intermixing the order of the DATA DIVISION and ENVIRONMENT DIVISION. OPM COBOL/400 issues severity 10 and severity 20 messages when it encounters clauses, phrases, sections and divisions that are not in the proper order.

ILE COBOL does not allow the order of the DATA DIVISION and ENVIRONMENT division to be intermixed. ILE COBOL issues severity 30 messages when it encounters clauses, phrases, sections and division that are not in the proper order.

FILE-CONTROL and I-O-CONTROL Paragraphs

If a duplicate clause occurs in a FILE-CONTROL entry or I-O-CONTROL entry, and only one such clause is allowed, OPM COBOL/400 uses the last such clause specified.

In the same situation, ILE COBOL uses the first such clause specified.

SELECT Clause

The OPM COBOL/400 compiler accepts multiple SELECT clauses that refer to a given file name, if the attributes specified are consistent. In some cases no error messages are issued. In others, severity 10, or severity 20 messages are issued. In the case where attributes specified are inconsistent, severity 30 messages are issued.

The ILE COBOL compiler issues severity 30 messages in all cases of multiple SELECT clauses that refer to a given file name.

Data Division

Order of DATA DIVISION and ENVIRONMENT DIVISION

See "Order of DATA DIVISION and ENVIRONMENT DIVISION" in section "Environment Division."

FD or SD Entries

If a duplicate clause occurs in a FD entry or SD entry, and only one such clause is allowed, OPM COBOL/400 uses the last such clause specified.

In the same situation, ILE COBOL uses the first such clause specified.

WORKING-STORAGE SECTION

In ILE COBOL, the storage allocation of independent Working-Storage items does not reflect the order in which these items are declared in the Working-Storage section, as was the case in OPM COBOL/400.

The potential impact of this change in the way storage is allocated, is on those programs that use a circumvention scheme to alleviate the 32K maximum table size limitation of OPM COBOL/400. If your program uses a circumvention scheme to increase table size where multiple independent Working-Storage items are consecutively declared and range checking is turned off, then this scheme will no

longer work. If a program that uses such a scheme is run using ILE COBOL, the program will produce unpredictable results.

For ILE COBOL, the maximum table size is now 16 711 568 bytes and thus the problem that triggered this circumvention scheme no longer exists. However, any programs that use this circumvention scheme will have to be recoded.

LIKE Clause

When a REDEFINES clause is found after a LIKE clause, the OPM COBOL/400 compiler issues a severity 20 message indicating that the REDEFINES clause has been ignored because it occurs after a LIKE clause.

In the same situation, the ILE COBOL compiler issues a severity 10 message when the REDEFINES clause is encountered and accepts the REDEFINES clause, but it also issues a severity 30 message indicating the LIKE clause is not compatible with the REDEFINES clause.

This scenario may occur in the case of other incompatible clauses such as LIKE and USAGE, or LIKE and PICTURE.

LINAGE Clause

OPM COBOL/400 flags a signed LINAGE integer with message LBL1350, but issues no message for signed FOOTING, TOP, and BOTTOM.

ILE COBOL issues message LNC1350 in all 4 cases.

PICTURE Clause

The PICTURE string `.$$` is not accepted by the ILE COBOL compiler. Similarly, the PICTURE strings `+.$$` and `-.$$` are not accepted either.

When CR or DB appear on character positions 30 and 31 of a character string, they are not accepted as valid by the ILE COBOL compiler. The entire PICTURE string must be contained within the 30 characters.

REDEFINES Clause

OPM COBOL/400 initializes redefined items.

ILE COBOL does not initialize redefined items. The initial value is determined by the default value of the original data item.

VALUE Clause

In ILE COBOL, a numeric literal specified in the VALUE clause will be truncated if its value is longer than the PICTURE string defining it. In OPM COBOL/400, a value of 0 will be assumed.

Procedure Division

General Considerations

Binary Data Items: In OPM COBOL/400, when you have data in binary data items, where the value in the item exceeds the value described by the picture clause, you will get unpredictable results. In general, when this item is used, it may or may not be truncated to the actual number of digits described by the picture clause. It usually depends on whether a PACKED intermediate is used to copy the value.

In ILE COBOL, you will also get unpredictable results, but they will be different from those generated by OPM COBOL/400.

8-Byte Binary Data Alignment: In OPM COBOL/400, 8-byte binary items are aligned with 4-byte boundaries if the *SYNC option is specified on the GENOPT parameter of the CRTCLPGM command.

In ILE COBOL, 8-byte binary items are aligned with 8-byte boundaries if the *SYNC option is specified on the OPTION parameter of the CRTCLMOD or CRTBNDCBL commands.

Duplicate Paragraph Names: When duplicate paragraph names are found in a COBOL program, the OPM COBOL/400 compiler generates a severity 20 message.

In the same situation, the ILE COBOL compiler generates a severity 30 message.

Number of Subscript: When an incorrect number of subscript are specified for an item (too many, two few, none for an item which requires them, or specified for an item which does not require them), a severity 30 message is generated in ILE COBOL.

In the same situation, OPM COBOL/400 generates a severity 20 message.

Segmentation: Segmentation is not supported in ILE COBOL. Consequently, syntax checking of segment numbers is not performed.

Common Phrases

(NOT) ON EXCEPTION Phrase: The (NOT) ON EXCEPTION PHRASE has been added to the DISPLAY statement. The addition of these phrases could require you to add the END-DISPLAY scope delimiter to prevent compile time errors.

For example:

```
ACCEPT B AT LINE 3 COLUMN 1
  ON EXCEPTION
    DISPLAY "IN ON EXCEPTION"
  NOT ON EXCEPTION
    MOVE A TO B
END-ACCEPT.
```

Both the ON EXCEPTION and NOT ON EXCEPTION phrases were meant for the ACCEPT statement; however, without an END-DISPLAY as shown below the NOT ON EXCEPTION would be considered part of the DISPLAY statement.

```
ACCEPT B AT LINE 3 COLUMN 1
  ON EXCEPTION
    DISPLAY "IN ON EXCEPTION"
  END-DISPLAY
  NOT ON EXCEPTION
    MOVE A TO B
END-ACCEPT.
```

INVALID KEY Phrase: In ILE COBOL, the INVALID KEY phrase is not allowed for sequential access of relative files since the meaning of the invalid key would be indeterminate under these circumstances. The ILE COBOL compiler issues a severity 30 error message in this situation.

The OPM COBOL/400 compiler does not issue any error messages in this situation.

ON SIZE ERROR Phrase: For arithmetic operations and conditional expressions in ILE COBOL, when ON SIZE ERROR is not specified and a size error occurs, the results are unpredictable. The results may be different than those that existed in OPM COBOL/400.

For arithmetic operations and conditional expressions in ILE COBOL, when ON SIZE ERROR is not specified and a divide by zero occurs, the results are unpredictable. The results may be different than those that existed in OPM COBOL/400.

DECLARATIVE Procedures

Declarative Implemented as an ILE Procedure: In ILE COBOL, each DECLARATIVE procedure is an ILE procedure. Thus, each DECLARATIVE procedure run in its own invocation separate from other declaratives and separate from the non-declarative part of the ILE COBOL program. As a result, using invocation sensitive system facilities such as sending and receiving messages, RCLRSC CL command, and overrides will be different in ILE COBOL than in OPM COBOL/400.

Invoking a Declarative from Another Declarative: In ILE COBOL, a declarative may be invoked from another declarative due to an I-O error provided that the former declarative is not already invoked for any reason.

OPM COBOL/400 prevents a declarative from being invoked from another declarative due to an I-O error.

Expressions

Class Condition Expressions: In ILE COBOL, the identifier in a class condition expression cannot be a group item containing one or more signed, numeric elementary items.

Abbreviated Conditional Expressions: For ILE COBOL, the use of parentheses in abbreviated combined relational conditions is not valid. OPM COBOL/400 does not enforce this rule.

Comparing Figurative Constants with Figurative Constants: In OPM COBOL/400, when a figurative constant is compared with another figurative constant, a severity 20 error message is issued and the statement is accepted.

In ILE COBOL, when a figurative constant is compared with another figurative constant, a severity 30 error message is issued and the statement is rejected.

Comparison of Zoned and Non-numeric Items: When comparing zoned items to a non-numeric item, OPM COBOL/400 issues a severity 20 message. ILE COBOL does not issue such a message.

NOT in a Relational Expression: The expression " A NOT NOT = B " is accepted by OPM COBOL/400 but a severity 20 message is generated.

In the same situation, ILE COBOL generates a severity 30 message.

NOT LESS THAN OR EQUAL TO: ILE COBOL allows some forms of conditional expression that are not permitted by OPM COBOL/400. In particular, these include NOT LESS THAN OR EQUAL and NOT GREATER THAN OR EQUAL.

Special Registers

DEBUG-ITEM Special Register: ILE COBOL no longer supports the DEBUG-ITEM special register. When encountered, it is syntax-checked only.

LINAGE-COUNTER Special Register: In OPM COBOL/400, when an integer occurs in the LINAGE clause, the LINAGE-COUNTER is defined as a 2-byte, 5-digit binary item.

In ILE COBOL, when an integer occurs in the LINAGE clause, the LINAGE-COUNTER is defined as a 4-byte, 9-digit binary item.

WHEN-COMPILED Special Register: In OPM COBOL/400, the WHEN-COMPILED special register can be used with just the MOVE statement.

In ILE COBOL, the WHEN-COMPILED special register can be used with any statement.

Extended ACCEPT and DISPLAY Statements

Compile Time Considerations: OPM COBOL/400 requires a value of EXTACCDSP in the GENOPT parameter of the CRTCLPGM command in order to enable the extended ACCEPT and DISPLAY statements. The EXTACCDSP option does not exist in the CRTCLMOD/CRTBNDCBL commands for ILE COBOL. In ILE COBOL, extended ACCEPT and DISPLAY statements are always enabled. Since the EXTACCDSP option no longer exists on the PROCESS statement for ILE COBOL, any OPM COBOL/400 program that specifies this option on its PROCESS statement may behave differently when it is compiled using the ILE COBOL compiler. The ILE COBOL compiler determines whether an ACCEPT or DISPLAY statement is extended by looking for CONSOLE IS CRT in the SPECIAL NAMES paragraph or by looking for phrases found in the Format 7 ACCEPT statement or the Format 3 DISPLAY statement.

In ILE COBOL, the following are COBOL reserved words at all times:

- AUTO
- BEEP
- BELL
- FULL
- BLINK
- COL
- COLUMN
- PROMPT
- UPDATE
- NO-ECHO
- REQUIRED
- AUTO-SKIP
- HIGHLIGHT
- UNDERLINE
- ZERO-FILL
- EMPTY-CHECK
- LEFT-JUSTIFY
- LENGTH-CHECK
- REVERSE-VIDEO
- RIGHT-JUSTIFY
- TRAILING-SIGN.

In OPM COBOL/400, only fixed tables are supported by the DISPLAY statement. In ILE COBOL, any table is supported by both the ACCEPT and DISPLAY statements.

Reference modified data is supported in the extended ACCEPT and DISPLAY statements for ILE COBOL. It is not supported in OPM COBOL/400.

In OPM COBOL/400 you have to use the *NOUNDSPCHR option to be able to use the extended character set in addition to the basic DBCS character set. In ILE COBOL, you can use either the *NOUNDSPCHR or *UNDSPCHR and still manage DBCS characters properly.

The OPM COBOL/400 compiler issues a severity 30 error message when it encounters a data item whose length is longer than the screen's capacity. The ILE COBOL compiler does not issue such an error.

In ILE COBOL, a severity 30 error message is issued when an identifier or integer on the COLUMN phrase exceeds 8 digits. OPM COBOL/400 does not issue an error.

For syntax checked only phrases in the extended DISPLAY statement, the ILE COBOL compiler performs complete syntax checking of the PROMPT, BACKGROUND-COLOR, and FOREGROUND-COLOR phrases. If any of these phrases are coded incorrectly, the ILE COBOL compiler issues severity 30 error messages. The OPM COBOL/400 compiler does not perform complete syntax checking on the PROMPT, BACKGROUND-COLOR, and FOREGROUND-COLOR phrases and does not issue any compile time error messages.

Run Time Considerations: In OPM COBOL/400, the PRINT key is disabled during the extend ACCEPT operation. In ILE COBOL, the PRINT KEY will be enabled at all times, unconditionally.

In OPM COBOL/400, the SIZE phrase is supported in the DISPLAY statement only. In ILE COBOL, the SIZE phrase is supported in both the ACCEPT and DISPLAY statements. When the specified size is greater than the size implied by the PICTURE clause data length, then OPM COBOL/400 pads blanks to the left when alphanumeric data is justified. ILE COBOL always pads blanks to the right.

In OPM COBOL/400, error message LBE7208 is issued when the data item cannot fit within the screen. In ILE COBOL, alphanumeric data that does not fit within the screen is truncated and numeric data that does not fit within the screen is not displayed. No runtime errors are issued.

When the HELP and CLEAR keys are used to complete the ACCEPT operation on a workstation attached to a 3174 or 3274 remote controller, a runtime error will be issued by ILE COBOL. OPM COBOL/400 will successfully complete this ACCEPT operation without issuing a runtime error.

OPM COBOL/400 always updates all the fields which are handled by the ACCEPT statement. ILE COBOL updates only the fields that the user has changed before pressing the ENTER key for one ACCEPT statement. As a result, the two compilers behave differently in three situations:

- When the SECURE phrase is specified on the ACCEPT statement and no value is entered
- When the ACCUPDNE option is in effect and data that is not numeric edited is handled by the ACCEPT statement

- When the field is predisplayed with alphanumeric data and the RIGHT-JUSTIFIED phrase is specified on the ACCEPT statement.

CALL Statement

Lower Case Characters in CALL/CANCEL Literal or Identifier: OPM COBOL/400 allows the CALL/CANCEL literal or identifier to contain lower case characters; however, a program object name that is not a quoted system name (extended name) does not allow lower case characters,. This means the resulting the CALL/CANCEL operation will fail.

