IBM

eserver

iSeries

CL Programming

Version 5

SC41-5721-06

IBM

@server

iSeries

CL Programming

Version 5

SC41-5721-06

Note

Before using this information and the product it supports, be sure to read the information in Appendix E, "Notices," on page 453.

Seventh Edition (April 2004)

This edition applies to version5, release3, modification 0 of IBM Operating System/400 (Program 5722-SS1) and to all subsequent releases and modifications until otherwise indicated in new editions. This version does not run on all reduced instruction set computer (RISC) models nor does it run on CISC models.

This edition replaces SC41-5721-05. This edition applies only to reduced instruction set computer (RISC) systems.

© Copyright International Business Machines Corporation 1997, 2004. All rights reserved.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Figures vii

About CL Programming (SC41-5721) . . ix
Who should read this book ix
Prerequisite and related information ix
How to send your comments x

Chapter 1. Introduction 1
Control Language 1
Procedure 1
Module 1
Program 2
Service Program 2
Command Syntax 2
CL Procedures 3
Command Definition 4
Menus 5
Command Prompters 5
Objects and Libraries 5
Objects 5
Libraries 6
Messages 8
Message Descriptions 8
Message Queues 9
Testing Functions 9

Chapter 2. CL Programming. 11
Creating a CL Program 12
Interactive Entry 12
Batch Entry 13
Parts of a CL Procedure 13
Example of a Simple CL Program 15
Commands Used in CL Procedures 16
Commands Entered on the RQSDTA and CMD Parameters 16
CL Commands 16
Using CL Procedures 18
Working with Variables 21
Declaring a Variable 23
Using Variables to Specify a List or Qualified Name 23
Lowercase Characters in Variables 24
Variables Replacing Reserved or Numeric Parameter Values 25
Changing the Value of a Variable 26
Trailing Blanks on Command Parameters . . . 27
Writing Comments in CL Procedures 29
Controlling Processing within a CL Procedure . . . 29
Using the GOTO Command and Labels 30
Using the IF Command 30
Using the ELSE Command 32
Using Embedded IF Commands 34
Using the DO Command and DO Groups . . . 35
Using the DOUNTIL Command 36
Using the DOWHILE Command 37
Using the DOFOR Command 38
Using the ITERATE Command 39
Using the LEAVE Command 40
Using the SELECT Command and SELECT Groups 40
Using the *AND, *OR, and *NOT Operators . . 41
Using the %BINARY Built-In Function 45
Using the %SUBSTRING Built-In Function . . . 47
Using the %SWITCH Built-In Function 49
Using the Monitor Message (MONMSG) Command 50
Values That Can Be Used as Variables 53
Retrieving System Values 53
Retrieving Configuration Source 55
Retrieving Configuration Status 56
Retrieving Network Attributes 56
Retrieving Job Attributes 56
Retrieving Object Descriptions 57
Retrieving User Profile Attributes 58
Retrieving Member Description Information . . 58
Working with CL Procedures 59
Logging CL Procedure Commands 60
CL Module Compiler Listings 61
Errors Encountered during Compilation 63
Obtaining a Procedure Dump 63
Displaying Module Attributes 65
Displaying Program Attributes 65
Return Code Summary 65
Compiling Source Programs for a Previous Release 66
Previous-Release (*PRV) Libraries 66
Installing CL Compiler Support for a Previous Release 67

Chapter 3. Controlling Flow and Communicating between Programs and Procedures 69
CALL Command 69
CALLPRC Command 70
RETURN Command 72
Passing Parameters between Programs and Procedures 72
Using the CALL Command 75
Common Errors When Calling Programs and Procedures 78
Using Data Queues to Communicate between Programs and Procedures 82
Remote Data Queues 85
Comparisons with Using Database Files as Queues 86
Similarities to Message Queues 86
Prerequisites for Using Data Queues 87
Managing the Storage Used by a Data Queue . . 87
Allocating Data Queues 87
Examples Using a Data Queue 88

© Copyright IBM Corp. 1997, 2004

Creating Data Queues Associated with an Output Queue . . . 92
Using Data Areas to Communicate between Procedures and Programs . . . 92
Local Data Area . . . 93
Group Data Area . . . 94
Program Initialization Parameter (PIP) Data Area 94
Remote Data Areas . . . 95
Creating a Data Area . . . 95
Data Area Locking and Allocation . . . 96
Displaying a Data Area . . . 96
Changing a Data Area . . . 96
Retrieving a Data Area . . . 96
Retrieve Data Area Examples . . . 96
Changing and Retrieving a Data Area Example 97

Chapter 4. Objects and Libraries . . . 99
Object Types and Common Attributes . . . 99
Functions Performed on Objects . . . 99
Functions the System Performs Automatically . . 99
Functions You Can Perform Using Commands 100
Libraries . . . 100
Library Lists. . . . 101
Displaying a Library List . . . 109
Using Generic Object Names . . . 109
Searching for Multiple Objects or a Single Object 110
Using Libraries . . . 110
Creating a Library . . . 111
Specifying Authority for Libraries . . . 111
Security Considerations for Objects . . . 113
Default Public Authority for Newly Created Objects . . . 113
Default Auditing Attribute for Newly Created Objects . . . 115
Placing Objects in Libraries . . . 115
Deleting and Clearing Libraries . . . 115
Displaying Library Names and Contents . . . 116
Displaying and Retrieving Library Descriptions 117
OS/400 Globalization. . . . 117
Describing Objects . . . 119
Displaying Object Descriptions . . . 119
Retrieving Object Descriptions. . . . 123
RTVOBJD Example . . . 124
Creation Information for Objects . . . 125
Detecting Unused Objects on the System . . . 125
Moving Objects from One Library to Another . . 131
Creating Duplicate Objects . . . 133
Renaming Objects . . . 135
Compressing or Decompressing Objects . . . 136
Compression of Objects . . . 137
Temporarily Decompressed Objects . . . 137
Automatic Decompression of Objects . . . 138
Deleting Objects . . . 139
Allocating Resources . . . 139
Displaying the Lock States for Objects . . . 142

Chapter 5. Working with Objects in CL Procedures and Programs . . . 145
Accessing Objects in CL Programs . . . 145
Exceptions: Accessing Command Definitions, Files, and Procedures . . . 146
Checking for the Existence of an Object . . . 147
Working with Files in CL Procedures . . . 148
Referring to Files in a CL Procedure . . . 151
Opening and Closing Files in a CL Procedure 151
Declaring a File . . . 152
Sending and Receiving Data with a Display File 153
Writing a CL Program to Control a Menu . . . 155
Overriding Display Files in a CL Procedure . . 156
Working with Multiple Device Display Files . . 157
Receiving Data from a Database File . . . 160
Overriding Database Files in a CL Procedure or Program . . . 160
Referring to Output Files from Display Commands . . . 161

Chapter 6. Advanced Programming Topics . . . 163
Using the QCAPCMD Program . . . 163
Using the QCMDEXC Program . . . 163
Using the QCMDEXC Program with DBCS Data 165
Using the QCMDCHK Program . . . 166
Using Message Subfiles in a CL Program or Procedure . . . 168
Allowing User Changes to CL Commands at Run Time . . . 168
Using the OS/400 Prompter within a CL Procedure or Program . . . 168
Selective Prompting for CL Commands. . . . 170
QCMDEXC with Prompting in CL Procedures and Programs . . . 173
Using the Programmer Menu . . . 173
Uses of the Start Programmer Menu (STRPGMMNU) Command. . . . 174
Command Analyzer Exit Points . . . 175
Application Programming for DBCS Data . . . 175
Designing DBCS Application Programs. . . . 175
Converting Alphanumeric Programs to Process DBCS Data . . . 175
Using DBCS Data in a CL Program . . . 176
Sample CL Programs . . . 177
Initial Program for Setup (Programmer) . . . 177
Moving an Object from a Test Library to a Production Library (Programmer) . . . 177
Saving Specific Objects in an Application (System Operator) . . . 178
Recovery from Abnormal End (System Operator). . . . 178
Submitting a Job (System Operator) . . . 178
Timing Out While Waiting for Input from a Device Display . . . 179
Performing Date Arithmetic . . . 180
Retrieving Program Attributes. . . . 181
Loading and Running an Application from Tape or Optical Media . . . 181
Responsibilities of the Application Writer . . . 181

Chapter 7. Defining Messages . . . 183
Creating a Message File . . . 185

Message Files in Independent ASPs 186
Determining the Size of a Message File 186
Adding Messages to a File 187
Assigning a Message Identifier 187
Defining Messages and Message Help 188
Assigning a Severity Code 189
Defining Substitution Variables 190
Specifying Validity Checking for Replies . . . 192
Sending an Immediate Message and Handling a Reply 192
Defining Default Values for Replies 194
Specifying Default Message Handling for Escape Messages 194
Example of Describing a Message 196
Defining Double-Byte Messages 196
System Message File Searches 197
Searching for a Message File 197
Overriding Message Files 197
Types of Message Queues 201
Creating or Changing a Message Queue . . . 202
Job Message Queues 206

Chapter 8. Working with Messages 211
Sending Messages to a System User 211
Sending Messages from a CL Program 212
Messages 213
Examples of Sending Messages 215
Call Stack Entry Identification on SNDPGMMSG 218
Receiving Messages in a CL Procedure or Program 232
Retrieving Messages in a CL Procedure 237
Removing Messages from a Message Queue . . 238
Monitoring for Messages in a CL Program or Procedure 239
Default Handling 244
Notify Messages 245
Status Messages 245
Preventing the Display of Status Messages . . 246
Break-Handling Programs 247
QSYSMSG Message Queue 249
Messages Sent to QSYSMSG Message Queue 249
Sample Program to Receive Messages from QSYSMSG 267
Using the System Reply List 269
| Reply Handling 272
Message Logging 272
Job Log 272
QHST History Log 282
Format of the History Log 285
Processing the QHST File 286
QHST Job Start and Completion Messages . . 286
Deleting QHST Files 288

Chapter 9. Defining Commands . . . 289
Overview of How to Define Commands 289
Step Description 290
Authority Needed for the Commands You Define 292
Example of Creating a Command 292
How to Define Commands 292
Using the CMD Statement 293
Defining Parameters 294
Data Type and Parameter Restrictions 299
Defining Lists for Parameters 305
Defining a Simple List 307
Defining a Mixed List 311
Defining Lists within Lists 313
Defining a Qualified Name 317
Defining a Dependent Relationship 320
Possible Choices and Values 320
Using Prompt Control 322
Conditional Prompting 322
Additional Parameters 325
Using Key Parameters and a Prompt Override Program 325
Procedure for Using Prompt Override Programs 325
CL Sample for Using the Prompt Override Program 329
Creating Commands 332
Command Definition Source Listing 333
Errors Encountered when Processing Command Definition Statements 335
Displaying a Command Definition 336
Effect of Changing the Command Definition of a Command in a Procedure or Program 337
Changing Command Defaults 339
Writing a Command Processing Program or Procedure 342
Writing a CL or HLL Command Processing Program 342
Writing a REXX Command Processing Procedure 344
Writing a Validity Checking Program 345
Examples of Defining and Creating Commands 346
Calling Application Programs 346
Substituting a Default Value 347
Displaying an Output Queue 347
Displaying Messages from IBM Commands More Than Once 348
Creating Abbreviated Commands 349
Deleting Files and Source Members 349
Deleting Program Objects 350

| **Chapter 10. Documenting Commands 353**
| Commands and Command Help 353
| Writing Command Help 354
| Generating UIM Source for Command Help . . 354
| Sharing Common Help 355
| Organizing Help Text into Help Modules . . . 356
| Generating HTML Source for Command Documentation 356

Chapter 11. Debugging Programs . . . 357
Debugging ILE Programs 357
The ILE Source Debugger 357
Debug Commands 358
Preparing a Program Object for a Debug Session 359
Starting the ILE Source Debugger 360
Adding Program Objects to a Debug Session 361

Removing Program Objects from a Debug Session 362
Viewing the Program Source 364
Changing a Module Object 364
Stepping through the Program Object 371
Stepping over Program Objects 372
Stepping into Program Objects 372
Displaying Variables 373
Changing the Value of Variables 375
Attributes of a Variable Examples 377
Equating a Name with a Variable, Expression, or Command 377
Source Debug National Language Support for ILE CL 377
Debugging OPM Programs 378
Debug Mode 379
The Call Stack 380
Handling Unmonitored Messages 381
Breakpoints 383
Traces 387
Display Functions 391
Displaying the Values of Variables 391
Changing the Values of Variables 392
Using a Job to Debug Another Job 393
Debugging at the Machine Interface Level . . . 396
Security Considerations 396

Appendix A. TFRCTL Command . . . 399
Using the TFRCTL Command 399
Passing Parameters 400

Appendix B. Job Log Output Files 403
Directing a Job Log 403
Model for the Primary Job Log 403

Appendix C. IBM-Supplied Libraries in Licensed Programs (LP). 413
IBM-Supplied Libraries for the OS/400 Licensed Program 413
IBM-Supplied Libraries for Other iSeries Licensed Programs 415

Appendix D. Abbreviations of CL Commands and Keywords. 419
CL Command Verb Abbreviations 419
CL Command Abbreviations 421
CL Command Keyword Abbreviations 432

Appendix E. Notices 453
Programming Interface Information 454
Trademarks 454

Bibliography. 457

Index 459

Figures

1. Example of Accessing a Remote Data Queue 86
2. Example of Call PGM 164
3. Example of an Application Using the LODRUN Command 182
4. Example of runtime call stack 220
5. Example of TOPGMQ(*PRV *) 221
6. Example of using a simple name 223
7. Example of using a complex name 224
8. Example 1 of using *PGMBDY 226
9. Example 2 of using *PGMBDY 227
10. Example 3 of using *PGMBDY 228
11. Example of runtime call stack 230
12. Example of using *CTLBDY. 231
13. Simple List Example 309
14. REXX Simple List Example 310
15. Command Relationships for CL and HLL 343
16. Command Relationships for REXX 344
17. Adding an ILE Program Object to a Debug Session. 361
18. Adding an ILE Program Object to a Debug Session. 362
19. Removing an ILE Program Object from a Debug Session 363
20. Removing an ILE Program Object from a Debug Session 363
21. Display a Module View 364
22. Changing a View of a Module Object 365
23. Setting a Conditional Breakpoint 368
24. Displaying a Variable using F11 (Display variable) 373

© Copyright IBM Corp. 1997, 2004

About CL Programming (SC41-5721)

This book provides a wide-range discussion of OS/400 programming topics, including:

- Control language programming.
- OS/400 programming concepts.
- Objects and libraries.
- Message handling.
- User-defined commands.
- User-defined menus.
- Testing functions.

Who should read this book

This book is intended for the OS/400 programmer or application programmer, including non-CL programmers. While CL programming is discussed in detail, much of the material in this book applies to the system in general and may be used by programmers of all high-level languages supported by the iSeries servers.

Prerequisite and related information

Use the iSeries Information Center as your starting point for iSeries technical information.

You can access the Information Center two ways:

- From the following Web site:

  `http://www.ibm.com/eserver/iseries/infocenter`
- From the *iSeries Information Center*, SK3T-4091-04 CD-ROM. This CD-ROM ships with your new iSeries hardware or IBM Operating System/400 software upgrade order. You can also order the CD-ROM from the IBM Publications Center:

  `http://www.ibm.com/shop/publications/order`

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

With every new hardware order, you receive the *iSeries Setup and Operations CD-ROM*, SK3T-4098-02. This CD-ROM contains IBM @server IBM e(logo)server iSeries Access for Windows and the EZ-Setup wizard. iSeries Access Family offers a powerful set of client and server capabilities for connecting PCs to iSeries servers. The EZ-Setup wizard automates many of the iSeries setup tasks.

For other related information, see the “Bibliography” on page 457.

© Copyright IBM Corp. 1997, 2004

How to send your comments

Your feedback is important in helping to provide the most accurate and high-quality information. If you have any comments about this book or any other iSeries documentation, fill out the readers' comment form at the back of this book.

- If you prefer to send comments by mail, use the readers' comment form with the address that is printed on the back. If you are mailing a readers' comment form from a country or region other than the United States, you can give the form to the local IBM branch office or IBM representative for postage-paid mailing.
- If you prefer to send comments by FAX, use either of the following numbers:
  - United States, Canada, and Puerto Rico: 1-800-937-3430
  - Other countries or regions: 1-507-253-5192
- If you prefer to send comments electronically, use one of these e-mail addresses:
  - Comments on books:

    RCHCLERK@us.ibm.com
  - Comments on the iSeries Information Center:

    RCHINFOC@us.ibm.com

Be sure to include the following:

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

Chapter 1. Introduction

This introduction describes several major concepts of Operating System/400® (OS/400). These concepts are discussed in more detail in the following chapters.

System operation is controlled by the following:

- CL commands. CL commands are used singly in batch and interactive jobs, (such as from the Command Entry display) and in CL programs and procedures.
- Menu options. System operation can be controlled by selecting menu options. Interactive users can use the iSeries server menus to perform many system tasks.
- System messages. System messages are used to communicate between programs and procedures and to communicate between programs and procedures and users. Messages can report both status information and error conditions.

Control Language

Control language (CL) is the primary interface to the operating system and can be used at the same time by users at different workstations. A single control language statement is called a **command**. Commands can be entered in the following ways:

- Individually from a workstation.
- As part of batch jobs.
- As source statements to create a CL program or procedure.

Commands can be entered individually from any command line or the Command Entry display.

To simplify the use of CL, all the commands use a consistent syntax. In addition, the operating system provides prompting support for all commands, default values for most command parameters, and validity checking to ensure that a command is entered correctly before the function is performed. Thus, CL provides a single, flexible interface to many different system functions that can be used by different system users.

For information about specific commands, see the CL topic in the iSeries™ Information Center.

Procedure

A **procedure** is a set of self-contained high-level language statements that performs a particular task and then returns to the caller.

In CL, a procedure usually begins with a PGM statement and ends with an ENDPGM statement.

Module

A **module** is the object that results from compiling high-level language source statements using an ILE (Integrated Language Environment®) compiler. A CL module is created by compiling CL source using the Create CL Module (CRTCLMOD) command. A module must be bound into a program to run.

© Copyright IBM Corp. 1997, 2004

A CL module consists of two parts: A user-written procedure, and a program entry procedure that is generated by the CL compiler. In other HLL (for example, C), a single module may contain multiple user-written procedures.

Program

OS/400® supports two types of programs. An **ILE program** is a program written in an Integrated Language Environment (ILE)-conforming high-level language. ILE programs are OS/400 objects that contain one or more modules. Modules cannot be run until they are bound into programs. These programs must have a program entry procedure. The CL compiler generates a program entry procedure in each module it creates. A single-module ILE program can be created using the Create Bound CL Program (CRTBNDCL) command. The Create Program (CRTPGM) command can be used to create an ILE program that contains module objects generated by different ILE compilers, including ILE CL. For more information about ILE modules, ILE programs and service programs, and binding directories, see ILE Concepts .

An **OPM CL program** is a program that conforms to the original program model (OPM). OPM CL programs are objects that result from compiling source using the Create CL Program (CRTCLPGM) command.

Service Program

A **service program** is an OS/400 object that contains one or more modules. You can run programs that are not bound to service programs if they do not require any procedures from the service program. However, you cannot run any procedures from a service program unless that service program is bound to a program. In order to call procedures in a service program, you must export the procedure name. A service program is created using the Create Service Program (CRTSRVPGM) command

While a program has only one entry point, a service program can have multiple entry points. You cannot call service programs directly. You can call procedures in a service program from other procedures in programs and service programs.

Command Syntax

A command name and parameters make up each command. A command name usually consists of a verb, or action, followed by a noun or phrase that identifies the receiver of the action. Abbreviated words, usually one to three letters, make up the command name. This reduces the amount of typing that is required to enter the command. For example, one of the CL commands is the Send Message command. You would use the command that is named SNDMSG to send a message from a user to a message queue.

The parameters used in CL commands are keyword parameters. The keyword, usually abbreviated the same way as commands, identifies the purpose of the parameter. However, when commands are entered, some keywords may be omitted by specifying the parameters in a certain order (positional specification).

For a list of abbreviations used in CL commands and keywords, see Appendix D, “Abbreviations of CL Commands and Keywords,” on page 419.

CL Procedures

CL programs and procedures are made up of CL commands. The commands are compiled into either an OPM program or an ILE module that can be bound into programs made up of modules written in CL or other languages. Advantages of using CL programs and procedures include:

- Using CL programs and procedures is faster than entering and running the commands individually.
- CL programs and procedures provide consistent processing of the same set of commands and logic.
- Some functions require CL commands that cannot be entered individually and must be part of a CL program or procedure.
- CL programs and procedures can be tested and debugged like other high-level language (HLL) programs and procedures.
- Parameters can be passed to CL programs and procedures to adapt the operations performed by the program or procedure to the particular requirements of that use.
- You can bind CL modules with other ILE high-level language modules into a program.

CL programs and procedures can be used for many kinds of applications. For example, CL procedures can be used to:

- Provide an interface to the user of an interactive application through which the user can request application functions without an understanding of the commands used in the program or procedure. This makes the workstation user's job easier and reduces the chances of errors occurring when commands are entered.
- Control the operation of an application by establishing variables used in the application (such as date, time, and external indicators) and specifying the library list used by the application. This ensures that these operations are performed whenever the application is run.
- Provide predefined routines for the system operator, such as procedures to start a subsystem, to provide backup copies of files, or to perform other operating functions. The use of CL programs and procedures reduces the number of commands the operator uses regularly, and ensures that system operations are performed consistently.

Most of the CL commands provided by the system can be used in CL programs and procedures. Some commands are specifically designed for use in CL programs and procedures and are not available when commands are entered individually. These commands include:

- Logic control commands that can be used to control which operations are performed by the program or procedure according to conditions that exist when the program or procedure is run. For example, *if* a certain condition exists, *then do* certain processing, *else* do some other operation. These logic operations provide both conditional and unconditional branching within the CL program or procedure.
- Data operations that provide a way for the program or procedure to communicate with a workstation user. Data operations let the program or procedure send formatted data to and receive data from the workstation, and allow limited access to the database.
- Commands that allow the program or procedure to send messages to the display station user.

- Commands that receive messages sent by other programs and procedures. These messages can provide normal communication between programs and procedures, or indicate that errors or other exceptional conditions exist.
- The use of variables and parameters for passing information between commands in the program or procedure and between programs and procedures.
- Commands that call other procedures. (Procedures cannot be called from the command line or in the batch job stream.)

Using CL programs and procedures, applications can be designed with a separate program or procedure for each function, and with a CL program or procedure controlling which programs or procedures are run within the application. The application can consist of both CL and other HLL programs or procedures. In this type of application, CL programs or procedures are used to:

- Determine which programs or procedures in the application are to be run.
- Provide system functions that are not available through other HLL languages.
- Provide interaction with the application user.

CL programs and procedures provide the flexibility needed to let the application user select what operations to perform and run the necessary procedures.

Command Definition

Command definition allows system users to create additional commands to meet specific application needs. These commands are similar to the system commands.

Each command on the system has a command definition object and a command processing program (CPP). The command definition object defines the command, including:

- The command name
- The CPP
- The parameters and values that are valid for the command
- Validity checking information the system can use to validate the command when it is entered
- Prompt text to be displayed if a prompt is requested for the command.
- Online help information

The **CPP** is the program called when the command is entered. Because the system performs validity checking when the command is entered, the CPP does not always have to check the parameters passed to it.

The command definition functions can be used to:

- Create unique commands needed by system users while keeping a consistent interface for CL command users.
- Define alternative versions of CL commands to meet the requirements of system users. This function might include having different defaults for parameter values, or simplifying the commands so that some parameters would not need to be entered. Constant values can be defined for those parameters. The IBM-supplied commands should not be changed.

See Chapter 9, “Defining Commands” for a detailed discussion of command definition. See Chapter 10, “Documenting Commands” for information about writing online help information for commands.

Menus

The system provides a large number of menus that allow users to perform many functions just by selecting menu options. The advantages of using menus to perform system tasks include:

- Users do not need to understand CL commands and command syntax.
- The amount of typing and the chance of errors are greatly reduced.

Information about creating menus that can be used like the system-supplied menus is described in the Application Display Programming book.

Command Prompters

Command prompters allow you to prompt for command parameters and values. The prompters can be invoked directly or called from application programs. Using the prompters allows you to easily build syntactically correct CL command strings, because the prompters insert parameter keyword names and parameter delimiters, such as apostrophes and parentheses, for you. The CL prompters also provide access to online command help, which can be used to describe the command, parameters and parameter values, command examples, and error messages signaled by the command.

iSeries Navigator provides a graphical CL command prompter for use on a client PC. iSeries Access for Web provides an HTML form-based CL command prompter for use in a web browser.

OS/400 provides a CL command prompter that you can use from the command line by pressing F4. In addition, the Display Command Line Window (QUSCMDLN) API allows you display a command line from within an application.

Objects and Libraries

An **object** is a named storage space that consists of a set of characteristics that describes the object and, in some cases, data. An object is anything that exists in and occupies space in storage and on which operations can be performed. The attributes of an object include its name, type, size, the date it was created, and a description provided by the user who created the object. The value of an object is the collection of information stored in the object. The value of a program, for example, is the code that makes up the program. The value of a file is the collection of records that makes up the file. The concept of an object simply provides a term that can be used to refer to a number of different items that can be stored in the system, regardless of what the items are.

Objects

The functions performed by most of the CL commands are applied to objects. Some commands can be used on any type of object and others apply only to a specific type of object.

The system supports various unique types of objects. Some types identify objects common to many data processing systems, such as:

- Files
- Programs
- Commands

- Libraries
- Queues
- Modules
- Service programs

Other object types are less familiar, such as:
- User profiles
- Job descriptions
- Subsystem descriptions
- Device descriptions

Different object types have different operational characteristics. These differences make each object type unique. For example, because a file is an object that contains data, its operational characteristics differ from those of a program, which contains instructions.

Each object has a name. The object name and the object type are used to identify an object. The object name is assigned by the user creating the object. The object type is determined by the command used to create the object. For example, if a program was created and given the name OEUPDT (for *order entry update*), the program could always be referred to by that name. The system uses the object name (OEUPDT) and object type (program) to locate the object and perform operations on it. Several objects can have the same name, but they must either be different object types or be stored in different libraries.

The system maintains integrity by preventing the misuse of certain functions, depending on the object type. For example, the command CALL causes a program object to run. If you specified CALL and named a file, the command would fail unless there happened to be a program with the same name.

Libraries

A **library** is an object that is used to group related objects, and to find objects by name when they are used. Thus, a library is a directory to a group of objects. You can use libraries to group the objects into any meaningful collection. For example, you can group objects according to security requirements, backup requirements, or processing requirements. The amount of available disk storage limits the number of objects that a library can contain, and the number of libraries on the system.

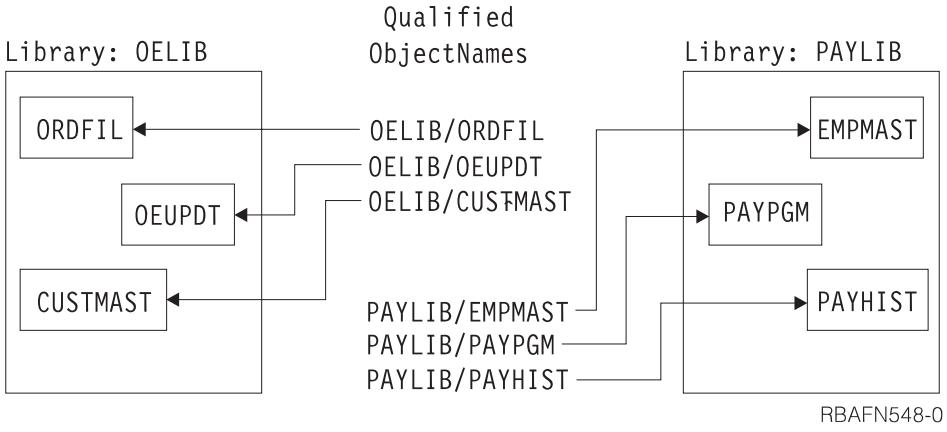
The object grouping performed by libraries is a logical grouping. When a library is created, you can specify into which user auxiliary storage pool (ASP) or independent auxiliary storage pool (independent disk pool) the library should be created. All objects created into the library are created into the same ASP as the library. Objects in a library are not necessarily physically adjacent to each other. The size of a library, or of any other object, is not restricted by the amount of adjacent space available in storage. The system finds the necessary storage for objects as they are stored in the system. See the Independent disk pools topic in the iSeries Information Center for details about independent ASPs.

Most types of objects are placed in a library when they are created. The AUT parameter on CRTLIB defines the public authority of the library. The CRTAUT parameter specifies the default authority for objects that are created into the library. If the command creating the object specifies ***LIBCRTAUT** for the AUT parameter, the object's public authority is the create authority that was specified for the library. You can move most object types from one library to another, but a

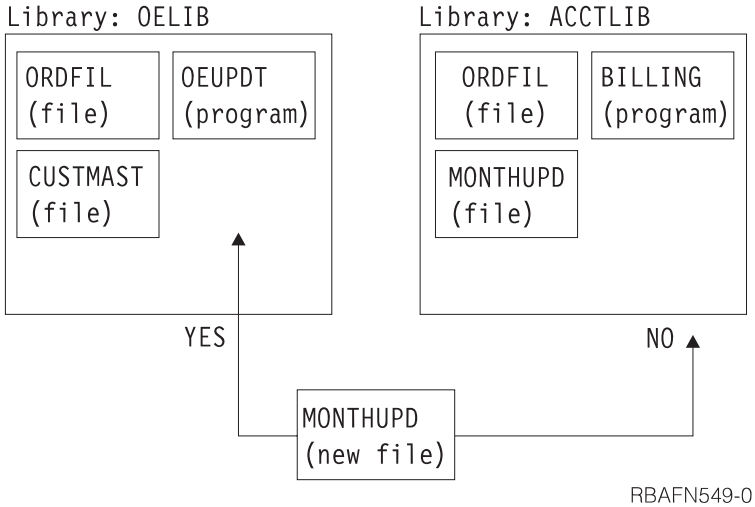
single object cannot be in more than one library at the same time. When you move an object to a different library, the object is not moved in storage. You now locate the object through the new library. You can also rename and copy most object types from one library into another.

A library name can be used to provide another level of identification to the name of an object. As described earlier, an object is identified by its name and its type. The name of the library further qualifies the object name. The combination of an object name and the library name is called the *qualified name* of the object. The qualified name tells the system the name of the object and the library it is in.

The following diagram shows two libraries and the qualified names of the objects in them:



Two objects with the same name and type can exist in different libraries. Two different objects with the same name cannot exist in the same library unless their object types differ. This design allows a program that refers to objects by name to work with different objects (objects with the same name but stored in different libraries) in successive runs of the program without changing the program itself. Also, a workstation user who is creating a new object does not need to be concerned about names used for objects in other libraries. For example, in the following diagram, a new file named MONTHUPD (monthly update) could be added to the library OELIB, but not to the library ACCTLIB. The creation of the file into ACCTLIB would fail because another object named MONTHUPD and of type file already exists in library ACCTLIB.



An object is identified within a library by the object name and type. Many CL commands apply only to a single object type, so the object type does not have to be explicitly identified. For those commands that apply to many object types, the object type must be explicitly identified.

See "Using Libraries" on page 110 for detail on how to use libraries to find objects.

Messages

A **message** is a communication sent from one user, program, or procedure to another. Most data processing systems provide communications between the system and the operator to handle errors and other conditions that occur during processing. OS/400 also provides message handling functions that support two-way communications between programs and system users, between programs, between procedures within a program, and between system users. Two types of messages are supported:

- Immediate messages, which are created by the program or system user when they are sent and are not permanently stored in the system.
- Predefined messages, which are created before they are used. These messages are placed in a message file when they are created, and retrieved from that file when they are used.

Because messages can be used to provide communications between programs, between procedures in a program, and between programs and users, using the OS/400 message handling functions should be considered when developing applications. The following concepts of message handling are important to application development:

- Messages can be defined in messages files, which are outside the programs that use them, and variable information can be provided in the message text when a message is sent. Because messages are defined outside the programs, the programs do not have to be changed when the messages are changed. This approach also allows the same program to be used with message files containing translations of the messages into different languages.
- Messages are sent to and received from message queues, which are separate objects on the system. A message sent to a queue can remain on the queue until it is explicitly received by a program or workstation user.
- A program can send messages to a user who requested the program regardless of what workstation that user has signed on to. Messages do not have to be sent to a specific device; one program can be used from different workstations without change.

See the *Globalization* topic in the **Programming** category of the iSeries Information Center for information on Coded Character Set Identifier (CCSID) for menus, messages, and message descriptions.

Message Descriptions

A **message description** defines a message to OS/400. The message description contains the text of the message and information about replacement variables, and can include variable data that is provided by the message sender when the message is sent.

Message descriptions are stored in message files. Each description must have an identifier that is unique within the file. When a message is sent, the message file and the message identifier tell the system which message description is to be used.

Message Queues

When a message is sent to a procedure, a program, or a system user, it is placed on a **message queue** associated with that procedure, program, or user. The procedure, program, or user sees the message by receiving it from the queue.

OS/400 provides message queues for:

- Each workstation on the system
- Each user enrolled on the system
- The system operator
- The system history log

Additional message queues can be created to meet any special application requirements. Messages sent to message queues are kept, so the receiver of the message does not need to process the message immediately.

Testing Functions

The system includes functions that let a programmer observe operations performed as a program runs. These functions can be used to locate operations that are not performing as intended. Testing functions can be used in either batch or interactive jobs from a workstation. In either case, the program being observed must be in the testing environment, called *debug mode*.

The **testing functions** narrow the search for errors that are difficult to find in the procedure's source statements. Often, an error is apparent only because the output produced is not what is expected. To find those errors, you need to be able to stop the program at a given point (called a *breakpoint*) and examine variable information in the program to see if it is correct. You might want to make changes to those variables before letting the program continue running.

You do not need to know machine language instructions, nor is there a need to include special instructions in the program to use the testing functions. The OS/400 testing functions lets you:

- Stop a running program at any named point in the program's source statements.
- Display information about procedure variables at any point where the program can be stopped. You can also change the variable information before continuing procedure processing.

See either "Debugging ILE Programs" on page 357, for more information on debugging Integrated Language Environment (ILE) programs or "Debugging OPM Programs" on page 378 for more information on debugging OPM programs.

See the appropriate ILE guide for debugging information with other ILE languages.