ILE COBOL supports two new values on the OPTION parameter of the CRTCBMOD and CRTBNDCBL commands: *MONOPRC and *NOMONOPRC. The default value, *MONOPRC, causes any lower case letters in the CALL/CANCEL literal or identifier to be converted to upper case. The *NOMONOPRC value specifies that the CALL/CANCEL literal or identifier is not to be converted to upper case.

Passing a File-Name on the USING Phrase: Both OPM COBOL/400 and ILE COBOL allow a file-name to be passed on the USING phrase of the CALL statement; however, OPM COBOL/400 passes a pointer to a FIB (file information block), whereas ILE COBOL passes a pointer to a NULL pointer.

Recursive Calls: ILE COBOL allows recursive programs to be called recursively. ILE COBOL generates a runtime error message when recursion is detected in a non recursive program..

OPM COBOL/400 does not prevent recursion. However, if recursion is attempted with OPM COBOL/400 the results may be unpredictable.

CANCEL Statement

In ILE COBOL, the CANCEL statement will only cancel ILE COBOL programs within the same activation group. In ILE COBOL, a list of called program objects is maintained at the activation group (run unit) level. If the program to cancel does not appear in this list, the cancel is ignored.

In OPM COBOL/400, the CANCEL statement will issue an error message if the program to cancel does not exist in the library list.

COMPUTE Statement

In some cases, the result of exponentiation in ILE COBOL may be slightly different than the results of exponentiation in OPM COBOL/400.

When a COMPUTE statement of an exponentiation expression with a negative value for the mantissa and a negative fractional value for the exponent is performed, OPM COBOL/400 yields undefined results. In the same situation, ILE COBOL generates a CEE2020 exception.

```
# The result of a COMPUTE statement that is performed in fixed-point arithmetic
# may be slightly different from the result in OPM COBOL/400 if the expression
# contains an exponentiation operation. In ILE COBOL, the exponentiation operation
# is performed in floating point arithmetic internally. When there are no floating
# point data items in the COMPUTE statement, the result of the exponentiation is
# converted to fixed-point format to compute the rest of the expression. This
# conversion can cause slight differences with the result of OPM COBOL/400.
```

DELETE Statement

In OPM COBOL/400, file status is set to 90 when a record format that is not valid for a file is use on the DELETE statement.

In ILE COBOL, file status is set to 9K when a record format that is not valid for a file is use on the DELETE statement.

EVALUATE Statement

In OPM COBOL/400, when the WHEN phrase is specified with ZERO THRU *alphabetic-identifier*, the statement is allowed and no diagnostic message is issued.

In the same situation, ILE COBOL issues a severity 30 error message.

Note: ILE COBOL has relaxed this rule in the case of *alphanumeric-identifier* THRU *alphabetic-identifier* since the alphanumeric-identifier can contain only alphabetic characters.

IF Statement

OPM COBOL/400 has a limit of 30 for the nesting depth of IF statements.

ILE COBOL has no practical limit to the nesting depth of IF statements.

In OPM COBOL/400, when the NEXT SENTENCE phrase is used in the same IF statement as the END-IF phrase, control passes to the statement following the END-IF phrase.

In the same situation in ILE COBOL, control passes to the statement following the next separator period, that is, to the first statement of the next sentence.

Note: The OPM COBOL/400 Reference manual states that the expected behaviour is the same as what actually occurs for ILE COBOL.

INSPECT Statement

ILE COBOL supports reference modification in the INSPECT statement.

OPM COBOL/400 does not include this support.

MOVE Statement

Alphanumeric Literals and Index Names: When an alphanumeric literal is moved to an index name, OPM COBOL/400 issues a severity 20 error message. In the same situation, ILE COBOL issues a severity 30 error message.

Alphanumeric Values and Numeric-Edited Literals: When an alphanumeric value containing only numeric characters is moved to a numeric-edited literal (for example, MOVE "12.34" TO NUMEDIT), OPM COBOL/400 defaults the literal to 0. In the same situation, ILE COBOL issues a severity 30 error message.

Boolean Values: OPM COBOL/400 allows a Boolean value to be moved to a reference-modified alphabetic identifier. ILE COBOL does not allow this and issues a severity 30 error message.

CORRESPONDING Phrase: The MOVE, ADD, and SUBTRACT CORRESPONDING statements in ILE COBOL use a difference algorithm from OPM COBOL/400 to determine which items correspond. ILE COBOL could generate a severity 30 error message where in OPM COBOL/400, no message would be issued.

```

01 A.
  05 B.
    10 C PIC X(5).
    05 C PIC X(5).
01 D.
  05 B.
    10 C PIC X(5).
    05 C PIC X(5).
MOVE CORRESPONDING A TO D.

```

OPM COBOL/400 issues no message; ILE COBOL will issue message LNC1463.

Overlapping Source and Target Strings: If source and target strings are overlapping for a MOVE statement, the result is unpredictable. The move may not behave as it did for OPM COBOL/400 in the same situation.

OPEN Statement

Dynamic File Creation: There are two compatibility issues regarding dynamic file creation:

- The OPM COBOL/400 compiler supported the dynamic creation of indexed files.

The ILE COBOL compiler does *not* provide this support.

- A file will be dynamically created only if it is assigned to a COBOL device type of DISK.

OPM COBOL/400 creates files (database files) that are assigned to COBOL device types other than DISK if there is an override to a database file (OVRDBF).

Opening FORMATFILES: In ILE COBOL, FORMATFILES can only be opened for OUTPUT. The WRITE statement can be used to write output records to the FORMATFILE.

In OPM COBOL/400, FORMATFILES can be opened for INPUT, I-O, and OUTPUT.

OPEN OUTPUT or OPEN I-O for OPTIONAL Files: In ILE COBOL, OPEN OUTPUT or OPEN I-O for OPTIONAL files will not create the file, if it does not exist, when the file's organization is INDEXED.

In OPM COBOL/400, the file is created.

PERFORM Statement

In ILE COBOL, within the VARYING...AFTER phrase of the PERFORM statement, *identifier-2* is augmented before *identifier-5* is set. In OPM COBOL/400, *identifier-5* is set before *identifier-2* is augmented.

The results of the Format 4 PERFORM statement with the AFTER phrase is different in ILE COBOL compared to OPM COBOL/400. Consider the following example:

```

PERFORM PARAGRAPH-NAME-1
  VARYING X FROM 1 BY 1 UNTIL X > 3
  AFTER Y FROM X BY 1 UNTIL Y > 3.

```

In OPM COBOL/400, *PARAGRAPH-NAME-1* is performed with (X,Y) values of (1,1), (1,2), (1,3), (2,1), (2,2), (2,3), (3,2), (3,3).

In ILE COBOL, *PARAGRAPH-NAME-1* is performed with (X,Y) values of (1,1), (1,2), (1,3), (2,2), (2,3), (3,3).

READ Statement

AT END Not Allowed for Random Reads of Relative Files: In ILE COBOL, the AT END phrase is not allowed for random reads of relative files since the meaning of the random read would be indeterminate under these circumstances. The ILE COBOL compiler issues a severity 30 error message in this situation.

The OPM COBOL/400 compiler does not issue any error messages in this situation.

Error Messages: For ILE COBOL, error message LNC1408, not LNC0651, is issued for the FORMAT phrase when a READ statement is to be performed on a FORMATFILE.

Error message LNC1408 is issued when the device to be read is something other than DATABASE. Error message LNC0651 is issued when the device is DATABASE, but ORGANIZATION is not indexed.

REWRITE Statement

In OPM COBOL/400, file status is set to 90 when a record format that is not valid for a file is used on the REWRITE statement.

In ILE COBOL, file status is set to 9K when a record format that is not valid for a file is used on the REWRITE statement.

SET Statement

When setting a condition-name to TRUE and the associated condition variable is an edited item, OPM COBOL/400 edits the value of the condition-name when it is moved to the condition variable.

ILE COBOL does not perform any editing when the value of the condition-name is moved to the condition variable.

SORT/MERGE Statements

GIVING Phrase and the SAME AREA/SAME RECORD AREA Clauses: In ILE COBOL, file-names associated with the GIVING phrase may not be specified in the same SAME AREA or SAME RECORD AREA clauses. The ILE COBOL compiler issues a severity 30 error message if this situation is encountered.

The OPM COBOL/400 compiler does not issue any messages in this situation.

STOP RUN Statement

When STOP RUN is issued in an ILE activation group, it will cause an implicit COMMIT to take place, which is not the case in OPM COBOL/400.

Note: A STOP RUN issued in the job default activation group (*DFACTGRP) will not cause an implicit COMMIT.

STRING/UNSTRING Statements

In OPM COBOL/400, the PROGRAM COLLATING SEQUENCE is used to determine the truth value of the implicit relational conditions in STRING/UNSTRING operations.

In ILE COBOL, the PROGRAM COLLATING SEQUENCE is ignored when determining the truth value of the implicit relational conditions in STRING/UNSTRING operations.

Application Programming Interfaces (APIs)

ILE COBOL Bindable APIs

ILE COBOL uses new bindable APIs instead of the OPM runtime routines:

- QlnRtvCobolErrorHandler ILE bindable API replaces QLRRTVCE
- QlnSetCobolErrorHandler ILE bindable API replaces QLRSETCE
- QlnDumpCobol ILE bindable API replaces QLREXHAN to produce a formatted dump
- QLRCHGCM is not supported in ILE COBOL. Use named ILE activation groups to obtain multiple run units.

Calling OPM COBOL/400 APIs

OPM COBOL/400 APIs can be called from ILE COBOL but they will only affect OPM COBOL/400 run units.

To affect ILE COBOL run units, use the corresponding ILE APIs or the ACTGRP parameter of the CRTPGM command.

Run Time

Preserving the OPM-compatible Run Unit Semantics

You can closely preserve OPM-compatible run unit semantics in:

- An application that consists of only ILE COBOL programs, or
- An application that mixes OPM COBOL/400 programs and ILE COBOL programs.

Preserving OPM-compatible Run Unit Semantics in an ILE COBOL

Application: To preserve the OPM-compatible run unit semantics in an ILE COBOL application, the following conditions must be met:

- All run unit participants (ILE COBOL or other ILE programs/procedures) must run in a single ILE activation group.

Note: By using a named ILE activation group for all participating programs, you need not specify a particular ILE COBOL program to be the main program before execution. On the other hand, if a particular ILE COBOL program is known to be main program before execution, you can specify *NEW attribute for the ACTGRP option when creating a *PGM object using the ILE COBOL program as the UEP. All other participating programs should specify the *CALLER attribute for the ACTGRP option.

- The oldest invocation of the ILE activation group must be that of ILE COBOL. This is the main program of the run unit.

If these conditions are not met, an implicit or explicit STOP RUN in an ILE activation group may not end the activation group. With the activation group still active, the various ILE COBOL programs will be in their last used state.

Note: The above condition dictates that an ILE COBOL program running in the *DFTACTGRP is generally run in a run unit that is not OPM-compatible. ILE COBOL programs running in the *DFTACTGRP will not have their static storage physically reclaimed until the job ends. An ILE COBOL program, with *CALLER specified for the ACTGRP parameter of the CRTPGM command, will run in the *DFTACTGRP if it is called by an OPM program.

Preserving OPM-compatible Run Unit Semantics in a Mixed OPM COBOL/400 and ILE COBOL Application: In order to mix OPM COBOL/400 programs with ILE COBOL programs and still preserve the OPM-compatible run unit semantics as closely as possible, the following conditions need to be met:

- OPM COBOL/400 program's invocation (not ILE COBOL's) must be the first COBOL invocation
- STOP RUN is issued by an OPM COBOL/400 program
- All participating programs in the (OPM COBOL/400) run unit must run in the *DFTACTGRP activation group.

If the above conditions are not met, the OPM-compatible run unit semantics is not preserved for OPM/ILE mixed application. For example, if an ILE COBOL program is running in the *DFTACTGRP and it issues a STOP RUN, both the OPM COBOL/400 and ILE COBOL programs will be left in their last used state.

In ILE COBOL, the flow of control operations, CALL, CANCEL, EXIT PROGRAM, STOP RUN, and GOBACK, will cause the run unit to behave differently unless an OPM-compatible run unit is used.

Error Messages

In ILE COBOL, runtime error messages are prefixed with LNR. Also, some of the message numbers are not always the same as in OPM COBOL/400.

In ILE COBOL, when the run unit terminates abnormally, the message CEE9901 is returned to the caller. In OPM COBOL/400, the message LBE9001 is returned to the caller under the same circumstances.

Due to differences between ILE exception handling and OPM exception handling, you may receive more exceptions in an ILE COBOL statement compared to an OPM COBOL/400 statement.

File Status 9A changed to 0A

In OPM COBOL/400, file status is set to 9A when a job is ended in a controlled manner.

In ILE COBOL, file status is set to 0A when a job is ended in a controlled manner.

File Status 9M changed to 0M

In OPM COBOL/400, file status is set to 9M when the last record is written to a subfile.

In ILE COBOL, file status is set to 0M when the last record is written to a subfile.

Appendix H. Glossary of Abbreviations

Abbreviation	Meaning	Explanation
AG	Activation Group	A partitioning of resources within a job. An activation group consists of system resources (storage for program or procedure variables, commitment definitions, and open files) allocated to one or more programs.
API	Application Programming Interface	A functional interface supplied by the operating system or by a separately orderable licensed program that allows an application program written in a high-level language to use specific data or functions of the operating system or licensed program.
ANSI	American National Standards Institute	An organization consisting of producers, consumers, and general interest groups, that establishes the procedures by which accredited organizations create and maintain voluntary industry standards in the United States.
ASCII	American National Standard Code for Information Interchange	The code developed by American National Standards Institute for information exchange among data processing systems, data communications systems, and associated equipment. The ASCII character set consists of 8-bit characters, consisting of 7-bit control characters and symbolic characters, plus one parity-check bit.
CICS [®]	Customer Information Control Service	An IBM licensed program that enables transactions entered at remote work stations to be processed concurrently by user-written application programs. The licensed program includes functions for building, using, and maintaining databases, and for communicating with CICS on other operating systems.
CL	Control Language	The set of all commands with which a user requests system functions.
DBCS	Double-Byte Character Set	A set of characters in which each character is represented by 2 bytes. Languages such as Japanese, Chinese, and Korean, which contain more symbols than can be represented by 256 code points, require double-byte character sets. Because each character requires 2 bytes, the typing, displaying, and printing of DBCS characters requires hardware and programs that support DBCS. Four double-byte character sets are supported by the system: Japanese, Korean, Simplified Chinese, and Traditional Chinese. Contrast with single-byte character set.
DDM	Distributed Data Management	A function of the operating system that allows an application program or user on one system to use data files stored on remote systems. The systems must be connected by a communications network, and the remote systems must also be using DDM.
DDS	Data Description Specifications	A description of the user's database or device files that is entered into the system in a fixed form. The description is then used to create files.
EBCDIC	Extended Binary-Coded Decimal Interchange Code.	A coded character set consisting of 256 eight-bit characters.