Chapter 2. CL Programming

The focus of this chapter is ILE rather than OPM. For this reason, 'procedure' is used rather than 'program' for this chapter. However, when the discussion is about CL commands in general, the word 'program' may still be used.

A **CL procedure** is a group of CL commands that tells the system where to get input, how to process it, and where to place the results. The procedure is assigned a name by which it can then be called by other procedures or bound into a program and called. As with other kinds of procedures, you must enter CL procedure source statements, compile, and bind them before you can run the procedure.

When you enter CL commands individually (from the Command Entry display, for instance, or as individual commands in an input stream), each command is separately processed. When you enter CL commands as source statements for a CL procedure, the source remains for later modification if you choose, and the commands are compiled into a module. This module remains as a permanent system object that can be bound into other programs and run. Thus, CL is actually a high-level programming language for system functions. CL procedures ensure consistent processing of groups of commands. You can perform functions with a CL procedure that you cannot perform by entering commands individually, and the CL program or procedure provides better performance at run time than the processing of several separate commands.

CL procedures can be used in batch or interactive processing. Certain commands or functions are restricted to either batch or interactive jobs.

CL source statements consist of CL commands. You cannot use all CL commands as CL source statements, and you can use some of them only in CL procedures or OPM programs.

CL source statements can be entered in a database source member either interactively from a workstation or in a batch job input stream from a device. To create a program using CL source statements, you must enter the source statements into a database source member. You can then create an ILE program by compiling the source member into a module and binding the module into a program object.

CL procedures can be written for many purposes, including:

- To control the sequence of processing and calling of other programs or procedures.
- To display a menu and run commands based on options selected from that menu. This makes the workstation user's job easier and reduces errors.
- To read a database file.
- To handle error conditions issued from commands, programs or procedures, by monitoring for specific messages.
- To control the operation of an application by establishing variables used in the application, such as date, time, and external indicators.
- To provide predefined functions for the system operator, such as starting a subsystem or saving files. This reduces the number of commands the operator uses regularly, and it ensures that system operations are performed consistently.

© Copyright IBM Corp. 1997, 2004

There are many advantages in using CL procedures for an application. For example:

- Because the commands are stored in a form that can be processed when the program is created, using programs is faster than entering and running the commands individually.
- CL procedures are flexible. Parameters can be passed to CL procedures to adapt the operations performed by the procedure to the requirements of a particular use.
- CL procedures can be tested and debugged like other high-level language programs and procedures.
- CL procedures and programs can incorporate conditional logic and special functions not available when commands are entered individually.
- CL procedures can be bound with procedures of other languages.

You cannot use CL procedures to:

- Add or update records in database files.
- Use printer or ICF files.
- Use subfiles within display files.
- Use program-described display files.

Creating a CL Program

All programs are created in steps:

1. Source creation. CL procedures consist of CL commands. In most cases, source statements are entered into a database file in the logical sequence determined by your application design.
2. Module creation. Using the Create Control Language Module (CRTCLMOD) command, this source is used to create a system object. The created CL module can be bound into programs. A CL module contains one CL procedure. Other HLL languages may contain multiple procedures for each module.
3. Program creation. Using the Create Program (CRTPGM) command, this module (along with other modules and service programs) is used to create a program.

Note: If you want to create a program consisting of only one CL module, you can use the Create Bound CL Program (CRTBNDCL) command, which combines steps 2 and 3. If you want to create an OPM CL program from the CL source statements, you can use the Create CL Program (CRTCLPGM) command.

Interactive Entry

OS/400 provides many menus and displays to assist the programmer, including the Programmer Menu, the Command Entry display, command prompt displays, and the Programming Development Manager (PDM) Menu. If you use the OS/400 security functions described in Security - Reference , your ability to use these displays is controlled by the authority given to you in your user profile. User profiles are generally created and maintained by a system security officer.

A frequently used source entry method is the source entry utility (SEU), which is part of the WebSphere Development Studio. You can also use the Edit File (EDTF) command to enter or change CL commands in a database source file. However, EDTF does not provide the integrated CL command prompting support that is built into SEU.

Batch Entry

You can create CL source, a CL module, and a program in one batch input stream. The following example shows the basic parts of the input stream. The input is submitted to a job queue using the Submit Job (SBMJOB) command. The input stream should follow this format:

```
// BCHJOB
CRTBNDCL PGM(QGPL/EDUPGM) SRCFILE(PERLIST)
// DATA FILE(PERLIST) FILETYPE(*SRC)
   .
   .              (CL Procedure Source)
   .
//
/*
// ENDINP
```

This stream creates a program from inline source. If you want to keep the source inline, a Copy File (CPYF) command could be used to copy the source into a database file. The program could then be created using the database file.

You can also create a CL module directly from CL source on external media, such as tape, using an IBM-supplied device file. The IBM-supplied tape source file is QTAPSRC. Assume, for instance, that the CL source statements are in a source file on tape named PGMA.

The first step is to identify the location of the source on tape by using the following override command with LABEL attribute override:

```
OVRTAPF FILE(QTAPSRC) LABEL(PGMA)
```

Now you can consider the QTAPSRC file as the source file on the Create CL Module (CRTCLMOD) command. To create the CL module based on the source input from the tape file, enter the following command:

```
CRTCLMOD MODULE(QGPL/PGMA) SRCFILE(QTAPSRC)
```

When the CRTCLMOD command is processed, it treats the QTAPSRC source file like any database source file. Using the override, the source is located on tape. PGMA is created in QGPL, and the source for that module remains on tape.

Parts of a CL Procedure

While each source statement entered as part of a CL procedure is actually a CL command, the source can be divided into the following basic parts used in many typical CL procedures.

PGM command

`PGM PARM(&A)`

Optional PGM command beginning the procedure and identifying any parameters received.

Declare commands

`(DCL, DCLF, COPYRIGHT)`

Mandatory declaration of procedure variables when variables are used. The declare commands must precede all other commands except the PGM command.

CL processing commands

`CHGVAR, SNDPGMMSG, OVRDBF, DLTF, ...`

CL commands used as source statements to manipulate constants or variables (this is a partial list).

Logic control commands

IF, THEN, ELSE, DO, ENDDO, DOWHILE, DOUNTIL, DOFOR, LEAVE, ITERATE, GOTO, SELECT, ENDSELECT, WHEN, OTHERWISE

Commands used to control processing within the CL procedure.

Built-in functions

%SUBSTRING (%SST), %SWITCH, and %BINARY (%BIN)

Built-in functions and operators used in arithmetic, relational or logical expressions.

Program control commands

CALL, RETURN

CL commands used to pass control to other programs.

Procedure control commands

CALLPRC, RETURN

CL commands used to pass control to other procedures.

ENDPGM command

ENDPGM

Optional End Program command.

The sequence, combination, and extent of these components are determined by the logic and design of your application.

A CL procedure may refer to other objects that must exist when the procedure is created, when the command is processed, or both. This distinction is discussed in "Accessing Objects in CL Programs" on page 145, and in the sections discussing various objects. In some circumstances, for your procedure to run successfully, you may need:

- A display file. Use display files to format information on a device display. If your procedure uses a display, you must enter and create the display file and record format by using the Create Display File (CRTDSPF) command before creating the module. You must declare it to the procedure in the declare section by using the Declare File (DCLF) command. See "Working with Files in CL Procedures" on page 148 for more information.
- A database file. Records in a database file may be read by a CL procedure. If your procedure uses a database file, the file must be created using the Create Physical File (CRTPF) command or the Create Logical File (CRTLF) command before the module is created. You can use Data Description Specifications (DDS), Structured Query Language (SQL), or interactive data definition utility (IDDU) to define the format of the records in the file. The file must also be declared to the procedure in the DCL section using the Declare File (DCLF) command. See "Working with Files in CL Procedures" on page 148 for more information.
- Other programs. If you use a CALL command, the called program must exist before running the CALL command. It does not have to exist when compiling the calling module. See "Accessing Objects in CL Programs" on page 145 and Chapter 3 for more information.
- Other procedures. If you use the CALLPRC command, the called procedure must exist at the time CRTPGM is run. It does not have to exist when CRTCLMOD is run.

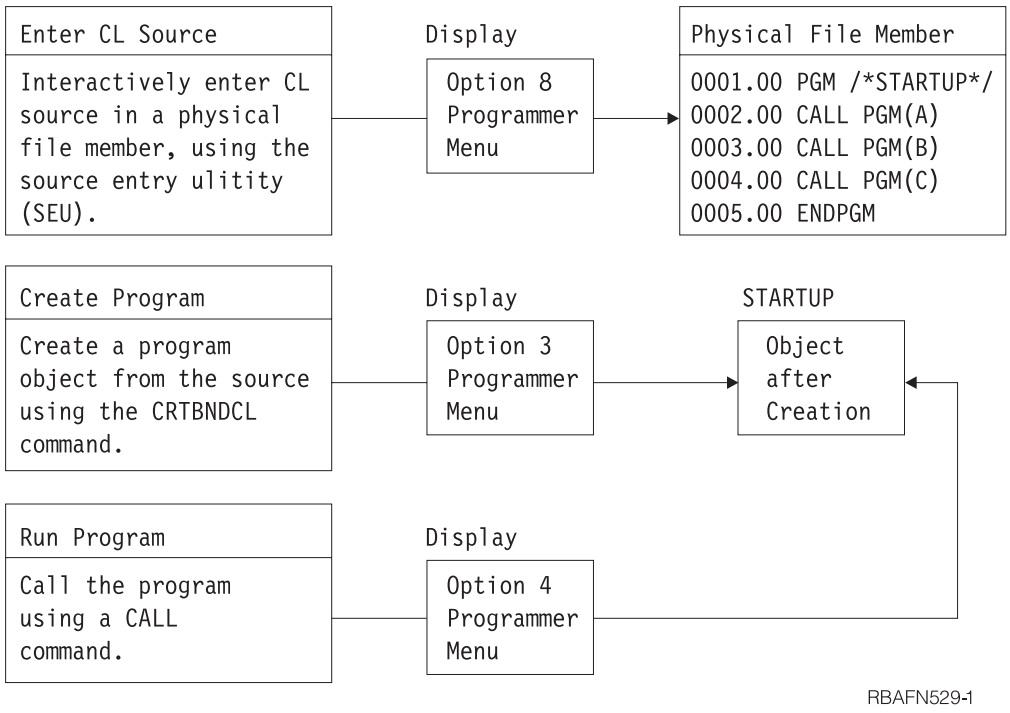
Example of a Simple CL Program

A CL program can be as simple or as complex as you want. To consolidate several activities normally done by the system operator at the beginning of the day (to call programs A, B, and C, for example), you can create a CL procedure STARTUP with the following code:

```
PGM  /*  STARTUP  */
CALL PGM(A)
CALL PGM(B)
CALL PGM(C)
ENDPGM
```

In this example, the Programmer Menu is used to create the program. You could also use the programming development manager (PDM), which is part of the WebSphere Development Studio.

To enter, create, and use this program, follow these steps:



RBAFN529-1

To enter CL source:

- Select option 8 (Edit source) on the Programmer Menu and specify STARTUP in the Parm field. (This option creates a source member named STARTUP that will also be the name of the program.)
- Specify `CLLE` in the Type field and press the Enter key.
- On the SEU display, use the `I` (insert) line command to enter the CL commands (CALL is a CL command).

```
Columns........:   1  71             Edit                        QGPL/QCLSRC
Find......: _____________________________________                    STARTUP
FMT A* .....A*. 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7
      ********** Beginning of data **********************************
.......
.......
.......
.......
.......
.......
```

When you have finished entering the source statements:

- Press F3 to exit from SEU.
- Accept the default on the exit display (option 2, Exit and update member) and press the Enter key to return to the Programmer Menu.
- Select option 3 (Create object) to create a program from the source statements you entered. You do not have to change any other information on the display.

Note: The referenced programs (A, B, and C) do not have to exist when the program STARTUP is created.

When the program is created, you can call it from the Programmer Menu by selecting option 4 (Call program) and specifying STARTUP in the Parm field. If you attempt to run this sample program, however, the referenced programs must exist by the time the CALL commands are run.

Commands Used in CL Procedures

A CL procedure can contain only CL commands. These can be IBM-supplied or commands defined by you. You cannot use some IBM-supplied commands in CL procedures. For information concerning the individual command descriptions and their applicability in CL procedures, see the online help for a particular command, or refer to the *CL* section of the *Programming* category in the **iSeries Information Center**.

Commands Entered on the RQSDTA and CMD Parameters

Certain CL commands, such as Transfer Job (TFRJOB) and Submit Job (SBMJOB) have RQSDTA or CMD parameters that can use another CL command as the parameter value. Commands that can only be used within CL procedures cannot be used as values on the RQSDTA or CMD parameter.

CL Commands

The following is a list of commands that are frequently used in CL procedures. You can use this list to select the appropriate command for the function you want. Refer to the *CL* section of the *Programming* category in the **iSeries Information Center** for information about how to determine the command you might need. Familiarity with the function of these commands will help you to understand subsequent topics in this chapter. Superscript 1 indicates the commands that you can use **only** in CL programs and procedures.

System Function	Command	Command Function
Change Procedure Control	CALL (Call)	Calls a program
	CALLPRC (Call Procedure) [1]	Calls a procedure.
	RETURN (Return)	Returns to the command following the command that caused a program or procedure to be run
CL Procedure Limits	PGM (Program) [1]	Indicates the start of CL procedure source
	ENDPGM (End Program) [1]	Indicates the end of CL procedure source
CL Procedure Logic	IF (If) [1]	Processes commands based on the value of a logical expression
	ELSE (Else) [1]	Defines the action to be taken for the else (false) condition of an IF command
	DO (Do) [1]	Indicates the start of a Do group

System Function	Command	Command Function
	DOWHILE (Do While)[1]	Indicates the start of a Do group that processes a set of commands while the value of a logical expression remains true
	DOUNTIL (Do Until)[1]	Indicates the start of a Do group that processes a set of commands until the value of a logical expression is true
	DOFOR (Do For)[1]	Indicates the start of a Do group that processes commands zero or more times based on specified values
	LEAVE (Leave) [1]	Ends processing of commands in a Do While, Do Until, or Do For group
	ITERATE (Iterate) [1]	Ends processing of commands in a Do While, Do Until, or Do For group, and evaluates the group conditions again
	ENDDO (End Do) [1]	Indicates the end of a Do group
	GOTO (Go To) [1]	Branches to another command
	SELECT (Select)[1]	Indicates the start of a Select group, which allows conditional processing of command groups
	WHEN (When)[1]	Processes commands in a Select group when the value of a logical expression is true
	OTHERWISE (Otherwise) [1]	Defines the commands to be processed if no conditions on a When command in a Select group are true
	ENDSELECT (End Select) [1]	Indicates the end of a Select group
CL Procedure Variables	CHGVAR (Change Variable)[1]	Changes the value of a CL variable
	DCL (Declare) [1]	Declares a variable
Conversion	CHGVAR (Change Variable)[1]	Changes the value of a CL variable
	CVTDAT (Convert Date) [1]	Changes the format of a date
Data Areas	CHGDTAARA (Change Data Area)	Changes a data area
	CRTDTAARA (Create Data Area)	Creates a data area
	DLTDTAARA (Delete Data Area)	Deletes a data area
	DSPDTAARA (Display Data Area)	Displays a data area
	RTVDTAARA (Retrieve Data Area) [1]	Copies the content of a data area to a CL variable
Files	ENDRCV (End Receive) [1]	Cancels a request for input previously issued by a RCVF, SNDF, or SNDRCVF command to a display file
	DCLF (Declare File) [1]	Declares a display or database file
	RCVF (Receive File) [1]	Reads a record from a display or database file
	RTVMBRD (Retrieve Member Description) [1]	Retrieves a description of a specific member of a database file
	SNDF (Send File) [1]	Writes a record to a display file
	SNDRCVF (Send/Receive File) [1]	Writes a record to a display file and reads that record after the user has replied
	WAIT (Wait) [1]	Waits for data to be received from an SNDF, RCVF, or SNDRCVF command issued to a display file
Messages	MONMSG (Monitor Message) [1]	Monitors for escape, status, and notify messages sent to a program's message queue
	RCVMSG (Receive Message) [1]	Copies a message from a message queue into CL variables in a CL procedure
	RMVMSG (Remove Message) [1]	Removes a specified message from a specified message queue
	RTVMSG (Retrieve Message) [1]	Copies a predefined message from a message file into CL procedure variables
	SNDPGMMSG (Send Program Message) [1]	Sends a program message to a message queue
	SNDRPY (Send Reply) [1]	Sends a reply message to the sender of an inquiry message
	SNDUSRMSG (Send User Message)	Sends an informational or inquiry message to a display station or system operator

System Function	Command	Command Function
Miscellaneous Commands	CHKOBJ (Check Object)	Checks for the existence of an object and, optionally, the necessary authority to use the object
	PRTCMDUSG (Print Command Usage)	Produces a cross-reference listing for a specified group of commands used in a specified group of CL procedures
	RTVCFGSRC (Retrieve Configuration Source)	Generates CL command source for creating existing configuration objects and places the source in a source file member
	RTVCFGSTS (Retrieve Configuration Status) [1]	Gives applications the capability to retrieve configuration status from three configuration objects: line, controller, and device.
	RTVJOBA (Retrieve Job Attributes) [1]	Retrieves the value of one or more job attributes and places the values in a CL variable
	RTVSYSVAL (Retrieve System Value) [1]	Retrieves a system value and places it into a CL variable
	RTVUSRPRF (Retrieve User Profile) [1]	Retrieves user profile attributes and places them into CL variables
Program Creation Commands	CRTCLMOD (Create CL Module)	Creates a CL module
	DLTMOD (Delete Module)	Deletes a module
	DLTPGM (Delete Program)	Deletes a program
	CRTBNDCL (Create Bound Control Language Program)	Creates a bound CL program.
	CRTCLPGM (Create CL Program)	Creates an OPM CL program.
	CRTPGM (Create Program)	Creates a program from one or more modules.
	CRTSRVPGM (Create Service Program)	Creates a service program from one or more modules.