Abbreviation	Meaning	Explanation
EPM	Extended Program Model	The set of functions for compiling source code and creating programs on the i5/OS in high-level languages that define procedure calls.
FIPS	Federal Information Processing Standard	An official standard to improve the utilization and management of computers and data processing in business.
HLL	high-level language	A programming language whose concepts and structures are convenient for human reasoning; for example, C, COBOL, and RPG. High-level languages are independent of the structures of computers and operating structures.
IBM i	IBM i	The i5 Operating System. Formerly OS/400.
ICF	Intersystem Communications Function	A function of the operating system that allows a program to communicate interactively with another program or system.
ILE	Integrated Language Environment	A set of constructs and interfaces that provides a common runtime environment and runtime bindable application programming interfaces (APIs) for all ILE-conforming high-level languages.
I/O	Input/Output	Data provided to the computer or data resulting from computer processing.
LVLCHK	Level Checking	A function that compares the record format-level identifiers of a file to be opened with the file description that is part of a compiled program to determine if the record format for the file changed since the program was compiled.
ODP	open data path	A control block created when a file is opened. An ODP contains information about the merged file attributes and information returned by input and output operations. The ODP only exists while the file is open.
ODT	Object Definition Table	A table built at compile time by the system to keep track of objects declared in the program. The program objects in the table include variables, constants, labels, operand lists and exception descriptions. The table resides in the compiled program object.
OPM	original program model	The set of functions for compiling source code and creating high-level language programs on the i5/OS before the Integrated Language Environment (ILE) model was introduced.
PEP	program entry procedure	A procedure provided by the compiler that is the entry point for an ILE program on a dynamic program call. Contrast with <i>user entry procedure</i> .
SDA	Screen Design Aid	A function of the Application Development ToolSet licensed program that helps the user design, create, and maintain displays and menus.
SEU	Source Entry Utility	A function of the Application Development ToolSet licensed program that is used to create and change source members.
SQL/400	Structured Query Language/400	An IBM licensed program supporting the relational database that is used to put information into a database and to get and organize selected information from a database.

Appendix I. ILE COBOL Documentation

The appendix describes the online information and hardcopy books available with ILE COBOL. For more information about ordering books, contact your IBM Representative.

Online Information

You can access the online information in several ways:

- Pressing F1 from a Create Bound COBOL Program (CRTBNDCBL) or Create COBOL Module (CRTCBLMOD) display
- Pressing F1 from within the Compiler Options or Program Verifier dialog boxes, or pressing F1 from within the Editor of CoOperative Development Environment/400
- Going to the i5/OS Infocenter: <http://www.ibm.com/eserver/iseriess/infocenter>
Select your geography, then the language and the latest release. In the left pane, select **System i supplemental manuals**. Then click **Complete list of manuals**. From the list of online books displayed, you can access the *IBM Rational Development Studio for i: ILE COBOL Programmer's Guide* and the *IBM Rational Development Studio for i: ILE COBOL Reference*.

#

Hardcopy Information

The following hardcopy books are available for the ILE COBOL product:

- *IBM Rational Development Studio for i: ILE COBOL Programmer's Guide*, SC09-2540-07
- *IBM Rational Development Studio for i: ILE COBOL Reference*, SC09-2539-07
- *ILE Concepts*, SC41-5606-09.

Note: Additional copies of any of the publications can be ordered for a fee.

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan, Ltd.
3-2-12, Roppongi, Minato-ku, Tokyo 106-8711

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Notices

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

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

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

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, IBM License Agreement for Machine Code, or any equivalent agreement between us.

This information 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.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Programming Interface Information

This publication is intended to help you write Integrated Language Environment[®] ILE COBOL programs. It contains information necessary for you to use the ILE COBOL compiler.

This manual does not document programming interfaces for use in writing programs that request or receive the services of the ILE COBOL compiler.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at Copyright and trademark information at www.ibm.com/legal/copytrade.shtml.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Java and all Java-based trademarks and logos are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

Acknowledgments

IBM acknowledges the use of the following research product in the ILE COBOL compiler:

S/SL ©Copyright 1981 by the University of Toronto

Notices

Bibliography

- # For additional information about topics related to
ILE COBOL programming on the i5/OS system,
refer to the following IBM i5/OS publications:
- # • *ADTS/400: Programming Development Manager*,
SC09-1771-00, provides information about using
the Application Development ToolSet
programming development manager (PDM) to
work with lists of libraries, objects, members,
and user-defined options to easily do such
operations as copy, delete, and rename.
Contains activities and reference material to
help the user learn PDM. The most commonly
used operations and function keys are
explained in detail using examples.
 - # • *ADTS for AS/400: Source Entry Utility*,
SC09-2605-00, provides information about using
the Application Development ToolSet source
entry utility (SEU) to create and edit source
members. The manual explains how to start
and end an SEU session and how to use the
many features of this full-screen text editor. The
manual contains examples to help both new
and experienced users accomplish various
editing tasks, from the simplest line commands
to using pre-defined prompts for high-level
languages and data formats.
 - # • *Application Display Programming*, SC41-5715-02,
provides information about:
 - # – Using DDS to create and maintain displays
for applications
 - # – Creating and working with display files on
the system
 - # – Creating online help information
 - # – Using UIM to define panels and dialogs for
an application
 - # – Using panel groups, records, or documents
 - # • *Recovering your system*, SC41-5304-10, provides
information about setting up and managing the
following:
 - # – Journaling, access path protection, and
commitment control
 - # – User auxiliary storage pools (ASPs)
 - # – Disk protection (device parity, mirrored, and
checksum)
 - # Provides performance information about
backup media and save/restore operations.
Also includes advanced backup and recovery
topics, such as using save-while-active support,
saving and restoring to a different release, and
programming tips and techniques.
 - # • *CICS for iSeries Application Programming Guide*,
SC41-5454-02, provides information on
application programming for CICS® for iSeries.
It includes guidance and reference information
on the CICS application programming interface
and system programming interface commands,
and gives general information about
developing new applications and migrating
existing applications from other CICS
platforms.
 - # • *CL Programming*, SC41-5721-06 provides a
wide-ranging discussion of iSeries
programming topics including a general
discussion on objects and libraries, CL
programming, controlling flow and
communicating between programs, working
with objects in CL programs, and creating CL
programs. Other topics include predefined and
impromptu messages and message handling,
defining and creating user-defined commands
and menus, application testing, including
debug mode, breakpoints, traces, and display
functions.
 - # • *Communications Management*, SC41-5406-02,
provides information about work management
in a communications environment,
communications status, tracing and diagnosing
communications problems, error handling and
recovery, performance, and specific line speed
and subsystem storage information.
 - # • *Experience RPG IV Tutorial*, GK2T-9882-00, 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
compiler code examples are shipped with the
tutorial and run directly on the iSeries.
 - # • *GDDM Programming Guide*, SC41-0536-00,
provides information about using IBM i
graphical data display manager (GDDM) to
write graphics application programs. Includes
many example programs and information to
help users understand how the product fits into
data processing systems.
 - # • *GDDM Reference*, SC41-3718-00, provides
information about using IBM i graphical data

display manager (GDDM) to write graphics application programs. This manual provides detailed descriptions of all graphics routines available in GDDM. Also provides information about high-level language interfaces to GDDM.

- # • *ICF Programming*, SC41-5442-00, provides information needed to write application programs that use iSeries communications and the IBM i intersystem communications function (IBM i-ICF). Also contains information on data description specifications (DDS) keywords, system-supplied formats, return codes, file transfer support, and program examples.
- # • *IDDU Use*, SC41-5704-00, describes how to use the iSeries interactive data definition utility (IDDU) to describe data dictionaries, files, and records to the system. Includes:
 - # – An introduction to computer file and data definition concepts
 - # – An introduction to the use of IDDU to describe the data used in queries and documents
 - # – Representative tasks related to creating, maintaining, and using data dictionaries, files, record formats, and fields
 - # – Advanced information about using IDDU to work with files created on other systems and information about error recovery and problem prevention.
- # • *IBM Rational Development Studio for i: ILE C/C++ Programmer's Guide*, SC09-2712-07, provides information on how to develop applications using the ILE C compiler 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 the ILE C compiler.
- # • *IBM Rational Development Studio for i: ILE C/C++ Language Reference*, SC09-7852-02, describes the syntax, semantics, and IBM implementation of the C and C++ programming languages.
- # • *IBM Rational Development Studio for i: ILE COBOL Reference*, SC09-2539-07, provides a description of the ILE COBOL programming language. It provides information on the structure of the ILE COBOL programming language and the structure of an ILE COBOL source program. It also provides a description of all Identification Division paragraphs, Environment Division clauses, Data Division clauses, Procedure Division statements, and Compiler-Directing statements.
- # • *ILE Concepts*, SC41-5606-09, explains concepts and terminology pertaining to the Integrated Language Environment (ILE) architecture of the iSeries licensed program. Topics covered include creating modules, binding, running programs, debugging programs, and handling exceptions.
- # • *IBM Rational Development Studio for i: ILE RPG Programmer's Guide*, SC09-2507-08, provides information about the ILE RPG compiler programming language, which is an implementation of the RPG IV language in the Integrated Language Environment (ILE) on the iSeries. It includes information on creating and running programs, with considerations for procedure calls and interlanguage programming. The guide also covers debugging and exception handling and explains how to use iSeries files and devices in RPG programs. Appendixes include information on migration to RPG IV and sample compiler listings. It is intended for people with a basic understanding of data processing concepts and of the RPG language.
- # • *IBM Rational Development Studio for i: ILE RPG Reference*, SC09-2508-08, provides information about the ILE RPG compiler 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.
- # • *Local Device Configuration*, SC41-5121-00, provides information about configuring local devices on the iSeries server. This includes information on how to configure the following:
 - # – Local work station controllers (including twinaxial controllers)
 - # – Tape controllers
 - # – Locally attached devices (including twinaxial devices)
- # • *Printer Device Programming*, SC41-5713-06, provides information to help you understand and control printing. Provides specific information on printing elements and concepts of the iSeries server, printer file and print spooling support for printing operations, and printer connectivity. Includes considerations for

using personal computers, other printing
functions such as IBM Business Graphics Utility
(IBGU), advanced function printing (AFP™),
and examples of working with the iSeries
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 iSeries are also
provided. Font substitution tables provide a
cross-reference of substituted fonts if attached
printers do not support application-specified
fonts.

- # • *Security reference, SC41-5302-11*, tells how
system security support can be used to protect
the system and the data from being used by
people who do not have the proper
authorization, protect the data from intentional
or unintentional damage or destruction, keep
security information up-to-date, and set up
security on the system.
- # • *Installing, upgrading, or deleting IBM i and related
software, SC41-5120-11*, provides step-by-step
procedures for initial installation, installing
licensed programs, program temporary fixes
(PTFs), and secondary languages from IBM.
This manual is also for users who already have
an iSeries server with an installed release and
want to install a new release.

For information about Systems Application
Architecture (SAA) Common Programming
Interface (CPI) COBOL, refer to the following
publication:

- *Systems Application Architecture Common
Programming Interface COBOL Reference*,
SC26-4354.