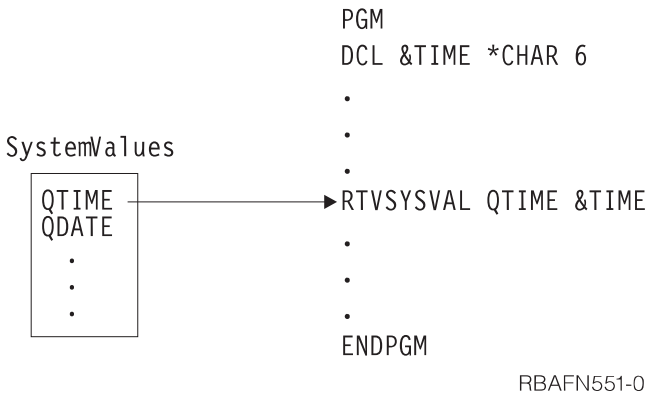
Using CL Procedures

CL programming is a flexible tool allowing you to perform a variety of operations. Each of the following uses is described in greater detail in individual sections later in this chapter. In general, you can:

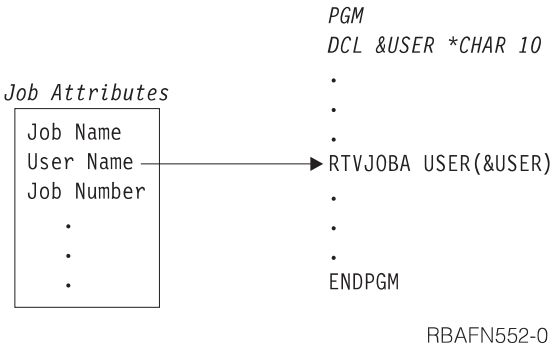
- Use variables, logic control commands, expressions, and built-in functions to manipulate and process data within a CL procedure:

```
PGM
DCL &C *LGL
DCL &A *DEC VALUE(22)
DCL &B *CHAR VALUE(ABCDE)
•
•
•
CHGVAR &A (&A + 30)
•
•
•
IF (&A < 50) THEN(CHGVAR &C '1')
•
DSPLIB ('Q' || &B)
•
IF (%SST(&B 5 1)=E) THEN(CHGVAR &A 12)
•
•
•
ENDPGM
```

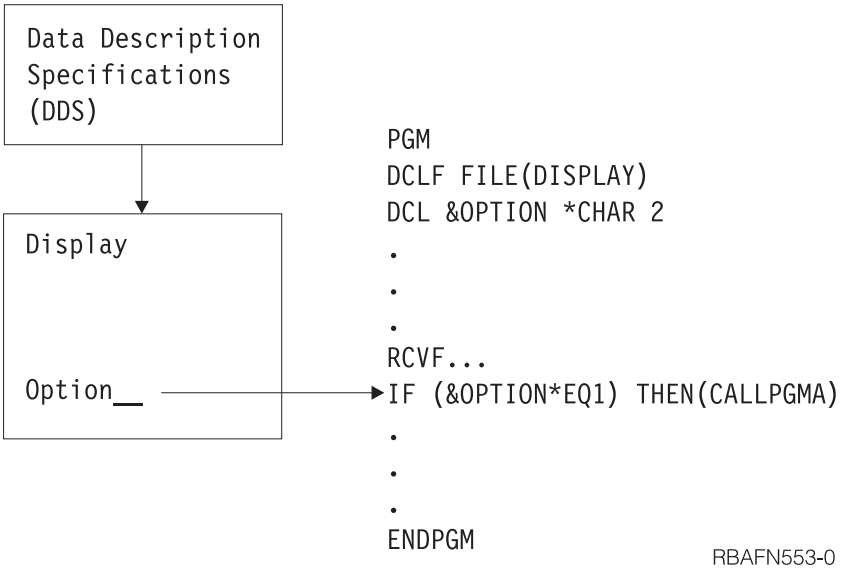
- Use a system value as a variable in a CL procedure.



- Use a job attribute as a variable in a CL procedure.



- Send and receive data to and from a display file with a CL procedure.



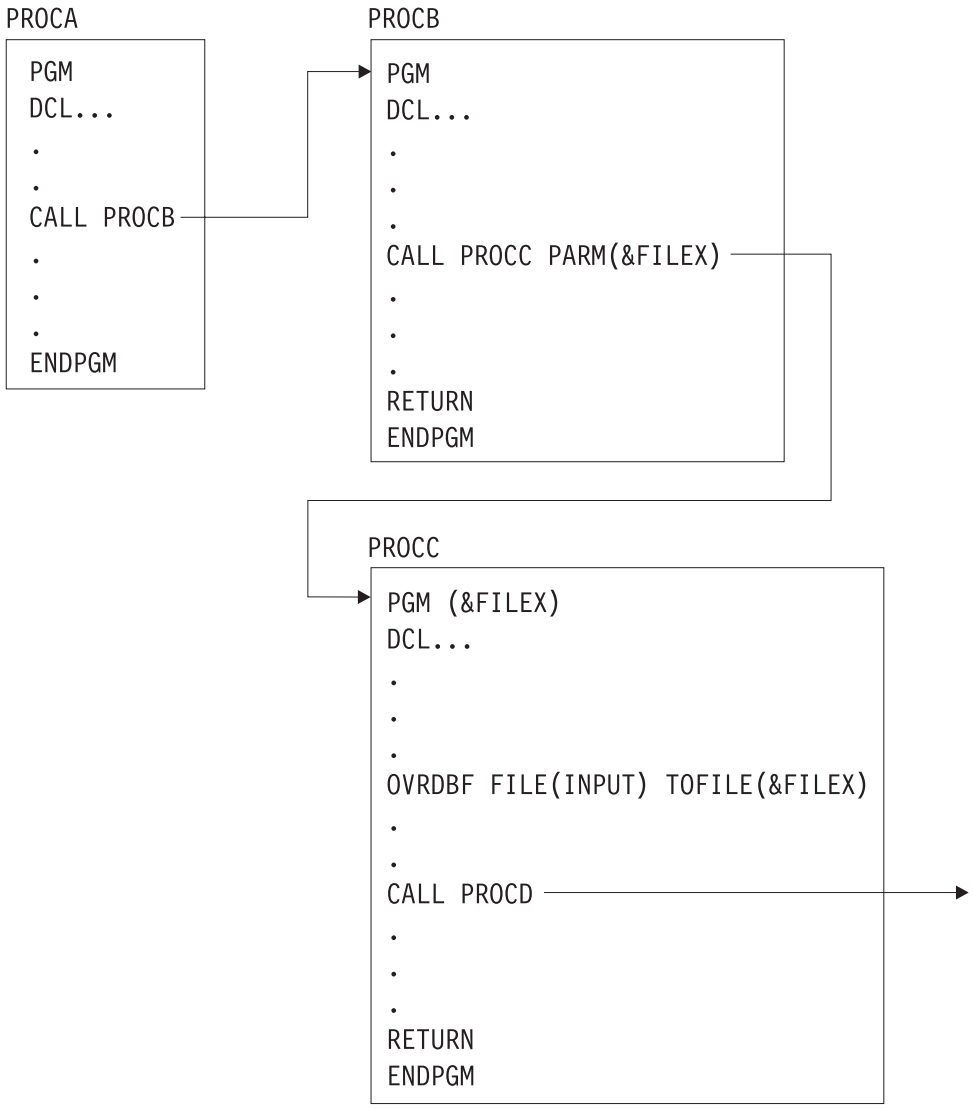
- Create a CL procedure to monitor error messages for a job, and take corrective action if necessary.

```
PGM

MONMSG MSGID(CPF0001) EXEC(GOTO ERROR)
CALL PROGA
```

```
        CALL PROGB
        RETURN
ERROR:  SNDPGMMSG MSG('A CALL command failed') MSGTYPE(*ESCAPE)
        ENDPGM
```

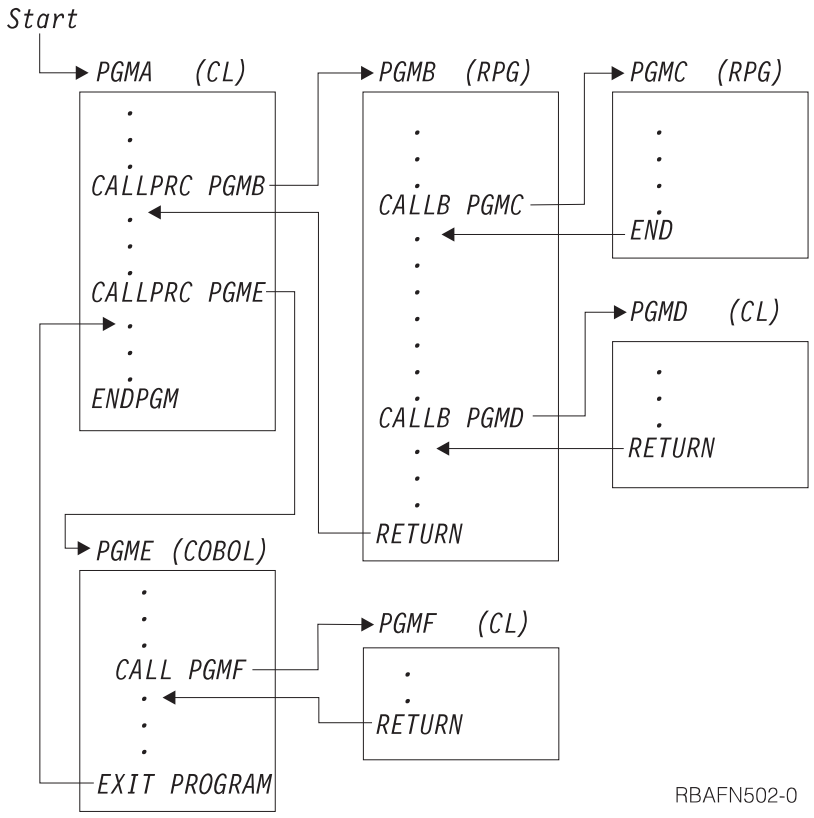
- Control processing among procedures and programs and pass parameters from a CL procedure to other procedures or programs to override files.



Used as a controlling procedure, a CL procedure can call procedures written in other languages. The preceding illustration shows how control can be passed between a CL procedure and RPG IV* and ILE COBOL procedures in an application. To use the application, a workstation user would request program A, which controls the entire application. The illustration shows a single bound program (PGMA) that is called using the CALL command with PGMA. PGMA consists of:

- A CL procedure (PGMA) calling an RPG IV procedure (PGMB)
- An RPG IV procedure (PGMB) calling another RPG IV procedure (PGMC)
- An RPG IV procedure (PGMB) calling a CL procedure (PGMD)
- A CL procedure (PGMA) calling an ILE COBOL procedure (PGME)

- An ILE COBOL procedure (PGME) calling a CL procedure (PGMF)



The procedures can be created as indicated in the following example. You can enter source for procedures in separate source members.

```
CRTCLMOD PGMA
CRTRPGMOD PGMB
CRTRPGMOD PGMC
CRTCLMOD PGMD
CRTCBLMOD PGME
CRTCLMOD PGMF
CRTPGM PGM(PGMA) +
       MODULE(PGMA PGMB PGMC PGMD PGME PGMF) +
       ENTMOD(*FIRST)
```

Working with Variables

CL procedures consist of CL commands, and the commands themselves consist of the command statement, parameters, and parameter values.

Parameter values may be expressed as variables, constants, or expressions. A **variable** is a named changeable value that can be accessed or changed by referring to its name. Variables can be used as substitutes for most parameter values on CL commands. When a CL variable is specified as a parameter value and the command containing it is run, the value of the variable is used as the parameter value. Every time the command is run, a different value can be substituted for the variable. Variables and expressions can be used as parameter values only in CL procedures and programs.

Variables are not stored in libraries; they are not objects; and their values are destroyed when the procedure that contains them is no longer active. The use of variables as values gives CL programming a special flexibility, because this allows

high-level manipulation of objects whose content may change by specific applications. You might, for instance, write a CL procedure to direct the processing of other programs or the operation of several workstations without specifying which programs or workstations are to be controlled. The system identifies these as variables in the CL procedure. You can define (specify) the value of the variables when running the CL procedure.

All variables must be declared (defined) to the CL procedure before they can be used by the procedure:

- Declare variable. Defining it is accomplished using the Declare CL Variable (DCL) command and consists of defining the attributes of the variable. The attributes are type, length, and initial value.

  ```
  DCL  VAR(&AREA)  TYPE(*CHAR)  LEN(4)  VALUE(BOOK)
  ```

- Declare file. If your CL procedure uses a file, you must specify the name of the file in the FILE parameter on the Declare File (DCLF) command. The file contains a description (format) of the records in the file and the fields in the records. During compilation, the DCLF command implicitly declares CL variables for the fields and indicators defined in the file.

  For example, if the DDS for the file has one record in it with two fields (F1 and F2), then two variables, &F1 and &F2, are automatically declared in the program.

  ```
  DCLF  FILE(MCGANN/GUIDE)
  ```

  If the file is a physical file which was created without DDS, one variable is declared for the entire record. The variable has the same name as the file, and its length is the same as the record length of the file.

The declare commands must precede all other commands in the procedure (except the PGM command), but they can be intermixed in any order.

In addition to the uses discussed in this section, variables can be used to:

- Pass information between procedures and jobs. See Chapter 3, “Controlling Flow and Communicating between Programs and Procedures,” on page 69.
- Pass information between procedures and device displays. See “Working with Multiple Device Display Files” on page 157.
- Conditionally process commands. See “Controlling Processing within a CL Procedure” on page 29.
- Create objects. A variable can be used in place of an object name or library name, or both. The following example shows the Create Physical File (CRTPF) command used with a specified library in the first line, and with a variable replacing the library name in the second line:

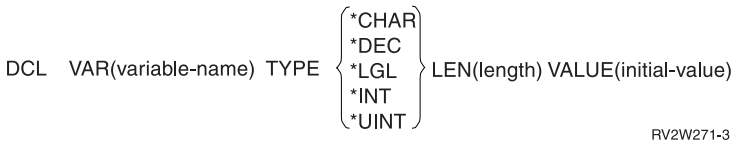
  ```
  CRTPF  FILE(DSTPRODLB/&FILE)
  CRTPF  FILE(&LIB/&FILE)
  ```

Variables cannot be used to change a command name or keyword or to specify a procedure name for the CALLPRC command. Command parameters, however, can be changed during the processing of a CL procedure through the use of the prompting function. See “Allowing User Changes to CL Commands at Run Time” on page 168 for more information.

It is also possible to assemble the keywords and parameters for a command and process it using the QCAPCMD API or QCMDEXC API. See “Using the QCAPCMD Program” on page 163 and “Using the QCMDEXC Program” on page 163 for more information.

Declaring a Variable

In its simplest form, the Declare CL Variable (DCL) command has the following parameters:



When you use a DCL command, you must use the following rules:

- The CL variable name must begin with an ampersand (&) followed by as many as 10 characters. The first character following the & must be alphabetic and the remaining characters alphanumeric. For example, &PART
- The CL variable value must be one of the following:
  - A character string as long as 5000 characters.
  - A packed decimal value totaling up to 15 digits with as many as 9 decimal positions.
  - A logical value '0' or '1', where '0' can mean off, false, or no, and '1' can mean on, true, or yes. A logical variable must be either '0' or '1'.
  - An integer value of two bytes or four bytes. The value can be negative if *INT is specified for the TYPE parameter.
- If you do not specify an initial value, the following is assumed:
  - '0' for decimal variables
  - Blanks for character variables
  - '0' for logical variables.
  - '0' for integer variables.

  For decimal and character types, if you specify an initial value and do not specify the LEN parameter, the default length is the same as the length of the initial value. For type *CHAR, if you do not specify the LEN parameter, the string can be as long as 5000 characters. For type *INT or *UINT, if you do not specify the LEN parameter, the default length is 4.
- Declare the parameters as variables in the program DCL statements.

Using Variables to Specify a List or Qualified Name

The value on a parameter may be a list. For example, the Change Library List (CHGLIBL) command requires a list of libraries on the LIBL parameter, each separated by blanks. The elements in this list can be variables:

```
CHGLIBL LIBL(&LIB1 &LIB2 &LIB3)
```

When variables are used to specify elements in a list, each element must be declared separately:

```
DCL VAR(&LIB1) TYPE(*CHAR) LEN(10) VALUE(QTEMP)
DCL VAR(&LIB2) TYPE(*CHAR) LEN(10) VALUE(QGPL)
DCL VAR(&LIB3) TYPE(*CHAR) LEN(10) VALUE(DISTLIB)
CHGLIBL LIBL(&LIB1 &LIB2 &LIB3)
```

Variable elements cannot be specified in a list as a character string:

Incorrect:

```
DCL   VAR(&LIBS) TYPE(*CHAR) LEN(20) +
      VALUE('QTEMP QGPL DISTLIB')
CHGLIBL LIBL(&LIBS)
```

When presented as a single character string, the system does not view the list as a list of separate elements, and an error will occur.

You can also use variables to specify a qualified name, if each qualifier is declared as a separate variable:

```
DCL VAR(&PGM) TYPE(*CHAR) LEN(10)
DCL VAR(&LIB) TYPE(*CHAR) LEN(10)
CHGVAR VAR(&PGM) VALUE(MYPGM)
CHGVAR VAR(&LIB) VALUE(MYLIB)
.
.
.
DLTPGM PGM(&LIB/&PGM)
ENDPGM
```

In this example, the program and library name are declared separately. The program and library name cannot be specified in one variable, as in the following example:

Incorrect:

```
DCL VAR(&PGM) TYPE(*CHAR) LEN(11)
CHGVAR VAR(&PGM) VALUE('MYLIB/MYPGM')
DLTPGM PGM(&PGM)
```

Here again the value is viewed by the system as a single character string, not as two objects (a library and an object). If a qualified name must be handled as a single variable with a character string value, you can use the built-in function %SUBSTRING and the *TCAT concatenation function to assign object and library names to separate variables. See “Using the %SUBSTRING Built-In Function” on page 47 and Chapter 9 for examples using the %SUBSTRING function.

Lowercase Characters in Variables

Reserved values, such as *LIBL, that can be used as variables must always be expressed in uppercase letters, especially if they are presented as character strings enclosed in apostrophes. For instance, if you wanted to substitute a variable for a library name on a command, the correct code is as follows:

```
DCL VAR(&LIB) TYPE(*CHAR) LEN(10) VALUE('*LIBL')
DLTPGM &LIB/MYPROG;
```

However, it would be *incorrect* to specify the VALUE parameter this way:

```
DCL VAR(&LIB) TYPE(*CHAR) LEN(10) VALUE('*libl')
```

Note that if this VALUE parameter had not been enclosed in apostrophes, it would have been correct, because without the apostrophes it would be translated to uppercase automatically. This error frequently occurs when the parameter is passed as input to a procedure or program from a display as a character string, and the display entry is made in lowercase.

Note: The above paragraph does not take into account the fact that conversion to uppercase is language dependent. **REMEMBER**: Relying on the system to convert values to uppercase may produce unexpected results.

Variables Replacing Reserved or Numeric Parameter Values

Some CL commands allow both numeric or predefined (reserved) values on certain parameters. Where this is true, you can also use character variables to represent the value on the command parameter.

Each parameter on a command can accept only certain types of values. The parameter may allow an integer, a character string, a reserved value, a variable of a specified type, or some mixture of these, as values. Some types of values are required for parameters. If the parameter allows numeric values (if the value is defined in the command as *INT2, *INT4, *UINT2, *UINT4, or *DEC) and also allows reserved values (a character string preceded by an asterisk), you can use a variable as the value for the parameter. The variable must be declared as TYPE(*CHAR) if you intend to use a reserved value.

For example, the Change Output Queue (CHGOUTQ) command has a job separator (JOBSEP) parameter that can have a value of either a number (0 through 9) or the predefined default, *SAME. Because both the number and the predefined value are acceptable, you can also write a CL procedure that substitutes a character variable for the JOBSEP value:

```
      PGM
      DCL &NRESP *CHAR LEN(6)
      DCL &SEP *CHAR LEN(4)
      DCL &FILNAM *CHAR LEN(10)
      DCL &FILLIB *CHAR LEN(10)
      DCLF.....
      .
      .
      .
LOOP: SNDRCVF.....
      IF (&SEP *EQ IGNR) GOTO END
      ELSE IF (&SEP *EQ NONE) CHGVAR &NRESP '0'
      ELSE IF (&SEP *EQ NORM) CHGVAR &NRESP '1'
      ELSE IF (&SEP *EQ SAME) CHGVAR &NRESP '*SAME'
      CHGOUTQ OUTQ(&FILLIB/&FILNAM) JOBSEP(&NRESP)
      GOTO LOOP
END:  RETURN
      ENDPGM
```

In the preceding example, the display station user enters information on a display describing the number of job separators desired for a specified output queue. The variable &NRESP is a character variable manipulating numeric and predefined values (note the use of apostrophes). The JOBSEP parameter on the CHGOUTQ command will recognize these values as if they had been entered as numeric or predefined values. The DDS for the display file used in this program should use the VALUES keyword to restrict the user responses to IGNR, NONE, NORM, or SAME.

If the parameter allows a numeric type of value (*INT2, *INT4, *UINT2, *UINT4, or *DEC) and you do not intend to enter any reserved values (such as *SAME), then you can use a decimal or integer variable in that parameter.

You can find information about the types of values that are allowed by command parameters in Chapter 9. The *CL* section of the *Programming* category in the **iSeries Information Center** contains additional information.

Another alternative for this function is to use the prompter within CL procedures.

Changing the Value of a Variable

You can change the value of a CL variable using the Change Variable (CHGVAR) command. The value can be changed:

- To a constant:

  ```
  CHGVAR   VAR(&INVCMPLT)   VALUE(0)
  ```

  or

  ```
  CHGVAR   &INVCMPLT  0
  ```

  &INVCMPLT is set to 0.
- To the value of another variable:

  ```
  CHGVAR   VAR(&A)   VALUE(&B)
  ```

  or

  ```
  CHGVAR   &A  &B
  ```

  &A is set to the value of the variable &B
- To the value of an expression after it is evaluated:

  ```
  CHGVAR   VAR(&A)   VALUE(&A + 1)
  ```

  or

  ```
  CHGVAR   &A  (&A + 1)
  ```

  The value of &A is increased by 1.
- To the value produced by the built-in function %SST (see "Using the %SUBSTRING Built-In Function" on page 47 for more information):

  ```
  CHGVAR   VAR(&A)   VALUE(%SST(&B 1 5))
  ```

  &A is set to the first five characters of the value of the variable &B
- To the value produced by the built-in function %SWITCH (see "Using the %SWITCH Built-In Function" on page 49 for more information):

  ```
  CHGVAR  VAR(&A)  VALUE(%SWITCH(0XX111X0))
  ```

  &A is set to 1 if job switches 1 and 8 are 0 and job switches 4, 5 and 6 are 1; otherwise, &A is set to 0.
- To the value produced by the built-in function %BIN (see "Using the %BINARY Built-In Function" on page 45 for more information):

  ```
  CHGVAR  VAR(&A)  VALUE(%BIN((%B 1 4))
  ```

  The first four characters of variable &B are converted to the decimal equivalent and stored in variable &A.

The CHGVAR command can be used to retrieve and to change the local data area also. For example, the following commands blank out 10 bytes of the local data area and retrieve part of the local data area:

```
CHGVAR %SST(*LDA 1 10) ' '

CHGVAR &A %SST(*LDA 1 10)
```

The following table shows valid assignments to variables from values (literals or variables).

Table 1. Valid assignments to variables from values

	Logical Value	Character Value	Decimal Value	Signed Integer Value	Unsigned Integer Value
Logical Variable	X				
Character Variable	X	X	X	X	X
Decimal Variable		X	X	X	X

Table 1. Valid assignments to variables from values (continued)

	Logical Value	Character Value	Decimal Value	Signed Integer Value	Unsigned Integer Value
Signed Integer Variable		X	X	X	X
Unsigned Integer Variable		X	X	X	X

Notes:

1. When specifying a numeric value for a character variable, remember the following:
   - The value of the character variable is right-justified and, if necessary, padded with leading zeros.
   - The character variable must be long enough to contain a decimal point and a minus (-) sign, when necessary.
   - When used, a minus (-) sign is placed in the leftmost position of the value.

   For example, &A is a character variable to be changed to the value of the decimal variable &B. The length of &A is 6. The length and decimal positions of &B are 5 and 2, respectively. The current value of &B is 123. The resulting value of &A is `123.00`.
2. When specifying a character value for a numeric variable, remember the following:
   - The decimal point is determined by the placement of a decimal point in the character value. If the character value does not contain a decimal point, the decimal point is placed in the rightmost position of the value.
   - The character value can contain a minus (-) sign or plus (+) sign immediately to the left of the value; no intervening blanks are allowed. If the character value has no sign, the value is assumed to be positive.
   - If the character value contains more digits to the right of the decimal point than can be contained in the numeric variable, the digits are truncated if it's a decimal variable, or rounded if it's an integer variable. If the excess digits are to the left of the decimal point, they are not truncated and an error occurs.

     For example, &C is a decimal variable to be changed to the value of the character variable &D The length of &C is 5 with 2 decimal positions. The length of &D is 10 and its current value is +123.1bbbb (where b=blank). The resulting value of &C is `123.10`.

Trailing Blanks on Command Parameters

Some command parameters are defined with the parameter value of VARY(*YES). This parameter value causes the length of the value passed to be the number of characters between the apostrophes. When a CL variable is used to specify the value for a parameter defined in this way, the system removes trailing blanks before determining the length of the variable to be passed to the command processor program. If the trailing blanks are present and are significant for the parameter, you must take special actions to ensure that the length passed includes them. Most command parameters are defined and used in ways which do not

cause this condition to occur. An example of a parameter defined where this condition is likely to occur is the key value element of the POSITION parameter on the OVRDBF command.

When this condition could occur, the desired result can be attained for these parameters by constructing a command string that delimits the parameter value with apostrophes and passing the string to QCMDEXC or QCAPCMD for processing.

The following is an example of a program that can be used to run the OVRDBF command so that the trailing blanks are included as part of the key value. This same technique can be used for other commands that have parameters defined using the parameter VARY(*YES); trailing blanks must be passed with the parameter.

```
            PGM        PARM(&KEYVAL &LEN)
 /*   PROGRAM TO SHOW HOW TO SPECIFY A KEY VALUE WITH TRAILING        */
 /*      BLANKS AS PART OF THE POSITION PARAMETER ON THE OVRDBF       */
 /*      COMMAND IN A CL PROGRAM.                                     */
 /*   THE KEY VALUE ELEMENT OF THE POSITION PARAMETER OF THE OVRDBF   */
 /*      COMMAND IS DEFINED USING THE VARY(*YES) PARAMETER.           */
 /*      THE DESCRIPTION OF THIS PARAMETER ON THE ELEM COMMAND        */
 /*      DEFINITION STATEMENT SPECIFIES THAT IF A PARAMETER           */
 /*      DEFINED IN THIS WAY IS SPECIFIED AS A CL VARIABLE THE        */
 /*      LENGTH IS PASSED AS THE VARIABLE WITH TRAILING BLANKS        */
 /*      REMOVED. A CALL TO QCMDEXC USING APOSTROPHES TO DELIMIT      */
 /*      THE LENGTH OF THE KEY VALUE CAN BE USED TO CIRCUMVENT        */
 /*      THIS ACTION.                                                 */
 /*    PARAMETERS--                                                   */
            DCL        VAR(&KEYVAL) TYPE(*CHAR) LEN(32)  /*  THE VALUE +
                         OF THE REQUESTED KEY. NOTE IT IS DEFINED AS +
                         32 CHAR.  */
            DCL        VAR(&LEN) TYPE(*INT)              /*  THE LENGTH +
                         OF THE KEY VALUE TO BE USED. ANY VALUE OF  +
                         1 TO 32 CAN BE USED  */
 /*   THE STRING TO BE FINISHED FOR THE OVERRIDE COMMAND TO BE        */
 /*        PASSED TO QCMDEXC (NOTE 2 APOSTROPHES TO GET ONE).         */
            DCL        VAR(&STRING) TYPE(*CHAR) LEN(100) +
                 VALUE('OVRDBF FILE(X3) POSITION(*KEY 1 FMT1  '' ')
 /*   POSITION MARKER  123456789 123456789 123456789 123456789       */
            DCL        VAR(&END) TYPE(*DEC) LEN(15 5)  /*  A VARIABLE +
                         TO CALCULATE THE END OF THE KEY IN &STRING  */

            CHGVAR     VAR(%SST(&STRING 40 &LEN)) VALUE(&KEYVAL)  /*  +
                         PUT THE KEY VALUE INTO COMMAND STRING FOR +
                         QCMDEXC IMMEDIATELY AFTER THE APOSTROPHE.  */
            CHGVAR     VAR(&END) VALUE(&LEN + 40)  /*  POSITION AFTER +
                         LAST CHARACTER OF KEY VALUE  */
            CHGVAR     VAR(%SST(&STRING &END 2)) VALUE(')')  /*  PUT +
                         A CLOSING APOSTROPHE & PAREN TO END  +
                         PARAMETER  */
            CALL       PGM(QCMDEXC) PARM(&STRING 100)  /*  CALL TO +
                         PROCESS THE COMMAND   */
            ENDPGM
```

Note: If you use VARY(*YES) and RTNVAL(*YES) and are passing a CL variable, the length of the variable is passed rather than the length of the data in the CL variable.

Writing Comments in CL Procedures

When you want to write comments in your CL procedures or add comments to commands in your procedures, use the character pairs /* and */. The comment is written between these symbols.

The starting comment delimiter, /*, requires three characters unless the /* characters appear in the first two positions of the command string. In the latter situation, /* can be used without a following blank before a command.

You can enter the three-character starting comment delimiters in any of the following ways (b represents a blank):

```
/*b
b/*
/**
```

Therefore, the starting comment delimiter can be entered four ways. The starting comment delimiter, /*, can:

- Begin in the first position of the command string
- Be preceded by a blank
- Be followed by a blank
- Be followed by an asterisk (/**).

Note: A comment cannot be imbedded within a comment.

For example, in the following procedure, comments are written to describe possible user responses to a set of menu options:

```
        PGM        /* ORD040C ORDER DEPT GENERAL MENU */
        DCLF       FILE(ORD040CD)
START:  SNDRCVF    RCDFMT(MENU)
        SELECT
          WHEN (&RESP=1) THEN(CALL CUS210)   /* CUSTOMER INQUIRY     */
          WHEN (&RESP=2) THEN(CALL ITM210)   /* ITEM INQUIRY         */
          WHEN (&RESP=3) THEN(CALL CUS210)   /* CUSTOMER NAME SEARCH */
          WHEN (&RESP=4) THEN(CALL ORD215)   /* ORDERS BY CUST       */
          WHEN (&RESP=5) THEN(CALL ORD220)   /* EXISTING ORDER       */
          WHEN (&RESP=6) THEN(CALL ORD410C)  /* ORDER ENTRY          */
          WHEN (&RESP=7) THEN(RETURN)
        ENDSELECT
        GOTO START
        ENDPGM
```

Controlling Processing within a CL Procedure

Commands in a CL procedure are processed in consecutive sequence. Each command is processed, one after another, in the sequence in which it is encountered. You can alter this consecutive processing using commands that change the flow of logic in the procedure. These commands can be conditional or unconditional.

Unconditional branching means that you can instruct processing to branch to commands or sets of commands located anywhere in the procedure without regard to what conditions exist at the time the branch instruction is processed. Unconditional processing commands include:

- GOTO
- ITERATE

- LEAVE

Conditional branching means that under certain specified conditions, processing may branch to sections or commands that are not consecutive within the procedure. The branching may be to any statement in the procedure. This is called conditional processing because the branching only occurs when the specified condition is true. Conditional processing is usually associated with the IF command. With the ELSE command, you can specify alternative processing if the condition is not true.The simple DO command allows you to create groups of commands that are always processed together, as a group, under specified conditions. Conditional processing commands include:

- IF and THEN
- SELECT, WHEN, and OTHERWISE
- DOFOR
- DOWHILE
- DOUNTIL

Using the GOTO Command and Labels

The GOTO command processes an unconditional branch. With the GOTO command, processing is directed to another part (identified by a label) of the procedure whenever the GOTO command is encountered. This branching does not depend on the evaluation of an expression. After the branch to the labeled statement, processing begins at that statement and continues in consecutive sequence; it does not return to the GOTO command unless specifically directed back by another instruction. You can branch forward or backward. You cannot use GOTO to go to a label outside the procedure. The GOTO command has one parameter, which contains the label of the statement branched to:

```
GOTO CMDLBL(label)
```

A label identifies the statement in the procedure to which processing is directed by the GOTO command. To use a GOTO command, the command you are branching to must have a label.

```
          PGM
          .
          .
          .
START:    SNDRCVF RCDFMT(MENU)
          IF (&RESP=1) THEN(CALL CUS210)
          .
          .
          .
          GOTO START
          .
          .
          .
          ENDPGM
```

The label in this example is START. A label can have as many as 10 characters and must be immediately followed by a colon, but blanks can occur between the label and the command name.

Using the IF Command

The IF command is used to state a condition that, if true, specifies some statement or group of statements in the procedure to be run. The ELSE command can be used with the IF command to specify a statement or group of statements to be run if the condition expressed by the IF command is false.

The command includes an expression, which is tested (true or false), and a THEN parameter that specifies the action to be taken if the expression is true. The IF command is formatted as follows:

```
IF COND(logical-expression) THEN(CL-command)
```

The logical expression on the COND parameter may be a single logical variable or constant, or it must describe a relationship between two or more operands; the expression is then evaluated as true or false. See "Using the *AND, *OR, and *NOT Operators" on page 41 for more detailed information on the construction of logical expressions.