Index

Special characters

/ (slash) 15, 62
* (asterisk) 15
*ACCUPDALL option 40
*ACCUPDNE option 41
*ALL option 38, 41
*APOST option 31
*BASIC option 39
*BLANK option 30
*BLK option 33
*CBL statement 63
*CHANGE option 41
*CHGPOSSGN option 34
*CONTROL statement 62
*CRTARKIDX option 35
*CRTDTA value 101
*CRTF option 33
*CURLIB option 29, 44, 86
*CURRENT option 42, 48
*DATETIME option 35, 440, 443
*DBGDTA value 101
*DDSFILLER option 34
*DFRWRT option 40
*DFTACTGRP (Default Activation Group) 212, 226, 330
*DUPKEYCHK option 33
*EVENTF option 34
*EXCLUDE option 42
*FULL option 39
*GEN option 31
*HEX option 43
*HIGH option 40
*IMBEDERR option 34, 69
*INHERIT option 45, 88
*INTERMEDIATE option 40
*INZDLT option 33, 477
*JOB option 43, 44
*JOB RUN option 43, 44
*LANGIDSHR option 43
*LANGIDUNQ option 43
*LIBCRTAUT option 41
*LIBL option 29, 43
*LINENUMBER option 31
*LINKLIT option 42
*LIST option 38
*MAP option 31, 62
*MINIMUM option 40
*MODULE option 29
*MODULE, system object type 21
*MONOPIC option 35
*MONOPRC option 32
*NEVER option 39
*NO option 41, 86, 87
*NOBLK option 33
*NOCHGPOSSGN option 34
*NOCRTARKIDX option 35
 *NOCRTARKIDX 35
*NOCRTF option 33
*NODATETIME option 35
*NODDSFILLER option 34
*NODFRWRT option 40

*NODUPKEYCHK option 33
*NOFIPS option 40
*NOGEN option 31
*NOIMBEDERR option 34
*NOINZDLT option 33
*NOMAP option 31
*NOMAX option 38
*NOMONOPRC option 32
*NONE option 30, 39
*NONUMBER option 31
*NOOBSOLTE option 40
*NOOPTIONS option 31
*NOPICGGRAPHIC option 36
*NOPICNGGRAPHIC option 36
*NOPICXGRAPHIC option 35
*NOPRTCORR option 32
*NORANGE option 32
*NOSECLVL option 32
*NOSEQUENCE option 31
*NOSOURCE option 30
*NOSRC option 30
*NOSTDINZ option 34
*NOSTDTRUNC option 34
*NOSYNC option 33
*NOUNDSPCHR option 40
*NOUNREF option 32
*NOVARCHAR option 35
*NOVBSUM option 31
*NOXREF option 31
*NUMBER option 31
*OBSOLETE option 40
*OPTIONS option 31, 62
*OWNER option 87
*PGM option 42
*PGM, system object type 77
*PGMID option 28, 86
*PICGGRAPHIC option 36
*PICNGGRAPHIC option 36
*PICXGRAPHIC option 36
*PRC option 42
*PRINT option 30
*PRTCORR option 32
*QUOTE option 31
*RANGE option 32
*SECLVL option 32
*SEQUENCE option 31
*SIMPLEPGM option 87
*SNGLVL option 45, 88
*SOURCE option 30, 38, 62
*SRC option 30
*SRCMBRTXT option 30
*SRVPGM, system object type 105
*STDINZ option 34
*STDINZHEX00 option 34
*STDTRUNC option 34
*STGMDL option 88
*STMT option 38
*SYNC option 33
*TERASPACE option 45, 88
*UNDSPCHR option 40
*UNREF option 33

*USE option 41
*USER option 87
*VARCHAR option 35
*VBSUM option 31, 62
*XREF option 31, 62
*YES option 41, 86, 87

Numerics

0 option 41
30 option 30, 38

A

abnormal program termination 114
about this manual xi
ACCEPT statement 416, 623
access mode 475, 477
 DYNAMIC 485
 RANDOM 485
access path
 description 405
 example for indexed files 486
 file processing 473
 specifications 398
ACQUIRE statement 526, 566
activation 211
activation group (AG) 211, 665
activation group level scoping 412, 422
ADDMSGD (Add Message Description)
 command 612
ADDRESS OF special register 234, 340
 description 340
 difference from calculated ADDRESS
 OF 340
addresses
 incrementing using pointers 358
 passing between programs 356
ADTS
 messages 613
ADVANCING PAGE phrase 460
ADVANCING phrase 460
 for FORMATFILEs 459
AG (activation group) 211, 665
ALCOBJ (Allocate Object) command 413
ALIAS keyword 403
alias, definition 403
Allocate Object (ALCOBJ) command 413
American National Standard Code for
 Information Interchange (ASCII) 665
American National Standards Institute
 (ANSI) xxvi, 605, 609
 COBOL run unit 212
 conforming to standards
 with indexed files 478
 with relative files 475
 with sequential files 474
 definition 665
 FIPS specifications 609
 standard xxvi, 609

ANSI (American National Standards Institute)
 See American National Standards Institute (ANSI)

API (Application Programming Interface) 665
 error-handling 116, 371
 using with pointers 343

Application Development ToolSet messages 613

Application Programming Interface (API) 665
 error-handling 116, 371
 using with pointers 343

argc/argv 315

arguments, describing in the calling program 235

arithmetic operations, handling errors 374

arithmetic operators xxviii

arrival sequence 405, 474, 477, 479

arrows, shown in syntax xxix

ASCII (American National Standard Code for Information Interchange) 665

ASSIGN clause
 and DBCS characters 621
 description 409, 458, 466, 469, 524
 device name 409

assignment name 409, 536, 537, 621

AT END condition 378

ATTR debug command 118

ATTRIBUTE-CHARACTER XML event 282

ATTRIBUTE-CHARACTERS XML event 282

ATTRIBUTE-NAME XML event 282

ATTRIBUTE-NATIONAL-CHARACTER XML event 282

attributes
 of data items 71
 of files 71
 of table items 72

ATTRIBUTES field 71

AUT parameter 41

authorization-list-name option 42

B

batch compiles 60

batch jobs, representation of DBCS data in 628

bibliography 675

binder information listing 94

binder language 106

binder listing 90

binding
 binding process 77
 definition 77
 example 90

binding statistics listing 96

blank lines 62

block, description 415

blocking code, generation of 417

blocking output records 415

Boolean data types 31, 536

Boolean literal 31

BOTTOM debug command 119

boundary
 definition 418
 violation 477

bracketed-DBCS 617

BREAK debug command 118

breakpoints
 characteristics 129
 conditional breakpoints 132
 considerations for using 129
 description 129
 relational operators for conditional breakpoints 132
 removing all 134
 unconditional breakpoints 130
 use of 129

brief summary table listing 93

browsing a compiler listing 63
 See source entry utility (SEU)

BY CONTENT, definition 233

BY REFERENCE, definition 233

C

C

argc/argv 315

C function call
 running a COBOL program using 112

calling C programs 313

data type compatibility 315

external data 317

passing data to 315

recursion 314

returning control from 317

calculation operations; on fixed-length fields 439

call by identifier 223

CALL CL command
 passing parameters 111
 running a COBOL program 111

call level scoping 412

call stack 212

CALL statement
 BY CONTENT identifier 234
 BY CONTENT LENGTH OF identifier 234
 BY CONTENT literal 234
 BY CONTENT, implicit MOVE 342
 by identifier 223
 BY REFERENCE ADDRESS OF record-name 234
 BY REFERENCE identifier 234
 error handling 385
 passing data with operational descriptors 235
 passing OMITTED data 234
 recursive, description 214
 running a COBOL program using 112
 to QCMDEXC 332
 using pointers 342

called program
 definition 214

calling programs
 BY CONTENT 233
 BY REFERENCE 233
 calling EPM programs 331
 calling programs (*continued*)
 calling ILE C for AS/400 programs 313
 calling ILE CL programs 327
 calling ILE RPG for AS/400 programs 323
 calling OPM COBOL/400 programs 330
 calling OPM programs 329
 definition 214
 nested programs 217
 using pointers 342

calling the COBOL compiler 24

CANCEL statement 223, 247
 with COBOL programs 247
 with non-COBOL programs 248

canceling a COBOL program 247

CBLLE (default member type) 13

CCSID
 conflict 297
 of PARSE statement 297
 of XML document 293, 297

CCSID (Coded Character Set Identifier)
 See Coded Character Set Identifier (CCSID)

CDRA (Character Data Representation Architecture) 17

CEE9901 escape message 371

CEEHDLR bindable API 371

century problem 186

Change Debug (CHGDBG)
 command 117, 122

change/date (CHGDATE) field 69

Character Data Representation Architecture (CDRA) 17

character set identifiers 16

characters, double-byte 617

checking DBCS literals 619

checking work station validity 521

checking, data 167

CHGDBG (Change Debug)
 command 117, 122

CL (Control Language)
 calling CL programs 327
 data type compatibility 328
 definition 665
 passing data to 327
 returning control from 329

CL (control language) commands
 for running programs 111
 for testing programs 117
 issuing using QCMDEXC in a program 332

CL (control language) entry codes xxx
 clauses
 ACCESS MODE 525
 ASSIGN 458, 466, 469, 524, 621
 CONTROL-AREA 525
 CURRENCY 15
 DECIMAL-POINT 15
 FILE STATUS 416
 INDICATOR 548
 JUSTIFIED 622
 LINAGE 459
 OCCURS 622
 ORGANIZATION 458, 466, 470, 525
 ORGANIZATION IS INDEXED 477

- clauses (*continued*)
 - PICTURE 622
 - RECORD KEY 406
 - REDEFINES 621
 - REPLACING identifier-1 BY identifier-2 clause 15
 - syntax, notation for xxviii
 - VALUE 622
- CLEAR debug command 118
- Client tools xxxi
 - Remote System Explorer xxxi
 - Remote Systems LPEX Editor xliii
 - Remote Systems view xxxvi
 - System i Table view xxxvii
- CLOSE statement 461, 468, 471
- closing files with the CANCEL statement 247
- COBOL (Common Business Oriented Language), description 3
- COBOL procedure 6
- code page identifiers 16
- Coded Character Set Identifier (CCSID)
 - assigning a CCSID 17
 - CCSID 65535 17
 - COBOL syntax checker and CCSIDs 18
 - copy member with different CCSIDs 17
 - default 17
 - definition 16
- coding form 12
- coding formats provided by SEU 12
- collating sequence, specifying 49
- command definition 114
- command option summary listing 91
- command summary listing 63
- command syntax, using xxviii
- commands
 - Add Message Description (ADDMSGD) 612
 - Allocate Object (ALCOBJ) 413
 - Change Debug (CHGDBG) 117
 - Create Diskette File (CRTDKTF) 469
 - Create Logical File (CRTLFL) 473
 - Create Physical File (CRTPFL) 473
 - Create Print File (CRTPRFL) 457
 - Create Tape File (CRTTAPF) 465
 - Monitor Message (MONMSG) 24
 - Override Message File (OVRMSGF) 612
 - Override to Diskette File (OVRDKTF) 410
 - Reorganize Physical File Member (RGZPFM) 477
 - Start Debug (STRDBG) 117
 - Start Source Entry Utility (STRSEU)
 - See source entry utility
- comment line 62
- COMMENT XML event 281
- comments with DBCS characters 620
- COMMIT statement 418, 420
- commitment boundary, definition 418
- commitment control
 - definition 387, 417
 - example 422
 - locking level 418
 - scope 421
- commitment definition 421
- COmmon Business Oriented Language (COBOL), description 3
- common keys 405
- Common Programming Interface (CPI) support 607
- communication module 606, 607
- communications, interactive
 - interprogram considerations 211, 629
 - recovery 388
 - with other programs 521
 - with work systems 521
 - with workstation users 521
- compilation unit 7, 12
- compile listing, viewing 48
- compiler failure 24
- compiler options
 - *ACCUPDALL 40
 - *ACCUPDNE 41
 - *ALL 38, 41
 - *APOST 31
 - *BASIC 39
 - *BLANK 30
 - *BLK 33
 - *CHANGE 41
 - *CHGPOSSGN 34
 - *CRTF 33
 - *CURLIB 29, 44, 86
 - *CURRENT 42, 48
 - *DATETIME 35, 440, 443
 - *DDSFILLER 34
 - *DFRWRT 40
 - *DUPKEYCHK 33
 - *EVENTF 34
 - *EXCLUDE 42
 - *EXTEND31 46
 - *FULL 39
 - *GEN 31
 - *HEX 43
 - *HIGH 40
 - *IMBEDERR 34
 - *INHERIT 45, 88
 - *INTERMEDIATE 40
 - *INZDLT 33
 - *JOB 43, 44
 - *JOBRUN 43, 44
 - *LANGIDSHR 43
 - *LANGIDUNQ 43
 - *LIBCRTAUT 41
 - *LIBL 29, 43
 - *LINENUMBER 31
 - *LIST 38
 - *MAP 31, 62
 - *MINIMUM 40
 - *MODULE 29
 - *MONOPIC 35
 - *MONOPRC 32
 - *NEVER 39
 - *NO 41, 86, 87
 - *NOBLK 33
 - *NOCHGPOSSGN 34
 - *NOCRTARKIDX 35
 - *NOCRTF 33
 - *NODATETIME 35
 - *NODDSFILLER 34
 - *NODFRWRT 40
 - *NODUPKEYCHK 33
- compiler options (*continued*)
 - *NOEVENTF 34
 - *NOEXTEND 46
 - *NOFIPS 40
 - *NOGEN 31
 - *NOIMBEDERR 34
 - *NOINZDLT 33
 - *NOMAP 31
 - *NOMAX 38
 - *NOMONOPIC 35
 - *NOMONOPRC 32
 - *NONE 30, 39
 - *NONUMBER 31
 - *NOOBSOLETE 40
 - *NOOPTIONS 31
 - *NOPICGGRAPHIC 36
 - *NOPICNGRAPHIC 36
 - *NOPICXGRAPHIC 35
 - *NOPRTCORR 32
 - *NORANGE 32
 - *NOSECLVL 32
 - *NOSEQUENCE 31
 - *NOSOURCE 30
 - *NOSRC 30
 - *NOSTDINZ 34
 - *NOSTDTRUNC 34
 - *NOSYNC 33
 - *NOUNDSPCHR 40
 - *NOUNREF 32
 - *NOVARCHAR 35
 - *NOVBSUM 31
 - *NOXREF 31
 - *NUMBER 31
 - *OBSOLETE 40
 - *OPTIONS 31, 62
 - *OWNER 87
 - *PGM 42
 - *PGMID 28, 86
 - *PICGGRAPHIC 36
 - *PICNGRAPHIC 36
 - *PICXGRAPHIC 36
 - *PRC 42
 - *PRINT 30
 - *PRTCORR 32
 - *QUOTE 31
 - *RANGE 32
 - *SECLVL 32
 - *SEQUENCE 31
 - *SINGLVL 45, 88
 - *SOURCE 30, 38, 62
 - *SRC 30
 - *SRMBRTXT 30
 - *STDINZ 34
 - *STDINZHEX00 34
 - *STDTRUNC 34
 - *STGMDL 88
 - *STMT 38
 - *SYNC 33
 - *TERASPACE 45, 88
 - *UNDSPCHR 40
 - *UNREF 33
 - *USE 41
 - *USER 87
 - *VARCHAR 35
 - *VBSUM 31, 62
 - *XREF 31, 62
 - *YES 41, 86, 87

- compiler options (*continued*)
 - See also* PROCESS statement; also parameters, CRTCBMOD command
 - 30 option 30, 38, 41
 - and syntax checking with SEU 15
 - ARITHMETIC parameter 46
 - as specified in PROCESS statement 51
 - authorization-list-name option 42
 - batch compiling 60
 - CCSID parameter 45
 - compiler options listing 61, 65
 - create cross-reference listing 73
 - create source listing 66
 - DATTIM option 59
 - 2-digit base year 59
 - 4-digit base century 59
 - ENBPFCOL parameter 44
 - error-severity-level option 38
 - GRAPHIC option 59
 - language-identifier-name option 44
 - library-name option 29, 44, 86
 - LICOPT parameter 46
 - list compiler options in effect 62, 66
 - maximum-number option 38
 - module-name option 28
 - NOGRAPHIC option 59
 - NTPADCHAR parameter 46
 - optimizing source code 39
 - parameters of the
 - CRTCBMOD/CRTBNDCBL commands 28, 49, 59
 - PRFDATA parameter 45
 - PROCESS statement, using to specify 51
 - program listings, DBCS characters in 630
 - program-name option 86
 - QCBLESRC (default source file) 29
 - release-level option 48
 - severity-level option 30, 41
 - source-file-member-name option 29
 - source-file-name option 29
 - STGMDL parameter 45
 - suppressing source listing 66
 - table-name option 43
 - text-description 30
 - THREAD option 59
 - multithreading 361
 - NOTHREAD 59
 - SERIALIZE 59
- compiler options listing 65
- compiler output
 - See also* messages
 - browsing 63
 - CCSID parameter 45
 - command summary listing 63
 - compiler output 60, 61
 - cross-reference listing 73
 - CRTCBMOD/CRTBNDCBL options 62
 - Data Division map 69
 - description 61
 - ENBPFCOL parameter 44
 - examples 61
 - FIPS messages listing 72
 - listing descriptions 61
- compiler output (*continued*)
 - listing options 65
 - messages 612
 - options listing 63, 65
 - PRFDATA parameter 45
 - program listings, DBCS characters in 630
 - STGMDL parameter 45
 - suppressing source listing 66
- compiling COBOL programs
 - abnormal compiler termination 24
 - example listing 63
 - example of 47
 - failed attempts 24
 - for the previous release 48
 - invoking the compiler 21
 - messages 612
 - multiple programs 60
 - output 61
 - redirecting files 410
 - TGTRLS, using 48
- Configuration Section, description 6, 621
- conforming to ANSI standards 609
- constant, NULL figurative 338
- CONTENT-CHARACTER XML event 283
- CONTENT-CHARACTERS XML event 283
- CONTENT-NATIONAL-CHARACTER XML event 284
- contiguous items, definition 480
- contiguous key fields, multiple 480
- control
 - returning 225
 - transferring 214
- control boundary 213
- Control Language (CL)
 - See also* CL (Control Language)
 - calling CL programs 327
 - data type compatibility 328
 - definition 665
 - passing data to 327
 - returning control from 329
- control language (CL) entry codes xxx
- Control Language commands
 - See* CL commands
- CONTROL-AREA clause 525
- control, returning from a called program 225
- control, transferring to another program 214
- conversion, data format 165
- CoOperative Development Environment/400 (the client product) 9
- copies of ANSI standard available xxvi
- COPY statement
 - and DBCS characters 628
 - DDS results 402
 - example of data structures generated by 534
 - format-1 COPY statement 61
 - key fields 479
 - listing source statements 63
 - suppressing source statements 63
 - use with PROCESS statement 61
 - use with TRANSACTION files 521
- COPYNAME field 69
- corresponding options, PROCESS and CRTCBMOD/CRTBNDCBL commands 52
- COUNT IN phrase
 - XML GENERATE 312
- counting
 - generated XML characters 302
- counting verbs in a source program 69, 76
- CPI (Common Programming Interface) support 607
- Create Bound COBOL (CRTBNDCBL) command
 - ARITHMETIC parameter 46
 - AUT parameter 41
 - CCSID parameter 45
 - compiling source statements 81, 90
 - CVTOPT parameter 35
 - DBGVIEW parameter 38
 - description of 78
 - ENBPFCOL parameter 44
 - EXTDSOPT parameter 40
 - FLAG parameter 41
 - FLAGSTD parameter 40
 - GENLVL parameter 30
 - invoking CRTPGM 88
 - LANGID parameter 44
 - LICOPT parameter 46
 - LINKLIT parameter 42
 - MSGLMT parameter 37
 - NTPADCHAR parameter 46
 - OPTIMIZE parameter 39
 - OPTION parameter 30, 62
 - OUTPUT parameter 29
 - PGM parameter 86
 - PRFDATA parameter 45
 - REPLACE parameter 86
 - SIMPLEPGM parameter 87
 - SRCFILE parameter 29
 - SRCMBR parameter 29
 - SRTSEQ parameter 43
 - STGMDL parameter 45, 88
 - syntax 82
 - TEXT parameter 30
 - TGTRLS parameter 42
 - using CRTBNDCBL 81
 - using prompt displays with 82
 - USRPRF parameter 86
- Create COBOL Module (CRTCBMOD) command
 - ARITHMETIC parameter 46
 - AUT parameter 41
 - CCSID parameter 45
 - compiling source statements 24, 47
 - CVTOPT parameter 35
 - DBGVIEW parameter 38
 - description of 21
 - ENBPFCOL parameter 44
 - EXTDSOPT parameter 40
 - FLAG parameter 41
 - FLAGSTD parameter 40
 - GENLVL parameter 30
 - LANGID parameter 44
 - LICOPT parameter 46
 - LINKLIT parameter 42
 - MODULE parameter 28

Create COBOL Module (CRTCBMOD)
 command (*continued*)
 MSGLMT parameter 37
 NTLPADCHAR parameter 46
 OPTIMIZE parameter 39
 OPTION parameter 30, 62
 OUTPUT parameter 29
 PRFDTA parameter 45
 REPLACE parameter 41
 SRCFILE parameter 29
 SRCMBR parameter 29
 SRTSEQ parameter 43
 STGMDL parameter 45
 syntax 25
 TEXT parameter 30
 TGTRLS parameter 42
 using CRTCBMOD 24
 using prompt displays with 24
 create data 101
 Create Diskette File (CRTDKTF)
 command 469
 Create library (CRTLIB) command 11, 15
 Create Logical File (CRTLFL)
 command 473
 Create Physical File (CRTPF)
 command 473
 Create Print File (CRTPRTF)
 command 457
 Create Program (CRTPGM) command
 description of 78
 invoking from CRTBNDCBL 88
 parameters 80
 using CRTPGM 79
 Create Service Program (CRTSRVPGM)
 command
 description of 106
 parameters 106
 using CRTSRVPGM 106
 Create Source Physical File (CRTSRCPF)
 command 11, 15
 Create Tape File (CRTTAPF)
 command 465
 creating a module object 47
 creating a program object 77
 creating a service program 105
 creating files
 indexed files 489, 500
 relative files 489, 493
 sequential files 489
 cross-reference listing
 description of listing 74
 example 73, 95
 CRTBNDCBL (Create Bound COBOL)
 command
See Create Bound COBOL
 (CRTBNDCBL) command
 CRTCBMOD (Create COBOL Module)
 command
See Create COBOL Module
 (CRTCBMOD) command
 CRTDKTF (Create Diskette File)
 command 469
 CRTLFL (Create Logical File)
 command 473
 CRTLIB (Create Library) command 11,
 15

CRTPF (Create Physical File)
 command 473
 CRTPGM (Create Program) command
See Create Program (CRTPGM)
 command
 CRTPRTF (Create Print File)
 command 457
 CRTSRCPF (Create Source Physical File)
 command 11, 15
 CRTSRVPGM (Create Service Program)
 command
See Create Service Program
 (CRTSRVPGM) command
 CRTTAPF (Create Tape File)
 command 465
 CVTOPT parameter 35

D

data
 EXTERNAL data 237
 global data 233
 local data 233
 OMITTED 234
 passing
 BY CONTENT and BY
 REFERENCE 234
 in groups 236
 to ILE C for AS/400
 programs 315
 to ILE CL programs 327
 to ILE RPG for AS/400
 programs 324
 with operational descriptors 235
 data area
 description 244
 local 244
 PIP 246
 data checking 167
 data class type (TYPE) field 70
 data communications file 521
 data description specifications (DDS)
 Create File commands 398
 date fields 442
 definition 521, 665
 description 398
 display management 521
 examples
 for a display device file 523
 for field reference file 400
 for subfile record format 551, 553
 formats, data structures generated
 by 534
 keyed access path for an indexed
 file 486
 specifications for a database
 file 402
 specifying a record format 401
 workstation programs 529, 601
 externally described files 397, 479
 FORMATFILE files 460
 function of 521
 graphic data fields 451
 incorporate description in
 program 401
 key fields 479
 multiple device files 555
 data description specifications (DDS)
 (*continued*)
 program-described files 397
 SAA fields 442
 subfiles 549
 time fields 442
 timestamp fields 442
 TRANSACTION files 521
 use of keywords 399
 variable-length fields 438
 Data Division
 arguments for calling program 235
 DBCS characters 621
 description 6
 map of, compiler option 69
 data dump 631
 data field 12
 data files, inline 411
 data format conversion 165
 data item
 attributes of 71
 defining as a pointer 336
 in subprogram linkage 236
 passing, with its length 234
 data type compatibility
 between C and COBOL 315
 between CL and COBOL 328
 between Java and COBOL 268
 between RPG and COBOL 325
 data types
 arithmetic, performing
 COMPUTE 168
 conversion of data, intrinsic
 functions 173
 expressions 169
 intrinsic functions, numeric 169
 introduction 168
 centry problem
 introduction 186
 solution, long-term 187
 solution, short-term 187
 class test, numeric 167
 computation data representation
 binary 157, 158
 external decimal 157
 external floating-point 159
 internal decimal 157
 internal floating-point 158
 USAGE clause and 156
 date 442
 defining numeric 155
 fixed-point, floating-point
 comparisons, arithmetic 184
 examples 185
 fixed-point 184
 floating-point 184
 introduction 183
 table items, processing 185
 format conversions 165
 graphic 451
 numeric editing 156
 portability and 156
 restrictions for SAA data types 442
 SAA data types 438
 sign representation 166
 time 442
 timestamp 442

- data types (*continued*)
 - year 200 problem
 - introduction 186
 - solution, long-term 187
 - solution, short-term 187
- DATABASE device 473
- database files
 - See also* disk files
 - DATABASE file considerations 473
 - DATABASE versus DISK 473
 - definition 473
 - DISK file considerations 473
 - processing methods 474
- date data type 442
- date-last-modified area 12
- DATTIM option 59
- DATTIM process statement option 51
- DB-FORMAT-NAME special register 478
- DBCS literal 617, 619, 628
- DBCS support
 - See* double-byte character set support
- DBCS-graphic data type 451, 617
- DBGVIEW parameter 38, 120
- DDM (distributed data management) 665
- DDS
 - See* data description specifications
- debug data 22, 101
 - watch condition 119
- debug session 120
 - watch condition 119
- debugging a program
 - adding programs to a debug session 125
 - breakpoints
 - See* breakpoints
 - changing the value of variables 148
 - debug commands 118
 - debug module 606, 607
 - debug session, preparing for 120
 - definition 117
 - displaying variables 143
 - file status 416
 - formatted dump 372
 - ILE COBOLCOLLATING SEQUENCE 118
 - ILE source debugger 118
 - national language support 150
 - protecting database files in production libraries 117
 - removing programs to a debug session 126
 - starting the ILE source debugger 122
 - stepping through a program 140
 - viewing program source 127
 - watch condition 119
- declarative procedures 380
- Default Activation Group
 - (*DFTACTGRP) 212, 226, 330
- default member type (CBLLE) 13
- default source file (QCBLLESRC) 13
- default values, indication of 25
- defined fields 74
- delays, reducing length of on initialization 477
- deleted records, initializing files with 477
- delimiting SQL statements 332
- descending file considerations 487
- descending key sequence, definition 487
- description and reference numbers
 - flagged field 72
- designing your program 4
- destination of compiler output 60
- device control information 523
- device dependence
 - examples 409
- device files
 - DATABASE file considerations 473
 - definition 457
 - DISK file considerations 473
 - DISKETTE device 469
 - FORMATFILE device 460
 - multiple 554, 555
 - PRINTER device 457
 - single 554
 - TAPE device 465
 - WORKSTATION device 524
- device independence 409
- device-dependent area, length of 417
- diagnostic levels 611
- diagnostic messages 74
- diagrams, syntax 25, 82
- direct files
 - See* relative files
- disclaimers
 - sending information to IBM ii
 - US government users ii
- DISK device 473
- disk files
 - processing methods 474
 - variable length records 487
- DISKETTE device 469
- diskette file
 - definition 469
 - describing 470
 - end of volume 470
 - naming 469
 - reading 470
 - writing 470
- displacement (DISP) field 70
- DISPLAY debug command 118
- display device
 - DDS for 521
 - record format 522, 523
- display device file 521
- display format data, definition 522
- DISPLAY statement 624
- DISPLAY-OF intrinsic function 175
- displaying a compiler listing 63
- displays
 - CRTBNDCBL prompt display 82
 - CRTCBLMOD prompt display 24
 - data description specifications (DDS) for 521
 - display program messages 613
 - for sample programs
 - order inquiry 582, 583
 - payment update 599, 600, 601
 - transaction inquiry 535
 - SEU display messages 613
 - subfiles 550
- distributed data management (DDM) 665
- division by zero 375
- divisions of programs
 - Data Division 621
 - Environment Division 621
 - Identification Division 6
 - optional 6
 - Procedure Division 623, 628
 - required 6
- do while structure, testing for end of chained list 357
- DOCUMENT-TYPE-DECLARATION
 - XML event 281
- documentary syntax xxx
- double spacing 62
- double-byte character set (DBCS) support
 - ACCEPT statement 623
 - and alphanumeric data 627
 - checking 619
 - comments with DBCS characters 620
 - communications between
 - programs 629
 - definition 665
 - description 617, 630
 - enabling in COBOL programs 618
 - graphic 629
 - in the Data Division 621
 - in the Environment Division 621
 - in the Identification Division 620
 - in the Procedure Division 623, 628
 - open 628
 - PROCESS statement 617, 625
 - representation of DBCS data in batch jobs 628
 - searching for in a table 628
 - sorting 628
 - specifying DBCS literals 618
- DOWN debug command 119
- DROP statement 529, 568
- dump, formatted 372, 631
- dynamic access mode 475, 479, 550
- dynamic file creation 33
- dynamic program call
 - description 215
 - performing 222
 - to a service program 108
 - using 223