If the condition described by the logical expression is evaluated as true, the procedure processes the CL command on the THEN parameter. This may be a single command, or a group of commands (see "Using the DO Command and DO Groups" on page 35). If the condition is not true, the procedure runs the next sequential command.

Both COND and THEN are keywords on the command, and they can be omitted for positional entry. The following are syntactically correct uses of this command:

```
IF COND(&RESP=1) THEN(CALL CUS210)
IF (&A *EQ &B) THEN(GOTO LABEL)
IF (&A=&B) GOTO LABEL
```

Blanks are required between the command name (IF) and the keyword (COND) or value (&A). No blanks are permitted between the keyword, if specified, and the left parenthesis enclosing the value.

The following is an example of conditional processing with an IF command. Processing branches in different ways depending on the evaluation of the logical expression in the IF commands. Assume, for instance, that at the start of the following code, the value of &A is 2 and the value of &C is 4.

```
       IF (&A=2) THEN(GOTO FINAL)
       IF (&A=3) THEN(CHGVAR &C 5)
       .
       .
       .
FINAL: IF (&C=5) CALL PROGA
       ENDPGM
```

In this case, the procedure processes the first IF command before branching to FINAL, skipping the intermediate code. It does not return to the second IF command. At FINAL, because the test for `&C=5` fails, PROGA is not called. The procedure then processes the next command, ENDPGM, which signals the end of the procedure, and returns control to the calling procedure.

Processing logic would be different if, using the same code, the initial values of the variables were different. For instance, if at the beginning of this code the value of &A is 3 and the value of &C is 4, the first IF statement is evaluated as false. Instead of processing the GOTO FINAL command, the procedure ignores the first IF statement and moves on to the next one. The second IF statement is evaluated as true, and the value of &C is changed to 5. Subsequent statements, not shown here, are also processed consecutively. When processing reaches the last IF statement, the condition `&C=5` is evaluated as true, and PROGA is called.

A series of consecutive IF statements are run independently. For instance:

```
      PGM /*  IFFY  */
      DCL &A..
      DCL &B..
      DCL &C..
      DCL &D..
      DCL &AREA *CHAR LEN(5) VALUE(YESNO)
      DCL &RESP..
      IF (&A=&B) THEN(GOTO END)   /*  IF #1  */
      IF (&C=&D) THEN(CALL PGMA)   /*  IF #2  */
      IF (&RESP=1) THEN(CHGVAR &C 2)  /*  IF #3  */
      IF (%SUBSTRING(&AREA 1 3) *EQ YES) THEN(CALL PGMB)  /*  IF #4  */
      CHGVAR &B &C
      .
      .
      .
END:  ENDPGM
```

If, in this example, &A is not equal to &B, the next statement is run. If &C is equal to &D, PGMA is called. When PGMA returns, the third IF statement is considered, and so on. Note the difference in logic and processing between these simple sequential IF statements and the use of IF with ELSE or the use of embedded IF commands described later in the chapter (see "Using the ELSE Command" and "Using Embedded IF Commands" on page 34).

An embedded command is a command that is completely enclosed in the parameter of another command. In the following examples, the CHGVAR command and the DO command are embedded:

```
IF (&A *EQ &B) THEN(CHGVAR &A (&A+1))


IF (&B *EQ &C) THEN(DO)
                    .
                    .
                    .
                    ENDDO
```

Using the ELSE Command

The ELSE command is a way of specifying alternative processing if the condition on the associated IF command is false.

The IF command can be used without the ELSE command:

```
IF (&A=&B) THEN(CALLPRC PROCA)
CALLPRC PROCB
```

In this case, PROCA is called only if &A=&B, but PROCB is always called.

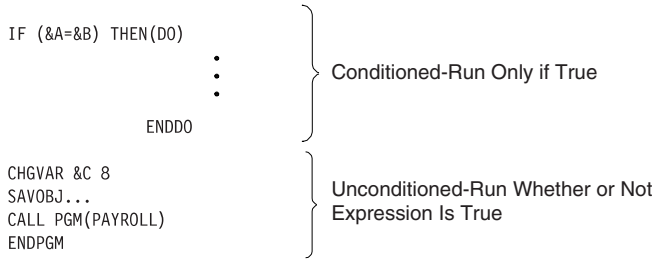
If you use an ELSE command in this procedure, however, the processing logic changes. In the following example, if &A=&B, PROCA is called, and PROCB is not called. If the expression &A=&B is not true, PROCB is called.

```
IF (&A=&B) THEN(CALLPRC PROCA)
ELSE CMD(CALLPRC PROCB)
CHGVAR &C 8
```

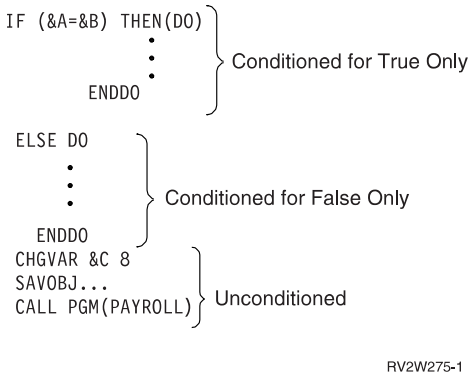
The ELSE command must be used when a false evaluation of an IF expression leads to a distinct branch (that is, an exclusive either/or branch).

The real usefulness of the ELSE command is best demonstrated when combined with Do groups. In the following example, the Do group may not be run,

depending on the evaluation of the IF expression, but the remaining commands are always processed.



With the ELSE command you can specify that a command or set of commands be processed only if the expression is not true, thus completing the logical alternatives:



Each ELSE command must have an associated IF command preceding it. If nested levels of IF commands are present, each ELSE command is matched with the innermost IF command that has not already been matched with another ELSE command.

```
IF ... THEN ...
IF ...THEN(DO)
         IF ...THEN(DO)
                    .
                    .
                    .
         ENDDO
         ELSE DO
                    IF ...THEN(DO)
                        .
                        .
                        .
                        ENDDO
                    ELSE DO
                        .
                        .
                        .
                        ENDDO
         ENDDO
ELSE IF ... THEN ...
IF ... THEN ...
IF ... THEN ...
```

In reviewing your procedure for matched ELSE commands, always start with the innermost set.

The ELSE command can be used to test a series of mutually exclusive options. In the following example, after the first successful IF test, the embedded command is processed and the procedure processes the RCLRSC command:

```
IF  COND(&OPTION=1) THEN(CALLPRC PRC(ADDREC))
 ELSE    CMD(IF COND(&OPTION=2) THEN(CALLPRC PRC(DSPFILE)))
  ELSE   CMD(IF COND(&OPTION=3) THEN(CALLPRC PRC(PRINTFILE)))
    ELSE CMD(IF COND(&OPTION=4) THEN(CALLPRC PRC(DUMP)))
RCLRSC
RETURN
```

Using Embedded IF Commands

An IF command can be embedded in another IF command. This would occur when the command to be processed under a true evaluation (the CL command placed on the THEN parameter) is itself another IF command:

```
IF (&A=&B) THEN(IF (&C=&D) THEN(GOTO END))
GOTO START
```

This can be useful when several conditions must be satisfied before a certain command or group of commands is run. In the preceding example, if the first expression is true, the system then reads the first THEN parameter; within that, if the &C=&D expression is evaluated as true, the system processes the command in the second THEN parameter, GOTO END. Both expressions must be true to process the GOTO END command. If one or the other is false, the GOTO START command is run. Note the use of parentheses to organize expressions and commands.

Up to 25 levels of such embedding are permitted in CL programming.

As the levels of embedding increase and logic grows more complex, you may wish to enter the code in free-form design to clarify relationships:

```
PGM
DCL &A *DEC 1
DCL &B *CHAR 2
DCL &RESP *DEC 1
IF (&RESP=1) +
    IF (&A=5) +
          IF (&B=NO) THEN(DO)
                      .
                      .
                      .
                      ENDDO
CHGVAR &A VALUE(8)
CALL PGM(DAILY)
ENDPGM
```

The preceding IF series is handled as one embedded command. Whenever any one of the IF conditions fails, processing branches to the remainder of the code (CHGVAR and subsequent commands). If the purpose of this code is to accumulate a series of conditions, all of which must be true for the Do group to process, it could be more easily coded using *AND with several expressions in one command. See “Using the *AND, *OR, and *NOT Operators” on page 41.

In some cases, however, the branch must be different depending on which condition fails. You can accomplish this by adding an ELSE command for each embedded IF command:

```
PGM
DCL &A ...
DCL &B ...
DCL &RESP ...
IF (&RESP=1) +
      IF (&A=5) +
             IF (&B=NO) THEN(DO)
                          .
                          .
                          .
                          SNDPGMMSG ...
                          .
                          .
                          .
                         ENDDO
             ELSE CALLPRC PROCA
      ELSE CALLPRC PROCB
CHGVAR &A 8
CALLPRC PROC(DAILY)
ENDPGM
```

Here, if all conditions are true, the SNDPGMMSG command is processed, followed by the CHGVAR command. If the first and second conditions (&RESP=1 and &A=5) are true, but the third (&B=NO) is false, PROCA is called; when PROCA returns, the CHGVAR command is processed. If the second conditions fails, PROCB is called (&B=NO is not tested), followed by the CHGVAR command. Finally, if &RESP does not equal 1, the CHGVAR command is immediately processed. The ELSE command has been used to provide a different branch for each test.

Note: The following three examples are correct syntactical equivalents to the embedded IF command in the preceding example:

```
IF (&RESP=1) THEN(IF (&A=5) THEN(IF (&B=NO) THEN(DO)))

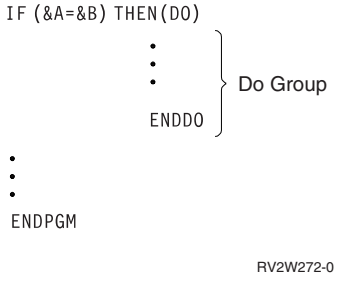
IF (&RESP=1) THEN +
           (IF (&A=5) THEN +
                       (IF (&B=NO) THEN(DO)))

IF (&RESP=1) +
        (IF (&A=5) +
              (IF (&B=NO) THEN(DO)))
```

Using the DO Command and DO Groups

The DO command lets you process a group of commands together. The group is defined as all those commands between the DO command and the corresponding ENDDO command.

Processing of the group is usually conditioned on the evaluation of an associated command. Do groups are most frequently associated with the IF, ELSE, or MONMSG commands. For instance:



RV2W272-0

If the logical expression (&A=&B) is true, then the Do group is processed. If the expression is not true, then processing starts after the ENDDO command; the Do group is skipped.

In the following procedure, if &A is not equal to &B, the system calls PROCB. PROCA is not called, nor are any other commands in the Do group processed.

```
IF (&A=&B) THEN(DO)        ┐
            CALLPRC PROCA  │
            CHGVAR &A &B   ├ Do Group
            SNDPGMMSG...   │
            ENDDO          ┘
 CALLPRC PROCB
 CHGVAR &ACCTS &B
```

RV3W198-0

Do groups can be nested within other Do groups, up to a maximum of 25 levels of nesting.

There are three levels of nesting in the following example. Note how each Do group is completed by an ENDDO command.

```
PGM
 •
 •
 •
IF (&A=&B) DO
          ┌ CALL PGMA
          │ IF (&A=5) DO
          │          ┌ CHGVAR &A 26
          │          │ CALL PGMB
          │          │ IF (&AREA=YES) DO
          │          │
First Nest │ Second │                 ┌ CHGVAR &AREA NO
          │ Nest   │     Third Nest  │ CHGVAR &P (&P+2)
          │          │                 └ ENDDO
          │          │ CALLPRC ACCTSPAY
          │          └ ENDDO
          └ ENDDO
 CALL PGMC
 ENDPGM
```

RV3W199-0

In this example, if &A in the first nest does not equal 5, PGMC is called. If &A does equal 5, the statements in the second Do group are processed. If &AREA in the second Do group does not equal YES, procedure ACCTSPAY is called, because processing moves to the next command after the Do group.

The CL compiler does not indicate the beginning or ending of Do groups. If the CL compiler notes any unbalanced conditions, it is not easy to detect the actual errors.

Using the DOUNTIL Command

The Do Until (DOUNTIL) command processes a group of CL commands one or more times. The group of commands is defined as those commands between the DOUNTIL and the matching ENDDO command.

After the group of commands is processed, the stated condition is evaluated. If the condition is true, the DOUNTIL group will be exited and processing will resume with the next command following the associated ENDDO. If the condition is false, the group will continue processing with the first command in the group.

The logical expression on the COND parameter may be a single logical variable or constant, or it must describe a relationship between two or more operands; the expression is then evaluated as true or false. See "Using the *AND, *OR, and *NOT Operators" on page 41 for more detailed information about the construction of logical expressions.

The following is an example of conditional processing with a DOUNTIL command.

```
DOUNTIL (&LGL)
  .
  .
  .
  CHGVAR &INT (&INT + 1)
  IF (&INT *GT 5) (CHGVAR &LGL '1')
ENDDO
```

The body of the DOUNTIL group will be run at least one time. If the initial value of the &INT variable is 5 or more, &LGL will be set to true on the first time and processing will continue following the ENDDO when the expression is evaluated at the end of the group. If the initial value is less than 5, the body of the group will continue to be repeated until the value of &INT is greater than 5 and the value of &LGL is changed to true.

The LEAVE command may be used to exit the DOUNTIL group and resume processing following the ENDDO. The ITERATE command may be used to skip the remaining commands in the group and evaluate the stated condition immediately.

Using the DOWHILE Command

The DOWHILE command lets you process a group of commands zero or more times while the value of a logical expression is true. The DOWHILE command is used to state a condition that, if true, specifies a command or group of commands in the procedure to run. The group of commands is defined as those commands between the DOWHILE and the matching ENDDO command.

After the group of commands is processed, the stated condition is evaluated. If the condition is false, the DOWHILE group will be exited and processing will resume with the next command following the associated ENDDO. If the condition is true, the group will continue processing with the first command in the group. When the ENDDO command is reached, control returns to the DOWHILE command to again evaluate the condition.

The logical expression on the COND parameter may be a single logical variable or constant, or it must describe a relationship between two or more operands; the expression is then evaluated as true or false. See "Using the *AND, *OR, and *NOT Operators" on page 41 for more detailed information about the construction of logical expressions.

The following is an example of conditional processing with a DOWHILE command.

```
DOWHILE (&LGL)
  .
  .
  .
  IF (&INT *EQ 2) (CHGVAR &LGL '0')
ENDDO
```

When the DOWHILE group is processed, the stated condition will be evaluated. If the condition is true, the group of commands in the DOWHILE group is processed. If the condition is false, processing continues with the command following the associated ENDDO command.

If the value of &LGL is true, the commands in the DOWHILE group will be run until &INT is equal to 2 causing the &LGL variable value to be set to false.

The LEAVE command may be used to exit the DOWHILE group and resume processing following the ENDDO. The ITERATE command may be used to skip the remaining commands in the group and evaluate the stated condition immediately.

Using the DOFOR Command

The DOFOR command lets you process a group of commands a specified number of times.

The DOFOR command specifies a variable, its initial value, an increment or decrement amount, and a terminal value condition. The format of the DOFOR command is:

```
DOFOR VAR(integer-variable) FROM(initial-value) TO(end-value) BY(integer-constant)
```

When processing of a DOFOR group is begun, the integer-variable specified on the VAR parameter is initialized to the initial-value specified on the FROM parameter. The value of the integer-variable is compared to the end-value as specified on the TO parameter. When the integer-constant on the BY parameter is positive, the comparison checks for integer-variable greater than the end-value. If the integer-constant on the BY parameter is negative, the comparison checks for integer-variable less than the end-value.

If the condition is not true, the body of the DOFOR group is processed. When the ENDDO is reached, the integer-constant from the BY parameter is added to the integer-value and the condition is evaluated again.

The following is an example of conditional processing with a DOFOR command.

```
CHGVAR &INT2 0
DOFOR VAR(&INT) FROM(2) TO(4) BY(1)
  .
  .
  .
  CHGVAR &INT2 (&INT2 + &INT)
ENDDO
/* &INT2 = 9 after running the DOFOR group 3 times */
```

When the DOFOR group is processed, &INT is initialized to 2 and the value of &INT is checked to see if it is greater than 4. It is not, so the body of the group is processed. On the second iteration of the group, one is added to &INT and the check is repeated. It is less than 4, so the DOFOR group is processed again. On reaching the ENDDO the second time, the value of &INT is again incremented by 1. &INT now has a value of 4. Since &INT is still less than or equal to 4, the DOFOR group is processed again. On reaching the ENDDO the third time, the value of &INT is again incremented by 1. This time, the value is 5, and processing continues with the command following the ENDDO.

The LEAVE command may be used to exit the DOFOR group and resume processing following the ENDDO. The ITERATE command may be used to skip

the remaining commands in the group, increment the controlling variable, and evaluate the end-value condition immediately.

Using the ITERATE Command

The ITERATE command can be used to skip the remaining commands in an active DOWHILE, DOUNTIL, or DOFOR group. ITERATE is not valid with simple DO command groups.

An ITERATE command without a label will skip to the ENDDO of the innermost active DO group. Specifying a label skips to the ENDDO of the DO associated with the label.