E

- EBCDIC (Extended Binary-Coded Decimal Interchange Code) 665
- editing source programs
 - See* source entry utility (SEU)
- EJECT statement 62
- elementary pointer data items 340
- embedded SQL 332
- encoding
 - controlling in XML output 311
 - XML documents 293
- encoding scheme identifiers 16
- ENCODING-DECLARATION XML event 281
- End Commitment Control (ENDCMTCTL) command 421
- End Debug (ENDDBG) command 122
- end of chained list, testing for 357
- END PROGRAM 6

- END-OF-CDATA-SECTION XML
 - event 284
 - END-OF-DOCUMENT XML event 285
 - END-OF-ELEMENT XML event 284
 - end-of-file condition 378
 - END-OF-PAGE phrase 459
 - ENDCMTCTL (End Commitment Control) command 421
 - ENDDBG (End Debug) command 122
 - ending a called program 225
 - ending a COBOL program 114, 371
 - enhancing XML output
 - example of converting hyphens in element names to underscores 310
 - example of modifying data definitions 307
 - rationale and techniques 306
 - entering source members 6
 - entering source programs 6, 11, 12, 13
 - entering your program
 - See* source entry utility (SEU)
 - entry codes, control language xxx
 - Environment Division
 - and DBCS characters 621
 - description 6
 - EPM (extended program model) 3, 331
 - EPM (Extended Program Model) 666
 - EQUATE debug command 118
 - error handling
 - APIs 116, 371
 - in arithmetic operations 374
 - in input-output operations
 - end-of-file condition (AT END phrase) 378
 - EXCEPTION/ERROR declaratives (USE statement) 380
 - file status key 381
 - invalid key condition (INVALID KEY phrase) 379
 - overview 376
 - in sort/merge operations 385
 - in string operations 374
 - on the CALL statement 385
 - overview 369
 - Program Status Structure 373
 - user-written error handling routines 386
 - error recovery, example 387
 - error-severity-level option 38
 - errors
 - ADVANCING phrase with FORMATFILE files 459
 - errors, in syntax
 - See* syntax errors
 - escape message 371
 - EVAL debug command 118
 - examples
 - access path for indexed file 486
 - activation group
 - multiple, *NEW and named 229
 - multiple, *NEW, named, and *DFACTGP 231
 - single activation group 227
 - two named activation groups 228
 - binder information listing 94
 - binding multiple modules 81
 - binding one module 90
 - examples (*continued*)
 - binding statistics listing 96
 - brief summary table listing 93
 - COBOL and files 404
 - command option summary listing 91
 - commitment control 417, 423
 - compiler options listing 61
 - compiling a source program 47
 - COPY DDS results 402
 - COPY statement in PROCESS statement 61
 - cross-reference listing 73, 95
 - Data Division map 69
 - DDS
 - for a display device file 521, 523
 - for a record format 401
 - for a record format with ALIAS keyword 403
 - for field reference file 400
 - for multiple device files 555
 - for subfiles 551, 553
 - diagnostic messages listing 74
 - END-OF-PAGE condition 464
 - entering CRTCLMOD from command line 47
 - entering source statements 15
 - error recovery 387
 - extended summary table listing 92
 - EXTERNAL files 238
 - externally described printer files 461
 - file processing
 - indexed files 500, 502
 - relative files 493, 495
 - sequential files 489, 491
 - FIPS messages listing 72
 - FORMATFILE file 459
 - formatted dump 631
 - generic START 480, 482
 - indicators 537
 - LENGTH OF special register with pointers 339
 - length of variable-length field 439
 - MOVE with pointers 341
 - multiple device files 557
 - pointers
 - aligning 337
 - and LENGTH OF special register 339
 - and REDEFINES clause 338
 - and results of MOVE 341
 - initializing with NULL 338
 - processing chained list 355
 - program object, creating 81
 - program structure 4
 - record format specifications 400, 402
 - returning from a called program 227
 - service program, creating 107
 - SEU display messages 613
 - sorting/merging files 435
 - source listing 66
 - using pointers in chained list 355
 - variable-length graphic data 452
 - verb usage by count listing 69
 - workstation application programs
 - order inquiry 569
 - payment update 583
 - transaction inquiry 529
 - exception condition
 - XML GENERATE 312
 - exception handling
 - See* error handling
 - EXCEPTION XML event 285
 - exceptions 24, 114, 382, 386
 - exclusive-allow-read lock state 414
 - EXIT PROGRAM statement 226, 247, 371
 - export list 106
 - expressions 623
 - EXTDSPOPT parameter 40
 - EXTEND mode, definition 413
 - Extended Binary-Coded Decimal Interchange Code (EBCDIC) 665
 - extended dump 631
 - extended program model (EPM) 3, 331
 - Extended Program Model (EPM) 666
 - extended summary table listing 92
 - extensions, IBM
 - double-byte character set (DBCS) support 617, 630
 - flagging 609
 - transaction files 521, 601
 - EXTERNAL data
 - shared with a service program 108
 - shared with other programs 237
 - external description
 - adding functions to 404
 - overriding functions to 404
 - external file status 381
 - EXTERNAL files 238
 - externally attached devices 457
 - externally described files
 - adding functions 404
 - advantages of using for printer files 459
 - considerations for using 398
 - COPY statement 464
 - DDS for 401
 - description 397
 - example 401
 - level checking 406
 - overriding functions 404
 - printer files, specifying with FORMATFILE 460
 - specifying record retrieval 405
 - externally described TRANSACTION files 521, 524
 - EXTERNALLY-DESCRIBED-KEY 479
- ## F
- failed I/O and record locking 414
 - failure of compiler 24
 - FD (Sort Description) entries 428
 - Federal Information Processing Standard (FIPS)
 - 1986 COBOL standard 609
 - definition 666
 - description 609
 - flagging deviations from 609, 629
 - FLAGSTD parameter 72
 - messages 72, 609, 612
 - standard modules 609
 - standards to which the compiler adheres xxvi

Federal Information Processing Standard (FIPS) (*continued*)
 with DBCS characters 629

FIB (file information block) 381

fields

- date 442
- fixed length 439
- null-capable 443
- time 442
- time separator 63
- timestamp 442
- variable-length
 - character 438, 439
 - graphic 439, 452
 - length of, example 439
 - restrictions 439

figurative constant, NULL 338

file and record locking 413, 419

file boundaries 477

file considerations 477

file control entry 409

file descriptions 401

file information block (FIB) 381

file locking 413

file organization 473

file redirection 410, 413

file status

- 0Q 477
- 9N 388
- 9Q 477
- after I/O 381, 388
- coded examples 491
- error handling 381
- how it is set 381
- internal and external 381
- obtaining 416
- statements that affect 246

FILE STATUS clause 416

files

- See also* disk files, externally described files, program-described files, source files
- access paths 473
- attributes of 71
- creation of
 - indexed 489, 500
 - relative 489, 493
 - sequential 489
- DATABASE 473, 474
- DATABASE versus DISK 473
- description 489
- DISK 473, 474
- examples
 - EXTERNAL files 238
 - indexed files 500, 502
 - relative files 493, 495
 - sequential files 489, 491
- EXTERNAL 238
- external description 398
- FORMATFILE 460
- indexed organization 477
- keys 406
- logical 485
- on AS/400 systems 397, 489
- preserving sequence of records 475
- PRINTER 459
- processing methods 474

files (*continued*)

- redirecting access to 410
- relative 475
- relative organization 475
- retrieval of, relative 489, 497
- sample programs 489, 500
- sequential 474
- sequential organization 474
- techniques for processing 489, 500
- TRANSACTION 521

FIND debug command 119

FIPS violations flagged, total 73

FIPS-ID field 72

fixed length graphic fields 451

fixed-point arithmetic 183

FLAG parameter 41

FLAGSTD parameter 40, 72

FLOAT option 36

floating-point arithmetic 183

FORMAT phrase 527, 528, 566, 568

format-1 COPY statement 61

format-2 COPY statement 24

FORMATFILE files

- description 460
- sample program 459

formatted dump 372, 631

function keys

- and CONTROL-AREA clause 525
- specifying with DDS
 - See* transaction files

functional processing modules 605

G

general-use programming interfaces

- QCMDEXC 332

generating XML output

- example 303
- overview 301

generation of message monitors 384

generic START statement 480

GENLVL parameter 30

global data 233

global names 220

GOBACK statement 247, 371

graphic data types 451

- restrictions 451

GRAPHIC option 59

group structures, aligning pointers within 337

H

hard control boundary 213

HELP debug command 119

high-level language (HLL) 666

highlights 489

HLL (high-level language) 666

I

I-O feedback 416, 417

I-O-FEEDBACK 417

I/O (input/output), definition 666

I/O devices 409

I/O operation, handling errors 376

i5/OS operating system

- and messages 612
- definition 666
- device control information 523
- device independence and device dependence 409
- input/output 523
- object names 24

IBM extensions

- double-byte character set (DBCS) support 617, 630
- flagging 609
- transaction files 521, 601

IBM Rational Development Studio for System i 8

ICF

- See* intersystem communications function

Identification Division

- and DBCS characters 620
- description 6

identifier

- call by 223

ILE (Integrated Language Environment) 3, 666

ILE C for AS/400

- See* C

ILE CL

- See* CL (Control Language)

ILE procedure 6

ILE RPG for AS/400

- See* RPG

INDARA keyword 536

independence, device 409

indexed files

- creation 489, 500
- description 477
- key fields 477
- processing methods for types DISK and DATABASE 477
- updating 489, 502

indexed I-O module 606

indicators

- and ASSIGN clause 536
- and Boolean data items 536
- and COPY statement 537
- associated with command keys 521
- data description entries 537
- description 536
- example, using in programs 537
- in a separate indicator area 536, 537
- in the record area 537
- INDARA DDS keyword 536
- INDICATOR clause 548
- INDICATORS phrase 537
- sample programs 537
- TRANSACTION file processing 536
- using 536

initialization of storage 214

initializing files with deleted records 477

initializing pointers

- with NULL figurative constant 338

inline data files 411

input field 522

input records 415

input spool 411

- Input-Output Section, description 6
- input-output devices 409
- input-output operations, handling errors 376
- input-output verbs, processing of 377
- input/output (I/O), definition 666
- inquiry messages 371
- INSPECT statement 625
- Integrated Language Environment (ILE) 3, 666
- internal file status 381
- International Standards Organization (ISO) xxvi
- interprogram calls using pointers 342
 - in teraspace memory 336
- interprogram communication considerations 211
- interprogram module 606
- intersystem communications function (ICF)
 - ACCESS MODE clause 525
 - ASSIGN clause 524
 - communications 549
 - CONTROL-AREA clause 525
 - definition 666
 - FILE STATUS clause 525
 - multiple and single device files 554
 - ORGANIZATION clause 525
 - using to specify subfiles 549
- intrinsic functions 607
 - collating sequence and 178
 - conversion uses
 - case, upper or lower 173
 - data items 173
 - numbers 174
 - order, reverse 174
 - reverse order 174
 - data items, evaluating 178
 - data types handled and 170
 - date and time 171
 - examples 171
 - financial 172
 - fixed-point arithmetic and 183, 185
 - floating-point arithmetic and 183, 185
 - largest
 - data items 179
 - length
 - data items 180
 - LENGTH OF special register 180
 - mathematical 173
 - number-handling and 171
 - numeric function nesting 170
 - smallest
 - data items 179
 - statistical 173
 - subscripting, all 170
 - table item processing 185
 - variable-length results 178
 - WHEN-COMPILED special register and 181
 - year 2000 problem and 186
- introduction to ILE COBOL 3
- invalid key condition 379
- INVALID KEY phrase 379
- invariant characters 16
- items grouped by level 73

J

- Java data types 268
- Java Native Interface (JNI) 250
- Java virtual machine (JVM) 250
- JDK11INIT member 275
- JNI member 270
- job failure, recovery 387
- job level scoping 412, 422
- JUSTIFIED clause 622

K

- key fields
 - contiguous, multiple 480
 - descending keys 487
 - for indexed files 477
 - partial keys 480
 - program-defined 485
- keyed read 410
- keyed sequence 405, 473, 477, 479, 487
- keys
 - common 405
 - record 405
 - validity 479
- keywords
 - DDS 403
 - in syntax diagrams xxvii
 - INDARA 536

L

- LANGID parameter 44
- language elements
 - See program structure
- language-identifier-name option 44
- last-used state, description 226, 247
- LDA (local data area) 244
- LEFT debug command 119
- length (LENGTH) field 70
- LENGTH OF special register 234, 339
- length of statement, maximum 13
- level checking (LVLCHK) 406, 666
- level of data item (LVL) field 70
- level of language support 605, 606, 609
- libraries, test 117
- library-name option 29, 44, 86
- library, definition 11
- limitations
 - TGTRLS parameter 49
- LINAGE clause 459
- LINAGE-COUNTER special register 459
- linkage items, setting the address of 340
- Linkage Section
 - describing data to be received 235
 - parameters for a called program 235
- linkage type, identifying 215
- listing view 120
- listings
 - binder 91
 - binder information 94
 - binding statistics 96
 - brief summary table 93
 - command option summary 91
 - command summary 63
 - cross-reference 73, 95
 - Data Division map 69, 70

- listings (*continued*)
 - DBCS characters in 630
 - example, source listing 66, 68
 - examples of 63, 65
 - extended summary table 92
 - FIPS messages 72
 - messages
 - description 75
 - example 74
 - from ILE COBOL compiler 613
 - options 65
 - scanning for syntax errors 63
 - verb usage by count 69
- literals
 - DBCS 617, 619, 628
 - delimiting 31
 - mixed 617
- LNC messages 613
- LNR messages 613
- local data 233
- local data area (LDA), definition 244
- local names 220
- Local Storage
 - recursive calls 225
- lock level
 - high, under commitment control 418
 - low, under commitment control 418
- lock state 413
- locking, file and record 413
- logical file considerations 485
- logical operators xxviii
- LVLCHK (level checking) 406, 666

M

- main program, description 213
- major/minor return codes 384
- manuals, other 675
- maximum source statement length 13
- maximum-number option 38
- member type
 - See source member type
- members 413
- memory management
 - See segmentation
- MERGE statement 430, 435, 628
- merging/sorting files
 - describing the file 428
 - ending sort/merge operation 434
 - example 435
 - input procedure 432
 - merge operation 430
 - output procedure 433
 - restrictions 433
 - return code 434
 - sort criteria 431
 - sort operation 430
 - sorting variable length records 435
- message files 612
- message monitor generation 384
- messages
 - Application Development ToolSet 613
 - compilation 612
 - compile-time 611
 - descriptions 611
 - diagnostic 74

- messages (*continued*)
 - field on diagnostic messages
 - listing 75
 - FIPS 612
 - inquiry 371
 - interactive 613
 - listing 613
 - responding to in an interactive environment 614
 - run time 613
 - severity levels 611
 - statistics 75
 - types 613
- methodology for entering programs 11
- migrating
 - to ILE COBOL language 647
- mismatched records, reducing occurrence 236
- mixed language application 330
- mixed literal 617
- module export 23
- module import 24
- module object
 - creating 7, 23
 - definition 7, 21, 77, 78
 - modifying 97
- module observability 101
- MODULE parameter 28
- module-name option 28
- Monitor Message (MONMSG)
 - command 24
- monitoring exceptions 24
- monitors, message 384
- MONMSG (Monitor Message)
 - command 24
- MOVE statement
 - moving DBCS characters 626
 - using pointers 340
- MQSeries 249
- MSGID and severity level field 75
- MSGLMT parameter 37
- multiple contiguous key fields 480
- multiple device files 554, 563
- multiple members 413
- multiple source programs 60
- multithreading 361

N

- name, assignment 409, 536, 537, 621
- NAMES field 74
- national data
 - in generated XML documents 301
 - in XML document 293
- national language sort sequence 49
- NATIONAL-OF intrinsic function 175
- nested program
 - calling 217
 - calling hierarchy 219
 - calls to, description 214
 - conventions for using 218
 - definition 4
 - global names 220
 - local names 220
 - structure of 217
- NEXT debug command 119
- NEXT MODIFIED phrase 567

- NLSSORT 49
- NO LOCK phrase, and performance 414
- NO REWIND phrase 468
- nonstandard language extensions
 - See* IBM extensions
- NOT AT END phrase 378
- NOT INVALID KEY phrase 379
- notation, syntax xxvii
- nucleus module 606
- NULL figurative constant 338
- null values 357, 443
- null-capable fields 443
- null-terminated strings
 - example 208
 - manipulating 208

O

- Object Definition Table (ODT) 666
- object names, i5/OS 24
- OCCURS clause 622
- ODP (open data path) 415, 666
- ODT (Object Definition Table) 666
- offset, relative to 16-byte boundary 342
- OMITTED data 234
- open data path (ODP) 415, 666
- OPEN operation, increasing speed of 415
- OPEN statement 460, 467, 470, 526, 566
- OPEN type 413
- OPEN-FEEDBACK 416, 624
- operational descriptors 235
- operators, arithmetic and logical xxviii
- OPM (original program model) 3, 329, 666
- optimization level, changing 98
- OPTIMIZE parameter 39
- optimizing code 39
- option indicator 536
- OPTION parameter 30, 62
- optional clauses xxix
- optional divisions 6
- optional items, syntax xxviii
- optional processing modules 606
- optional words, syntax xxvii
- options
 - for the PROCESS statement 61
 - listing 65
 - of CRTCBMOD/CRTBNDCBL
 - command parameters 28, 49, 59
- OPTIONS listing 65
- ORGANIZATION clause 458, 466, 470
- ORGANIZATION IS INDEXED
 - clause 477
- original program model (OPM) 3, 329, 666
- output
 - compiler 61
 - compiler, displaying 63
 - output field 522
 - OUTPUT parameter 29
 - output spool 411
 - overflow condition 374
- Override Message File (OVRMSGF)
 - command 612
- Override to Diskette File (OVRDKTF)
 - command 410

- overriding compiler options 51
- overriding messages 612
- overriding program specifications 412
- OVRDKTF command 410
- OVRMSGF command 612

P

- paging and spacing control for printer files 459
- paper positioning 459
- parameters
 - describing in the called program 235
 - matching the parameter list 313
 - parameters of CRTBNDCBL command
 - See* Create Bound COBOL (CRTBNDCBL) command
 - parameters of CRTCBMOD
 - command
 - See* Create COBOL Module (CRTCBMOD) command
- parsing
 - XML documents 277, 279
- partial key, referring to 480
- parts of a COBOL program
 - See* program structure
- parts of a program 4
- passing data
 - CALL...BY REFERENCE or CALL...BY CONTENT 233
 - in groups 236
 - to ILE C for AS/400 programs 315
 - to ILE CL programs 327
 - to ILE RPG for AS/400 programs 324
- passing data item and its length 234
- passing pointers between programs 356
- PCML 250
 - COBOL and 250
 - Example 253
 - Support for COBOL Datatypes 251
- PEP (program entry procedure) 22, 211, 666
- Performance collection 102
- performing arithmetic 168
- PGM parameter 86
- phrases
 - ADVANCING 460
 - ADVANCING PAGE 460
 - AT END 378
 - END-OF-PAGE 459
 - FORMAT 527, 528, 566, 568
 - INDICATORS 537
 - INVALID KEY 379
 - NEXT MODIFIED 567
 - NO REWIND 468
 - NOT AT END 378
 - NOT INVALID KEY 379
 - REEL/UNIT 468
 - ROLLING 527
 - STARTING 527
 - SUBFILE 550
 - TERMINAL 527, 528, 566, 568
- PICTURE clause 155, 622
- PIP (program initialization parameters)
 - data area 246
- pointer alignment, definition 336

- pointer data items
 - definition 335
 - elementary items 340
- pointers
 - aligning on boundaries
 - 01-level items 337
 - 77-level items 337
 - automatically using FILLER 337
 - with blocking in effect 337
 - and REDEFINES clause 338
 - assigning null value 357
 - defining 336
 - defining alignment 336
 - definition 335
 - description 335
 - examples
 - accessing user space 343
 - processing chained list 355
 - in CALL statement 342
 - in File Section 337
 - in Linkage Section 235
 - in MOVE statement 340
 - in records 339
 - in tables 337
 - in teraspace memory 336
 - in Working-Storage 337
 - initializing 338
 - length of 335
 - manipulating data items 336
 - moving between group items 342
 - null value 357
 - procedure pointer 358
 - processing a chained list 355
 - reading 339
 - writing 339
- portability considerations
 - See segmentation
- position of PROCESS statement 52
- preface xi
- prestart job 246
- PREVIOUS debug command 119
- previous release, compiling for 48
- PRINTER device 457
- printer file
 - definition 457
 - describing FORMATFILE files 460
 - describing PRINTER files 459
 - example 461
 - naming 458
 - writing to 460
- printing
 - based on indicators 459
 - editing field values 459
 - in overflow area 459
 - maintaining print formats 459
 - multiple lines 459
 - paging 459
 - paper positioning 459
 - spacing 459
 - to a printer file 460
- procedure
 - COBOL procedure 6
 - ILE procedure 6
- procedure branching statements 628
- Procedure Division
 - and DBCS characters 623
 - and transaction files 526, 565

- Procedure Division (*continued*)
 - description 6
 - using SET statement to specify address 340
- procedure-pointer 358
- PROCESS statement
 - and DBCS characters 618
 - compiler options specified in 52
 - compiler output 62
 - considerations
 - blocking output records 415
 - commitment control
 - considerations 417
 - DATABASE files 473
 - DISK files 473
 - file and record locking 413
 - overriding program
 - specifications 412
 - processing methods for types DISK and DATABASE 477
 - program-described and externally described files 397
 - spooling 410
 - unblocking input records 415
 - COPY statement, using with 61
 - date window algorithm, overriding 51
 - description 51
 - options 61
 - position of statement 52
 - rules for 51
 - scope of options with
 - CRTBLMOD/CRTBNDCBL commands 61
 - specifying compiler options 65
 - techniques
 - file processing 489
 - indexed file creation 500
 - indexed file updating 502
 - relative file creation 493
 - relative file retrieval 497
 - relative file updating 495
 - sequential file creation 489
 - sequential file updating and extension 491
 - using to specify compiler options 51
 - processing methods for DATABASE files 474
 - processing methods for DISK files 474
 - PROCESSING-INSTRUCTION-DATA
 - XML event 283
 - PROCESSING-INSTRUCTION-TARGET
 - XML event 283
 - producing XML output 301
 - program control
 - returning 225
 - transferring 214
 - program device 526, 529, 566, 568
 - program entry procedure (PEP) 22, 211, 666
 - program initialization parameters (PIP)
 - data area
 - See PIP data area
 - program listings, DBCS characters in 630
 - program object
 - calling 7

- program object (*continued*)
 - major steps in creating 3
 - running 7, 111
- program parts 4
- Program status structure 373
- program structure
 - Data Division 6
 - Data Division map 70
 - Environment Division 6
 - example 4
 - Identification Division 6
 - level of language support 606, 607
 - Procedure Division 6
 - required and optional divisions 6
 - skeleton program 4
- program template 4
- program termination
 - abnormal 114
 - file considerations 211
 - initialization 214
 - passing return code information 232
 - returning control 225, 317, 326, 329
 - STOP RUN statement 225, 227
 - with the CANCEL statement 247
- program-defined key fields 485
- program-described files
 - considerations for using 398
 - description 397
 - externally described by DDS with
 - Create File commands 397
 - TRANSACTION files 521
- program-name option 86
- publications 675
- purpose of this manual xi

Q

- QCBLESRC (default source file) 13
- QCBLESRC option 29
- QCMDEXC, using in a program 332
- QDKT diskette file 469
- QLBLMSG compile-time message file 612
- QLBLMSG run-time message file 612
- QlnDumpCobol bindable API 372
- QlnRtvCobolErrorHandler bindable API 372
- QlnSetCobolErrorHandler bindable API 116, 372
- QPXXCALL, using in a program 331
- QPXXDLTE, using in a program 331
- QTAPE tape file 465
- QTIMSEP system value 63
- quadruple spacing 62
- QUAL debug command 118

R

- READ statement
 - DBCS data items 624
 - format, nonsubfile 527
 - format, subfile 567
- READ WITH NO LOCK 413, 418
- record format
 - composition for display device 522
 - DDS for subfiles 551, 553

- record format (*continued*)
 - example, record format
 - specification 398, 400, 402
 - fields 522
 - indicators 536
 - specification, use of DDS keywords
 - in 399
 - subfiles 549
- RECORD KEY clause
 - description 406
 - EXTERNALLY-DESCRIBED-KEY 406
- record keys 405
- RECORD KEYS, valid 479
- record length of source file 12
- records
 - blocking output 415
 - containing pointers 339
 - locking
 - and failed I/O 414
 - by COBOL 413
 - updating database records 413
 - preserving sequence of 475
 - reducing mismatches 236
 - unblocking input 415
- recovery
 - description 387
 - example 389
 - procedure in program
 - definition 388
 - with multiple acquired devices 388
 - with one acquired device 388
 - transaction files 388
 - with commitment control 387
- recursion 214, 314
- Recursion 225
- recursive call, definition 214
- REDEFINES Clause
 - DBCS characters 621
 - pointer data item as subject or object 338
- redirecting files 410, 413
- REEL/UNIT phrase 468
- reference modification
 - calculating offset 342
- reference numbers 68, 75
- REFERENCES field 74
- references to other manuals xi
- referring to a partial key 480
- Register a User-Written Condition Handler (CEEHDLR) bindable API 371
- related printed information 675
- relative files
 - creating 489, 493
 - definition 475
 - in COBOL 475
 - initializing for output 477
 - retrieval of 489, 497
 - sequential access 476
 - updating 489, 495
- relative I-O module 606
- relative key, definition 550
- RELEASE statement 433, 628
- releasing a record read for update 414
- remote systems, communications
 - between 246, 521
- RENAMES 623

- RENAMES clause 623
- Reorganize Physical File Member (RGZPFM) command 477
- REPLACE parameter 41, 86
- REPLACE statement 66
- replacement text 66
- reply modes 115
- report writer module 606
- required
 - clauses xxix
 - divisions 6
 - items, in syntax xxviii
- responding to messages in an interactive environment 614
- response indicator 536
- return codes 384
- return of control from called program
 - from a main program 226
 - from a subprogram 226
 - passing return code information 232
- RETURN statement 433, 628
- RETURN-CODE special register 232, 313
- REUSEDLT option
 - See reusing deleted records
- reusing deleted records
 - indexed files 478
 - relative files 477
 - sequential files 475
- REWRITE statement
 - and DBCS 624
 - for TRANSACTION file 568
- RGZPFM (Reorganize Physical File Member) command 477
- RIGHT debug command 119
- ROLLBACK statement 418
- ROLLING phrase 527
- RPG
 - CALL/CALLB operation code
 - running a COBOL program using 112
 - calling RPG programs 323
 - data type compatibility 325
 - passing data to 324
 - returning control from 326
- run time
 - concepts 211
 - description 211
 - messages 613
 - monitoring exceptions 24
 - program termination 114
 - redirecting files 410
- run unit
 - ANSI defined 212
 - definition 212
 - OPM COBOL/400 run unit 212, 330
- running ILE COBOL programs
 - CALL CL command, using 111
 - description 111
 - HLL CALL statement, using 112
 - menu-driven application, from 113
 - system reply list and reply modes 115
 - user created command, using 114