The following illustrates use of the ITERATE command:

```
DO_1:
DO_2:DOWHILE &LGL
DO_3:  DOFOR &INT FROM(0) TO(99)
         .
         .
         .
         IF (&A *EQ 12) THEN (ITERATE DO_1)
         .
         . /* Not processed if &A equals 12        */
         .
         IF (&A *GT 12) ITERATE
         .
         .  /* Not processed if &A greater than 12 */
         .
       ENDDO
         .
         .
         .
       IF (&A *LT 0) (ITERATE DO_1)
         .
         . /* Not processed if &A less than zero   */
         .
     ENDDO
```

In this example, the labels DO_1 and DO_2 are associated with the DOWHILE group. They can be specified on an ITERATE command appearing in either the DOWHILE or DOFOR group. When &A is equal to 12, the ITERATE DO_1 command is run. Processing continues at the ENDDO associated with the DOWHILE command. The value of &LGL is evaluated and, if true, continues with the DOFOR following the DOWHILE. If &LGL is false, processing continues with the CL command following the second ENDDO.

If &A is not equal to 12 but is greater than 12, processing continues with the ENDDO of the DOFOR group. The value of &INT is incremented and compared to the ending value of 99. If &INT is less than or equal to 99, processing continues with the first command following the DOFOR command. If &INT is greater than 99, processing continues with the next command following the first ENDDO.

When the third IF command is processed and &A is less than zero, processing continues with the second ENDDO. The value of &LGL is evaluated and, if false, control passes to the command following the ENDDO. If true, processing resumes with the DOFOR command following the DOWHILE.

Using the LEAVE Command

The LEAVE command can be used to exit an active DOWHILE, DOUNTIL, or DOFOR group. It provides a structured manner to leave an active group without resorting to the use of the GOTO command. LEAVE is not valid with simple DO command groups.

A LEAVE command without a label will leave the innermost active DO group. Specifying a label allows the processing to break out of one or more enclosing groups.

The following illustrates use of the LEAVE command:

```
DO_1:
DO_2:DOWHILE &LGL
DO_3:  DOFOR &INT FROM(0) TO(99)
         .
         .
         .
         IF (&A *EQ 12) THEN(LEAVE DO_1)
         .
         . /* Not processed if &A equals 12        */
         .
         IF (&A *GT 12) LEAVE
         .
         .  /* Not processed if &A greater than 12 */
         .
       ENDDO
         .
         .
         .
       IF (&A *LT 0) (LEAVE DO_1)
         .
         . /* Not processed if &A less than zero   */
         .
     ENDDO
```

In this example, the labels DO_1 and DO_2 are associated with the DOWHILE group. They can be specified on a LEAVE command appearing in either the DOWHILE or DOFOR group. When &A is equal to 12, the LEAVE DO_1 command is run and processing continues with the CL command following the second ENDDO.

If &A is not equal to 12 but is greater than 12, the DOFOR group is exited and processing continues with next command following the first ENDDO.

When the third IF command is processed and &A is less than zero, processing continues with the next command following the first ENDDO.

Using the SELECT Command and SELECT Groups

The SELECT command is used to identify one or more conditions and an associated group of commands to process when that condition is true. A special group of commands may also be specified to be processed when none of the stated conditions are true. Only one of the groups of commands identified by WHEN or OTHERWISE commands will be processed within the group.

The general structure of the SELECT command is as follows:

```
SELECT
  WHEN (condition-1) THEN(command-1)
  .
  .
```

```
  .
  WHEN (condition-n) THEN(command-n)
  OTHERWISE command-x
ENDSELECT
```

A SELECT group must specify at least one WHEN command. The WHEN command includes an expression, which is tested (true or false), and an optional THEN parameter that specifies the action to take if the condition is true.

The logical expression on the COND parameter may be a single logical variable or constant, or it must describe a relationship between two or more operands; the expression is then evaluate as true or false. See "Using the *AND, *OR, and *NOT Operators" for more detailed information about the construction of logical expressions.

If the condition described by the logical expression is evaluated as true, the procedure processes the CL command on the THEN parameter. This may be a single command or a group of commands specified by the DO, DOWHILE, DOUNTIL, or DOFOR commands. If the condition is not true, the condition specified on the next WHEN command in the SELECT group is evaluated. If there is no WHEN command following this one, the command identified by the OTHERWISE command, if any, is processed. If there is no next WHEN and no OTHERWISE command, processing continues with the next command following the associated ENDSELECT command.

```
SELECT
  WHEN (&LGL)
  WHEN (&INT *LT 0) THEN(CHGVAR &INT 0)
  WHEN (&INT *GT 0) (DOUNTIL (&INT *EQ 0))
                       CHGVAR &INT (&INT - 1)
                     ENDDO
  OTHERWISE (CHGVAR &LGL '1')
ENDSELECT
```

If the initial value of &LGL is true ('1'), processing continues with the command following the ENDSELECT, because there is no THEN parameter.

If the initial value of &LGL is false ('0'), the COND of the second WHEN is evaluated. If &INT is less than zero, the CHGVAR is processed, setting the value of &INT to zero. Processing then continues with the command following the ENDSELECT.

If the first two conditions are not met, the value of &INT is checked to determine if it is greater than zero. If the value is greater than zero, the DOUNTIL group is entered and the value of &INT decremented until it reaches zero. When &INT reaches zero, the DOUNTIL group is exited and processing continues with the next command following the ENDSELECT.

If none of the conditions on any of the WHEN commands is evaluated as true, the CHGVAR specified on the CMD parameter of the OTHERWISE command is processed. The value of &INT remains unchanged while &LGL is set to true. Processing then continues with the next command following the ENDSELECT.

Using the *AND, *OR, and *NOT Operators

*AND and *OR are the reserved values for logical operators used to specify the relationship between operands in a logical expression. The ampersand symbol (&) can replace the reserved value *AND, and the vertical bar (|) can replace *OR. The reserved values must be preceded and followed by blanks. The operands in a

logical expression consist of relational expressions or logical variables or constants separated by logical operators. The *AND operator indicates that both operands (on either side of the operator) have to be true to produce a true result. The *OR operator indicates that one or the other of its operands must be true to produce a true result.

Note: Using the ampersand symbol or the vertical bar can cause problems because the symbols are not at the same code point for all code pages. To avoid this, use *AND and *OR instead of the symbols.

Use operators, other than logical operators, in expressions to indicate the actions to perform on the operands in the expression or the relationship between the operands. There are three kinds of operators other than logical operators:

- Arithmetic (+, -, *, /)
- Character (*CAT, ||, *BCAT, |>, *TCAT, |<)
- Relational (*EQ, =, *GT, >, *LT, <, *GE, >=, *LE, <=, *NE, ¬=, *NG, ¬>, *NL, ¬<)

You can find information about these operators in the *CL* section of the *Programming* category in the **iSeries Information Center**.

The following are examples of logical expressions:

```
((&C *LT 1) *AND (&TIME *GT 1430))
(&C *LT 1 *AND &TIME *GT 1430)
((&C < 1) & (&TIME>1430))
((&C< 1) & (&TIME>1430))
```

In each of these cases, the logical expression consists of three parts: two operands and one operator (*AND or *OR, or their symbols). It is the type of operator (*AND or *OR) that characterizes the expression as logical, not the type of operand. Operands in logical expressions can be logical variables or other expressions, such as relational expressions. (Relational expressions are characterized by &gt, &lt, or = symbols or corresponding reserved values.) For instance, in the example:

```
((&C *LT 1) *AND (&TIME *GT 1430))
```

the entire logical expression is enclosed in parentheses, and both operands are relational expressions, also enclosed separately in parentheses. As you can see from the second example of logical expressions, the operands need not be enclosed in separate parentheses, but it is recommended for clarity. Parentheses are not needed because *AND and *OR have different priorities. *AND is always considered before *OR. For operators of the same priority, parentheses can be used to control the order in which operations are performed.

A simple relational expression can be written as the condition in a command:

```
IF (&A=&B) THEN(DO)
             .
             .
             .
             ENDDO
```

The operands in this relational expression could also be constants.

If you wish to specify more than one condition, you can use a logical expression with relational expressions as operands:

```
IF ((&A=&B) *AND (&C=&D)) THEN(DO)
                            .
                            .
                            .
                            ENDDO
```

The series of dependent IF commands cited as an example in "Using Embedded IF Commands" on page 34 could be coded:

```
PGM
DCL &RESP *DEC 1
DCL &A *DEC 1
DCL &B *CHAR 2
IF ((&RESP=1) *AND (&A=5) *AND (&B=NO)) THEN(DO)
                                          .
                                          .
                                          .
                                          ENDDO
CHGVAR &A VALUE(8)
CALLPRC PROC(DAILY)
ENDPGM
```

Here the logical operators are again used between relational expressions.

Because a logical expression can also have other logical expressions as operands, quite complex logic is possible:

```
IF (((&A=&B) *OR (&A=&C)) *AND ((&C=1) *OR (&D='0'))) THEN(DO)
```

In this case, &D is defined as a logical variable.

The result of the evaluation of any relational or logical expression is a '1' or '0' (true or false). The dependent command is processed only if the complete expression is evaluated as true ('1'). The following command is interpreted in these terms:

```
IF   ((&A = &B) *AND (&C = &D)) THEN(DO)

     ((true'1') *AND (not true'0'))
          (not true '0')
```

The expression is finally evaluated as not true ('0'), and, therefore, the DO is not processed. For an explanation of how this evaluation was reached, see the matrices later in this section.

This same process is used to evaluate a logical expression using logical variables, as in this example:

```
PGM
DCL &A *LGL
DCL &B *LGL
IF (&A *OR &B) THEN(CALL PGM(PGMA))
.
.
.
ENDPGM
```

Here the conditional expression is evaluated to see if the value of &A or of &B is equal to '1' (true). If either is true, the whole expression is true, and PGMA is called.

The final evaluation arrived at for all these examples of logical expressions is based on standard matrices comparing two values (referred to here as &A and &B) under an *OR or *AND operator.

Use the following matrix when using *OR with logical variables or constants:

If &A is:
'0' '0' '1' '1'

and &B is:
'0' '1' '0' '1'

the OR expression is:
'0' '1' '1' '1'

In short, for multiple OR operators with logical variables or constants, the expression is false ('0') if all values are false. The expression is true ('1') if any values are true.

```
PGM
DCL &A *LGL VALUE('0')
DCL &B *LGL VALUE('1')
DCL &C *LGL VALUE('1')
IF (&A *OR &B *OR &C) THEN(CALL PGMA)
.
.
.
ENDPGM
```

Here the values are not all false; therefore, the expression is true, and PGMA is called.

Use the following matrix when evaluating a logical expression with *AND with logical variables or constants:

If &A is:
'0' '0' '1' '1'

and &B is:
'0' '1' '0' '1'

the ANDed expression is:
'0' '0' '0' '1'

For multiple AND operators with logical variables or constants, the expression is false ('0') when any value is false, and true when they are all true.

```
PGM
DCL &A *LGL VALUE('0')
DCL &B *LGL VALUE('1')
DCL &C *LGL VALUE('1')
IF (&A *AND &B *AND &C) THEN(CALL PGMA)
.
.
.
ENDPGM
```

Here the values are not all true; therefore, the expression is false, and PGMA is not called.

These logical operators can only be used *within* an expression when the operands represent a logical value, as in the preceding examples. It is *incorrect* to attempt to use OR or AND for variables that are not logical. For instance:

```
PGM
DCL &A *CHAR 3
DCL &B *CHAR 3
DCL &C *CHAR 3

Incorrect:   IF (&A *OR &B *OR &C = YES) THEN...
```

The correct coding for this would be:

```
IF ((&A=YES) *OR (&B=YES) *OR (&C=YES)) THEN...
```

In this case, the ORing occurs between relational expressions.

The logical operator *NOT (or ¬) is used to negate logical variables or constants. Any *NOT operators must be evaluated before the *AND or *OR operators are evaluated. Any values that follow *NOT operators must be evaluated before the logical relationship between the operands is evaluated.

```
PGM
DCL &A *LGL '1'
DCL &B *LGL '0'
IF (&A *AND *NOT &B) THEN(CALL PGMA)
```

In this example, the values are all true; therefore, the expression is true, and PGMA is called.

```
PGM
DCL &A *LGL
DCL &B *CHAR 3 VALUE('ABC')
DCL &C *CHAR 3 VALUE('XYZ')
CHGVAR &A VALUE(&B *EQ &C)
IF (&A) THEN(CALLPRC PROCA)
```

In this example, the value is false, therefore, PROCA is not called.

For more information about logical and relational expressions, see the *CL* section of the *Programming* category in the **iSeries Information Center**.

Using the %BINARY Built-In Function

The binary built-in function (%BINARY or %BIN) interprets the contents of a specified CL character variable as a signed binary integer. The starting position begins at the position specified and continues for a length of 2 or 4 characters.

The syntax of the binary built-in function is:

```
%BINARY(character-variable-name starting-position length)
```

or

```
%BIN(character-variable-name starting-position length)
```

The starting position and length are optional. However, if the starting position and length are *not specified*, a starting position of 1 and length of the character variable that is specified are used. In that case, you must declare the length of the character variable as either 2 or 4.

If the starting position *is specified*, you must also specify a constant length of 2 or 4. The starting position must be a positive number equal to or greater than 1. If the sum of the starting position and the length is greater than the length of the character variable, an error occurs. (A CL decimal or integer variable may also be used for the starting position.)

You can use the binary built-in function with both the IF and CHGVAR commands. It can be used by itself or as part of an arithmetic or logical expression. You can also use the binary built-in function on any command parameter that is defined as numeric (TYPE of *DEC, *INT2, *INT4, *UINT2, or *UINT4) with EXPR(*YES).

When the binary built-in function is used with the condition (COND) parameter on the IF command or with the VALUE parameter on the Change Variable (CHGVAR) command, the contents of the character variable is interpreted as a binary-to-decimal conversion.

When the binary built-in function is used with the VAR parameter on the CHGVAR command, the decimal value in the VALUE parameter is converted to a 2-byte or 4-byte signed binary integer and the result stored in the character variable at the starting position specified. Decimal fractions are truncated.

The system uses the binary built-in function on the RTNVAL parameter of the CALLPRC command to indicate that the calling procedure expects the called procedure to return a signed binary integer.

A 2-byte character variable can hold signed binary integer values from -32 768 through 32 767. A 4-byte character variable can hold signed binary integer values from -2 147 483 648 through 2 147 483 647.

The following are examples of the binary built-in function:

- ```
  DCL    VAR(&B2)  TYPE(*CHAR)  LEN(2)    VALUE(X'001C')
  DCL    VAR(&N)   TYPE(*DEC)   LEN(3 0)
  CHGVAR   &N   %BINARY(&B2)
  ```
  The contents of variable &B2 is treated as a 2-byte signed binary integer and converted to its decimal equivalent of 28. It is then assigned to the decimal variable &N
- ```
  DCL    VAR(&N)   TYPE(*DEC)   LEN(5 0)  VALUE(107)
  DCL    VAR(&B4)  TYPE(*CHAR)  LEN(4)
  CHGVAR   %BIN(&B4)    &N
  ```
  The value of the decimal variable &N is converted to a 4-byte signed binary number and is placed in character variable &B4 Variable &B4 will have the value of X'0000006B'.
- ```
  DCL    VAR(&P)  TYPE(*CHAR)  LEN(100)
  DCL    VAR(&L)  TYPE(*DEC)   LEN(5 0)
  CHGVAR   &L  VALUE(%BIN(&P 1 2) * 5)
  ```
  The first two characters of variable &P is treated as a signed binary integer, converted to its decimal equivalent, and multiplied by 5. The product is assigned to the decimal variable &L.
- ```
  DCL    VAR(&X)   TYPE(*CHAR)  LEN(50)
  CHGVAR   %BINARY(&X 15 2)  VALUE(122.56)
  ```
  The number 122.56 is truncated to the whole number 122 and is then converted to a 2-byte signed binary integer and is placed at positions 15 and 16 of the character variable &X. Positions 15 and 16 of variable &X will contain the hexadecimal equivalent of X'007A'.
- ```
  DCL    VAR(&B4)  TYPE(*CHAR)  LEN(4)
  CHGVAR   %BIN(&B4)  VALUE(-57)
  ```
  The value -57 is converted to a 4-byte signed binary integer and assigned to the character variable &B4. The variable &B4 will then contain the value X'FFFFFFC7'.
- ```
  DCL    VAR(&B2)  TYPE(*CHAR)  LEN(2)   VALUE(X'FF1B')
  DCL    VAR(&C5)  TYPE(*CHAR)  LEN(5)
  CHGVAR   &C5  %BINARY(&B2)
  ```

The contents of variable &B2 is treated as a 2-byte signed binary integer and converted to its decimal equivalent of -229. The number is converted to character form and stored in the variable character &C5. The character variable &C5 will then contain the value '-0229'.

- ```
  DCL    VAR(&C5)  TYPE(*CHAR)  LEN(5)   VALUE(' 1253')
  DCL    VAR(&B2)  TYPE(*CHAR)  LEN(2)
  CHGVAR   %BINARY(&B2)  VALUE(&C5)
  ```
  The character number 1253 in character variable &C5 is converted to a decimal number. The decimal number 1253 is then converted to a 2-byte signed binary integer and stored in the variable &B2. The variable &B2 will then have the value X'04E5'.
- ```
  DCL    VAR(&S)  TYPE(*CHAR)  LEN(100)
  IF     (%BIN(&S 1 2) *GT 10)
         THEN( SNDPGMMSG MSG('Too many in list.') )
  ```
  The first 2 bytes of the character variable &S are treated as a signed binary integer when compared to the number 10. If the binary number has a value larger than 10, then the SNDPGMMSG (Send Program Message) command is run.
- ```
  DCL    VAR(&RTNV) TYPE(*CHAR) LEN(4)
  CALLPRC PRC(PROCA) RTNVAL(%BIN(&RTNV 1 4))
  ```
  Procedure PROCA returns a 4-byte integer which is stored in variable &RTNV.