S

- SAA Common Programming Interface (CPI) support 607
- SAA CPI (Common Programming Interface) support 607
- SAA data types 438
- sample listing 63
- scoping
 - commitment control 421
 - file override 412
- Screen Design Aid (SDA) 666
- screens
 - See displays
- SD (Sort Description) entries 428
- SDA (Screen Design Aid) 666
- SEARCH statement 628
- searching DBCS characters in a table 628
- SECTION field 70
- segmentation 435, 606, 607
- SELECT statement, EXTERNALLY-DESCRIBED-KEY 486
- separate indicator area (SI) attribute 524, 536
- sequence
 - number 12
 - of records, preserving 475
 - sequence error indicator (S) 69
- sequential access mode 474, 477
- sequential files
 - creation 474, 489
 - definition 474
 - in COBOL 474
 - updating and extension 489, 491
- sequential I-O module 606
- service program
 - binder language 106
 - calling 108
 - canceled 109
 - creating 105
 - definition 105
 - example 107
 - sharing data with 108
 - using 105
- SET statement 626
- SEU (source entry utility)
 - browsing a compiler listing 63
 - description 666
 - editing source programs 6, 11, 12
 - entering source programs 6, 11, 12
 - errors
 - listing 74
 - messages at run time 613
 - formats, using 12
 - prompts and formats 12
 - Start Source Entry Utility (STRSEU) command 13
 - syntax-checking 13, 15, 613
 - TYPE parameter 13
- severity level of messages 611
- severity-level 30, 41
- shared files 413
- shared ODP (open data path) 415
- shared records 413
- shared-for-read 413
- shared-for-read lock state 413
- shared-for-update 413

- shared-no-update 413
- shift-in character, definition 618
- shift-out character, definition 618
- sign representation 166
- signature 106
- single device files 554
- size error condition 375
- skeleton program 4
- SKIP statement 62
- SKIP1 statement 62
- SKIP2 statement 62
- SKIP3 statement 62
- slash (/) 15, 62
- soft control boundary 213
- SORT statement 430, 435, 628
- sort-merge module 606
- SORT-RETURN special register 385, 434
- sort/merge operation, handling errors 385
- sorting/merging files
 - describing the file 428
 - ending sort/merge operation 434
 - example 435
 - input procedure 432
 - merge operation 430
 - output procedure 433
 - restrictions 433
 - return code 434
 - sort criteria 431
 - sort operation 430
 - sorting variable length records 435
- source debugger, ILE 7, 118
- Source Entry Utility
 - See* SEU
- source file
 - default 13
 - fields 12
 - program, suppressing listing 66
 - record length 12
- source file format
 - description 12
 - record length 12
- source listing, example 66
- source member type
 - compiling 24
 - specifying 13
 - SQLCBLE 332
 - syntax-checking 13, 332
- SOURCE NAME field 70
- source physical file, definition 11
- source program
 - compiling 21
 - definition 4
 - editing source programs
 - See* SEU (source entry utility)
 - listing 66
- source text manipulation module 606
- source view 121
- source-file-name option 29
- space pointer, definition 335
- spacing 62
- spacing and paging control for printer files 459
- special register
 - XML-CODE 286
 - XML-EVENT 286
 - XML-NTEXT 286
- special register (*continued*)
 - XML-TEXT 286
- special registers
 - ADDRESS OF 234
 - DB-FORMAT-NAME 478
 - LENGTH OF 234
 - implicit definition 339
 - in Procedure Division 339
 - LINAGE-COUNTER 459
 - RETURN-CODE 313
 - SORT-RETURN 385, 434
- SPECIAL-NAMES paragraph 15
- spooling 410, 411
- SQL (Structured Query Language)
 - statements 332, 666
- SQLCBLE member type 332
- SRCFILE parameter 29
- SRCMBR parameter 29
- SRTSEQ parameter 43
- STANDALONE-DECLARATION XML event 281
- standard record length, COBOL source file 12
- standard, for COBOL xxvi
- Start Commitment Control (STRCMTCTL) command 421
- Start Debug (STRDBG) command 117, 122
- Start Source Entry Utility (STRSEU) command 6, 13, 15
- START statement 480, 625
- START-OF-CDATA-SECTION XML event 284
- START-OF-DOCUMENT XML event 280
- START-OF-ELEMENT XML event 281
- STARTING phrase 527
- starting the compiler 24
- statement length, maximum 13
- statement number (STMT) field 70, 75
- statement number, compiler-generated (STMT) 68
- statement view 121
- statements
 - ACCEPT 416, 623
 - ACQUIRE 526, 566
 - arithmetic, in DBCS processing 625
 - CANCEL 247
 - CLOSE 461, 468, 471
 - COLLATING SEQUENCE 118
 - COMMIT 418
 - compiler output 62
 - COPY 397, 628
 - DISPLAY 624
 - DROP 529, 568
 - EJECT 62
 - EXIT PROGRAM 371
 - GOBACK 371
 - in syntax diagrams xxviii
 - INSPECT 625
 - MERGE 430, 435, 628
 - MOVE 626
 - OPEN 460, 467, 470, 526, 566
 - PROCESS 51, 618
 - READ 624
 - RELEASE 433, 628
 - REPLACE 66
 - RETURN 433, 628
- statements (*continued*)
 - REWRITE 624
 - ROLLBACK 418
 - SEARCH 628
 - SET 626
 - SKIP 62
 - SORT 430, 435, 628
 - START 625
 - START, generic 480
 - STOP 227, 628
 - STOP RUN 371
 - STRING 627
 - TITLE 62, 63, 629
 - UNSTRING 627
 - USE 380
 - WRITE 625
- static procedure call
 - description 214
 - performance advantages 220
 - performing 221
 - using 221
- STEP debug command 118, 119
- STGMDL parameter 45, 88
- STOP RUN statement 226, 227, 247, 371
- STOP statement 628
- storage optimization
 - See* segmentation
- storage, initialization of 214
- STRCMTCTL (Start Commitment Control) command 421
- STRDBG (Start Debug) command 117, 122
- string operations, handling errors 374
- STRING statement 627
- strong definition 92
- STRSEU (Start Source Entry Utility) command 6, 13, 15
- structure, program
 - See* program structure
- Structured Query Language (SQL)
 - statements 332, 666
- subfiles
 - acquiring program devices 566
 - closing 568
 - defining using DDS 549
 - describing 565
 - description 549
 - device file 554
 - display file 550
 - dropping program devices 568
 - naming 564
 - opening 566
 - reading 567
 - replacing 568
 - rewriting 568
 - uses of 550
 - writing 566
- subprogram 213
 - linkage 236
- substring
 - See* reference modification
- support for COBOL standard 605
- suppressing source listing 66
- suppression of messages 612
- symbols used in syntax xxviii
- synchronize changes to database records 418

- syntax
 - arithmetic operators xxviii
 - arrows xxix
 - checking, in SEU 13, 15, 63
 - checking, unit of 13
 - diagrams, using xxviii
 - keywords xxvii
 - logical operators xxviii
 - notation xxvii
 - of CRTBNDCBL command 82
 - of CRTBBLMOD command 25
 - optional clauses xxix
 - optional items xxviii
 - optional words xxvii
 - required clauses xxix
 - required items xxviii
 - symbols xxviii
 - user-supplied names xxvii
 - variables xxvii
- syntax checked only clauses and statements xxx
- system override considerations 412
- system reply list 115

T

- table items, attributes of 72
- table-name option 43
- tape file
 - definition 465
 - describing 466
 - end of volume 467
 - naming 465
 - reading 467
 - rewinding and unloading the volume 468
 - storing variable length records 466, 468
 - writing 467
- TAPEFILE device 465
- target release 42, 48
 - *PRV 42, 48
- template, program 4
- teraspace memory 336
- TERMINAL phrase 527, 528, 566, 568
- termination, program 114
- testing ILE COBOL programs
 - and debugging 117
 - breakpoints 129
 - changing variable contents 148
 - displaying table elements 146
 - displaying variables 143
 - file status 416
 - formatted dump 372
 - test libraries 117
- TEXT parameter 30
- text-description option 30
- TGTRLS parameter 42, 48
 - *PRV 42, 48
- THREAD option 59, 361
- time data type 442
- time-separation characters 63
- timestamp data type 442
- TITLE statement 62, 63, 629
- tools for entering source programs 11
- TOP debug command 119

- transaction files
 - ACCESS MODE clause 525
 - acquiring program devices 526
 - and subfiles 550
 - ASSIGN clause 524
 - closing 529
 - command attention (CA) keys 521
 - CONTROL-AREA clause 525
 - data description specifications (DDS) for 521
 - defining 521
 - describing 525
 - description 521
 - display management 521
 - dropping program devices 529
 - externally described 521
 - file status, setting of 381
 - function keys 521
 - major return code 381
 - minor return code 381
 - naming 524
 - opening 526
 - ORGANIZATION clause 525, 564
 - processing externally described 523
 - program-described 521
 - reading from 527
 - RELATIVE KEY clause 550
 - return codes 381
 - sample programs, workstation 529, 537, 555, 569
 - WORKSTATION device 524
 - workstation validity checking 521
 - writing to 527
- transferring control to another program 214
- transferring program control 214
- transforming COBOL data to XML
 - example 303
 - overview 301
- triple spacing 62

U

- UEP (user entry procedure) 22, 211, 667
- UFCB (user file control block) 381
- unattended mode, running the program 612
- unblocking code, generation of 417
- unblocking input records 415
- unit of syntax checking 13
- UNKNOWN-REFERENCE-IN-ATTRIBUTE XML event 284
- UNKNOWN-REFERENCE-IN-CONTENT XML event 285
- UNSTRING statement 627
- UP debug command 119
- updating
 - and extension of sequential files 489, 491
 - indexed files 489, 502
 - relative files 489, 495
 - sequential files 491
- UPSI (user program status indicator) switch 665, 667
- USAGE clause 156
 - USAGE IS POINTER 335

- USAGE clause (*continued*)
 - USAGE IS PROCEDURE-POINTER 335, 358
- USE statement
 - coded examples 491
 - error handling 380, 381
- user entry procedure (UEP) 22, 211, 667
- user file control block (UFCB) 381
- user program status indicator (UPSI) switch 665, 667
- user spaces
 - accessing using APIs 343
- user-supplied names, syntax xxvii
- user-written error handling routines 386
- using a subfile for display 550
- using double-byte characters 617
- USRPRF parameter 86

V

- valid RECORD KEYS 479
- validity checking 521
- VALUE clause 622
- VALUE IS NULL 357
- variable length records 435, 466, 468, 487
- variable-length fields
 - defining 438
 - example of 438, 452
 - length of, example of 439
 - restrictions 439
- variables
 - changing values while testing 148
 - syntax xxvii
- verbs usage by count listing 69
- VERSION-INFORMATION XML event 280
- viewing a compile listing 48
- VisualAge RPG 9

W

- WDS 8
- weak definition 92
- where DBCS characters can be used 620
- Work with Modules (WRKMOD) command 99
- WORKSTATION device 524
- workstations
 - communications between 521
 - sample programs
 - order inquiry 569
 - payment update 583
 - transaction inquiry 529
 - validity checking 521
- WRITE statement
 - and DBCS 625
 - for TRANSACTION file 527, 566
 - format, nonsubfile 527
 - format, subfile 566

X

- XML 249
- XML document
 - accessing 279

- XML document (*continued*)
 - controlling the encoding of 311
 - enhancing
 - example of converting hyphens in element names to underscores 310
 - example of modifying data definitions 307
 - rationale and techniques 306
 - generating
 - example 303
 - overview 301
 - handling errors 294
 - national language 293
 - parser 277
 - parsing 279
 - example 288
 - processing 277
- XML event
 - ATTRIBUTE-CHARACTER 282
 - ATTRIBUTE-CHARACTERS 282
 - ATTRIBUTE-NAME 282
 - ATTRIBUTE-NATIONAL-CHARACTER 282
 - COMMENT 281
 - CONTENT-CHARACTER 283
 - CONTENT-CHARACTERS 283
 - CONTENT-NATIONAL-CHARACTER 284
 - DOCUMENT-TYPE-DECLARATION 281
 - ENCODING-DECLARATION 281
 - END-OF-CDATA-SECTION 284
 - END-OF-DOCUMENT 285
 - END-OF-ELEMENT 284
 - EXCEPTION 285
 - PROCESSING-INSTRUCTION-DATA 283
 - PROCESSING-INSTRUCTION-TARGET 283
 - STANDALONE-DECLARATION 281
 - START-OF-CDATA-SECTION 284
 - START-OF-DOCUMENT 280
 - START-OF-ELEMENT 281
 - UNKNOWN-REFERENCE-IN-ATTRIBUTE 284
 - UNKNOWN-REFERENCE-IN-CONTENT 285
 - VERSION-INFORMATION 280
- XML events
 - description 277
 - processing 280
 - processing procedure 279
- XML exception codes
 - for generating 312, 645
 - handleable 635
 - not handleable 639
- XML GENERATE statement
 - COUNT IN 312
 - NOT ON EXCEPTION 303
 - ON EXCEPTION 311
- XML generation
 - counting generated characters 302
 - description 301
- XML generation (*continued*)
 - enhancing output
 - example of converting hyphens in element names to underscores 310
 - example of modifying data definitions 307
 - rationale and techniques 306
 - example 303
 - handling errors 311
 - ignored data items 302
 - overview 301
- XML output
 - controlling the encoding of 311
 - enhancing
 - example of converting hyphens in element names to underscores 310
 - example of modifying data definitions 307
 - rationale and techniques 306
 - generating
 - example 303
 - overview 301
- XML PARSE statement
 - description 277
 - NOT ON EXCEPTION 294
 - ON EXCEPTION 294
 - using 279
- XML parser
 - conformance 643
 - description 277
- XML parsing
 - CCSID conflict 297
 - description 279
 - overview 277
 - special registers 285
 - terminating 297
- XML processing procedure
 - example 288
 - specifying 279
 - using special registers 285
 - writing 285
- XML-CODE special register
 - description 286
 - using 277
 - using in generating 303
 - with exceptions 294
 - with generating exceptions 311
- XML-EVENT special register
 - description 286
 - using 277, 280
- XML-NTEXT special register 286
 - using 277
- XML-TEXT special register 286
 - using 277

Y

- year 2000 problem 186



Product Number: 5770-WDS

Printed in U.S.A.

SC09-2540-07