Using the %SUBSTRING Built-In Function

The substring built-in function (%SUBSTRING or%SST) produces a character string that is a subset of an existing character string and can only be used within a CL procedure. In a CHGVAR command, the %SST function can be specified in place of the variable (VAR parameter) to be changed or the value (VALUE parameter) to which the variable is to be changed. In an IF command, the %SST function can be specified in the expression.

The format of the substring built-in function is:

```
%SUBSTRING(character-variable-name starting-position length)
```

or

```
%SST(character-variable-name starting-position length)
```

You can code *LDA in place of the character variable name to indicate that the substring function is performed on the contents of the local data area.

The substring function produces a substring from the contents of the specified CL character variable or the local data area. The substring begins at the specified starting position (which can be a variable name) and continues for the length specified (which can also be a variable name). Neither the starting position nor the length can be 0 or negative. If the sum of the starting position and the length of the substring are greater than the length of the entire variable or the local data area, an error occurs. The length of the local data area is 1024.

The following are examples of the substring built-in function:

- If the first two positions in the character variable &NAME are IN, the program INV210 is called. The entire value of &NAME is passed to INV210 and the value of &ERRCODE is unchanged. Otherwise, the value of &ERRCODE is set to 99.
  ```
  DCL &NAME *CHAR VALUE(INVOICE)
  DCL &ERRCODE *DEC (2 0)
  IF (%SST(&NAME 1 2) *EQ 'IN') +
  THEN(CALL INV210 &NAME)
  ELSE CHGVAR &ERRCODE 99
  ```
  ```

- If the first two positions of &A match the first two positions of &B, the program CUS210 is called.

```
DCL &A *CHAR VALUE(ABC)
DCL &B *CHAR VALUE(DEF)
IF (%SST(&A 1 2) *EQ %SUBSTRING(&B 1 2)) +
CALL CUS210
```

- Position and length can also be variables: This example changes the value of &X beginning at position &Y for the length &Z to 123.

```
CHGVAR %SST(&X &Y &Z) '123'
```

- If &A is ABCDEFG before this CHGVAR command is run, &A is

```
CHGVAR %SST(&A 2 3) '123'
```

  A123EFG after the command runs.

- In this example, the length of the substring, 5, exceeds the length of the operand YES to which it is compared. The operand is padded with blanks so that the comparison is between YESNO and YESbb (where b is a blank). The condition is false.

```
DCL VAR(&NAME) TYPE(*CHAR) LEN(5) VALUE(YESNO)
.
.
.
IF (%SST (&NAME 1 5) *EQ YES) +
  THEN(CALL PROGA)
```

  If the length of the substring is shorter than the operand, the substring is padded with blanks for the comparison. For example:

```
DCL VAR(&NAME) TYPE(*CHAR) LEN(5) VALUE(YESNO)
.
.
.
IF (%SST(&NAME 1 3 ) *EQ YESNO) THEN(CALL PROG)
```

  This condition is false because YESbb (where bb is two blanks) does not equal YESNO.

- The value of the variable &A is placed into positions 1 through 10 of the local data area.

```
CHGVAR %SST(*LDA 1 10) &A
```

- If the concatenation of positions 1 through 3 of the local data area plus the constant 'XYZ' is equal to variable &A, then PROCA is called. For example, if positions 1 through 3 of the local data area contain 'ABC' and variable &A has a value of ABCXYZ, the test is true and PROCA is called.

```
IF (((%SST*LDA 1 3) *CAT 'XYZ') *EQ &A) THEN(CALLPRC PROCA)
```

- This procedure scans the character variable &NUMBER and changes any leading zeros to blanks. This can be used for simple editing of a field before displaying in a message.

```
      DCL &NUMBER *CHAR LEN(5)
      DCL &X *DEC LEN(3 0) VALUE(1)
      .
      .
LOOP:IF (%SST(&NUMBER &X 1) *EQ '0') DO
      CHGVAR (%SST(&NUMBER &X 1)) ' '  /* Blank out */
      CHGVAR &X (&X + 1)  /* Increment */
      IF (&X *NE 4) GOTO LOOP
      ENDDO
```

The following procedure uses the substring built-in function to find the first sentence in a 50-character field &INPUT and to place any remaining text in a field &REMAINDER It assumes that a sentence must have at least 2 characters, and no embedded periods.

```
|                PGM (&INPUT &REMAINDER)   /*  SEARCH  */
|                DCL &INPUT     *CHAR LEN(50)
|                DCL &REMAINDER *CHAR LEN(50)
|                DCL &X *INT  /* INDEX            */
|                DCL &L *INT  /* REMAINING LENGTH */
|
|                DOFORL:
|                DOFOR &X 3 50
|                  IF (%SST(&INPUT &X 1) *EQ '.') THEN(DO)
|                     CHGVAR &L (50-&X)
|                     CHGVAR &X (&X+1)
|                     CHGVAR &REMAINDER %SST(&INPUT &X &L)
|                     LEAVE
|                     ENDDO
|                ENDDO
|           ENDPGM
```

The procedure starts by checking the third position for a period. Note that the substring function checks &INPUT from position 3 to a length of 1, which is position 3 only (length cannot be zero). If position 3 is a period, the remaining length of &INPUT is calculated. The value of &X is advanced to the beginning of the remainder, and the remaining portion of &INPUT is moved to &REMAINDER.

If position 3 is not a period, the procedure checks to see if it is at position 49. If so, it assumes that position 50 is a period and returns. If it is not at position 49, the procedure advances &X to position 4 and repeats the process.

Using the %SWITCH Built-In Function

The switch built-in function (%SWITCH) compares one or more of eight switches with the eight switch settings already established for the job and returns a logical value of '0' or '1'. The initial values of the switches for the job are determined first by the Create Job Description (CRTJOBD) command; the default value is `00000000`. You can change this if necessary using the SWS parameter on the SBMJOB, CHGJOB, or JOB command; the default for these is the job description setting. Other high-level languages may also set job switches.

If, in the comparison of your %SWITCH values against the job values, every switch is the same, a logical value of '1' (true) is returned. If any switch tested does not have the value indicated, the result is a '0' (false).

The syntax of the %SWITCH built-in function is:

```
%SWITCH(8-character-mask)
```

The 8-character mask is used to indicate which job switches are to be tested, and what value each switch is to be tested for. Each position in the mask corresponds with one of the eight job switches in a job. Position 1 corresponds with job switch 1, position 2 with switch 2, and so on. Each position in the mask can be specified as one of three values: 0, 1, or X.

0 The corresponding job switch is to be tested for a 0 (off).

1 The corresponding job switch is to be tested for a 1 (on).

X The corresponding job switch is not to be tested. The value in the switch does not affect the result of %SWITCH.

If %SWITCH(0X111XX0) is specified, job switches 1 and 8 are tested for 0s; switches 3, 4, and 5 are tested for 1s; and switches 2, 6, and 7 are not tested. If each job switch contains the value (1 or 0 only) shown in the mask, the result of %SWITCH is true '1'.

Switches can be tested in a CL procedure to control the flow of the procedure. This function is used in CL procedures with the IF and CHGVAR commands. Switches can be changed in a CL procedure by the Change Job (CHGJOB) command. For CL procedures, these changes take effect immediately.

%SWITCH with the IF Command

On the IF command, %SWITCH can be specified on the COND parameter as the logical expression to be tested. In the following example, 0X111XX0 is compared to the predetermined job switch setting:

```
IF  COND(%SWITCH(0X111XX0))  THEN(GOTO C)
```

If job switches 1, 3, 4, 5, and 8 contain 0, 1, 1, 1, and 0, respectively, the result is true and the procedure branches to the command having the label C. If one or more of the switches tested do not have the values indicated in the mask, the result is false, and the branch does not occur.

In the following example, switches control conditional processing in two procedures.

```
SBMJOB  JOB(APP502) JOBD(PAYROLL) CMD(CALL APP502)
    SWS(11000000)

PGM  /* CONTROL */
IF (%SWITCH(11XXXXXX)) CALLPRC PROCA
IF (%SWITCH(10XXXXXX)) CALLPRC PROCB
IF (%SWITCH(01XXXXXX)) CALLPRC PROCC
IF (%SWITCH(00XXXXXX)) CALLPRC PROCD
ENDPGM

PGM  /* PROCA */
CALLPRC TRANS
IF (%SWITCH(1XXXXXXX)) CALLPRC CUS520
ELSE CALLPRC CUS521
ENDPGM
```

%SWITCH with the CHGVAR Command

On the CHGVAR command, you can specify %SWITCH to change the value of a logical variable. The value of the logical variable is determined by the results of comparing your %SWITCH settings with the job switch settings. If the result of the comparison is true, the logical variable is set to '1'. If the result is false, the variable is set to '0'. For instance, if the job switch is set to 10000001 and this procedure is processed:

```
PGM
DCL &A *LGL
CHGVAR VAR(&A) VALUE(%SWITCH(10000001))
.
.
.
ENDPGM
```

then the variable &A has a value of '1'.

Using the Monitor Message (MONMSG) Command

Escape messages are sent to CL procedures by the commands in the CL procedures and by the programs and procedures they call. These escape messages are sent to

tell the procedures that errors were detected and requested functions were not performed. CL procedures can monitor for the arrival of escape messages, and you can specify through commands how to handle the messages. For example, if a CL procedure tries to move a data area that has been deleted, an object-not-found escape message is sent to the procedure by the Move Object (MOVOBJ) command.

Using the Monitor Message (MONMSG) command, you can direct a procedure to take predetermined action if specific errors occur during the processing of the immediately preceding command. The MONMSG command is used to monitor for escape, notify, or status messages sent to the call stack of the procedure in which the MONMSG command is used. The MONMSG command has the following parameters:

```
MONMSG   MSGID(message-identifier) CMPDTA(comparison-data) +
         EXEC(CL-command)
```

Each message that is sent for a specific error has a unique identifier. You can enter as many as 50 message identifiers on the MSGID parameter. (See the online help for messages and identifiers). The CMPDTA parameter allows even greater specification of error messages because you can check for a specific character string in the MSGDTA portion of the message. On the EXEC parameter, you can specify a CL command (such as a Call Program (CALL), Do (DO), or a Go To (GOTO)), which directs the procedure to perform error recovery.

In the following example, the MONMSG command follows the Receive File (RCVF) command and, therefore, is only monitoring for messages sent by the RCVF command:

```
READLOOP:  RCVF                          /* Read a file record  */
           MONMSG  MSGID(CPF0864)  EXEC(GOTO CMDLBL(EOF))
           /*  Process the file record  */
           GOTO  CMDLBL(READLOOP)       /*  Get another record  */
EOF:       /*  End of file processing  */
```

The escape message, CPF0864, is sent to the procedure's invocation queue when there are no more records in the file to read. Because the example specifies MSGID(CPF0864), the MONMSG monitors for this condition. When it receives the message, the GOTO CMDLBL(EOF) command is run.

You can also use the MONMSG command to monitor for messages sent by any command in a CL procedure. The following example includes two MONMSG commands. The first MONMSG command monitors for the messages CPF0001 and CPF1999; these messages might be sent by any command run later in the procedure. When either message is received from any of the commands running in the procedure, control branches to the command identified by the label EXIT2.

The second MONMSG command monitors for the messages CPF2105 and MCH1211. Because no command is coded for the EXEC parameter, these messages are ignored.

```
PGM
DCL
MONMSG MSGID(CPF0001 CPF1999) EXEC(GOTO EXIT2)
MONMSG MSGID(CPF2105 MCH1211)
.
.
.
ENDPGM
```

Message CPF0001 states that an error was found in the command that is identified in the message itself. Message CPF1999, which can be sent by many of the debugging commands, such as Change Program Variable (CHGPGMVAR), states that errors occurred on the command, but it does not identify the command in the message.

All error conditions monitored for by the MONMSG command with the EXEC parameter specified (CPF0001 or CPF1999) are handled in the same way at EXIT2, and it is not possible to return to the next sequential statement after the error. You can avoid this by monitoring for specific conditions after each command and branching to specific error correction procedures.

All error conditions monitored for by the MONMSG command without the EXEC parameter specified (CPF2105 or MCH1211) are ignored, and procedure processing continues with the next command.

If the error occurs when evaluating the expression on an IF command, the condition is considered false. In the following example, MCH1211 (divide by zero) could occur on the IF command. The condition would be considered false, and PROCA would be called.

```
IF(&A / &B *EQ 5) THEN(DLTF ABC)
ELSE CALLPRC PROCA
```

If you code the MONMSG command at the beginning of your CL procedure, the messages you specify are monitored throughout the program, regardless of which command produces these messages. If the EXEC parameter is used, only the GOTO command can be specified.

You can specify the same message identifier on a procedure-level or a command-level MONMSG command. The command-level MONMSG commands take precedence over the procedure-level MONMSG commands. In the following example, if message CPF0001 is received on CMDB, CMDC is run. If message CPF0001 is received on any other command in the procedure, the procedure branches to EXIT2. If message CPF1999 is received on any command, including CMDB, the procedure branches to EXIT2.

```
PGM
MONMSG MSGID(CPF0001 CPF1999) EXEC(GOTO EXIT2)
CMDA
CMDB
MONMSG MSGID(CPF0001) EXEC(CMDC)
CMDD
EXIT2:  ENDPGM
```

Because many escape messages can be sent to a procedure, you must decide which ones you want to monitor for and handle. Most of these messages are sent to a procedure only if there is an error in the procedure. Others are sent because of conditions outside the procedure. Generally, a CL procedure should monitor for those messages that pertain to its basic function and that it can handle appropriately. For all other messages, OS/400 assumes an error has occurred and takes appropriate default action.

For more information about handling messages in CL procedures, see Chapter 7 and Chapter 8.

Values That Can Be Used as Variables

Retrieving System Values

A system value contains control information for the operation of certain parts of the system. IBM® supplies several types of system values. For example, QDATE and QTIME are date and time system values, which you set when OS/400 is started.

You can bring system values into your procedure and manipulate them as variables using the Retrieve System Value (RTVSYSVAL) command:

```
RTVSYSVAL  SYSVAL(system-value-name)  RTNVAR(CL-variable-name)
```

The RTNVAR parameter specifies the name of the variable in your CL procedure that is to receive the value of the system value.

The type of the variable must match the type of the system value. For character and logical system values, the length of the CL variable must equal the length of the value. For decimal values, the length of the variable must be greater than or equal to the length of the system value. System value attributes are defined in the iSeries Information Center under the **Systems Management** category of information.

System Value QTIME

In the following example, QTIME is received and moved to a variable, which is then compared with another variable.

```
PGM
DCL  VAR(&PWRDNTME)  TYPE(*CHAR)  LEN(6)  VALUE('162500')
DCL  VAR(&TIME)  TYPE(*CHAR)  LEN(6)
RTVSYSVAL  SYSVAL(QTIME)  RTNVAR(&TIME)
IF  (&TIME *GT &PWRDNTME)  THEN(DO)
SNDBRKMSG('Powering down in 5 minutes.  Please sign off.')
PWRDWNSYS  OPTION(*CNTRLD)  DELAY(300)  RESTART(*NO)  +
         IPLSRC(*PANEL)

ENDDO
ENDPGM
```

See the **Systems Management** category of information in the iSeries Information Center for a list of system values and how you can change and display them.

System Value QDATE

In many applications, you may want to use the current date in your procedure by retrieving the system value QDATE and placing it in a variable. You may also want to change the format of that date for use in your procedure. To convert the format of a date in a CL procedure, use the Convert Date (CVTDAT) command.

The format for the system date is the system value QDATFMT. The shipped value of QDATFMT varies according to country or region. For example, `062488` is the MDY (monthdayyear) format for June 24 1988. You can change this format to the YMD, DMY, or the JUL (Julian) format. For Julian, the QDAY value is a 3-character value from 001 to 366. It is used to determine the number of days between two dates. You can also delete the date separators or change the character used as a date separator with the CVTDAT command.

The format for the CVTDAT command is:

```
CVTDAT    DATE(date-to-be-converted) TOVAR(CL-variable) +
          FROMFMT(old-format) TOFMT(new-format) +
          TOSEP(new-separators)
```

The DATE parameter can specify a constant or a variable to be converted. Once the date has been converted, it is placed in the variable named on the TOVAR parameter. In the following example, the date in variable &DATE, which is formatted as MDY, is changed to the DMY format and placed in the variable &CVTDAT.

```
CVTDAT    DATE(&DATE) TOVAR(&CVTDAT) FROMFMT(*MDY) TOFMT(*DMY)
          TOSEP(*SYSVAL)
```

The date separator remains as specified in the system value QDATSEP.

The CVTDAT command can be useful when creating objects or adding a member that uses a date as part of its name. For example, assume that a member must be added to a file using the current system date. Also, assume that the current date is in the MDY format and is to be converted to the Julian format.

```
PGM
DCL &DATE6 *CHAR  LEN(6)
DCL &DATE5  *CHAR  LEN(5)
RTVSYSVAL  QDATE  RTNVAR(&DATE6)
CVTDAT  DATE(&DATE6)  TOVAR(&DATE5)  TOFMT(*JUL)  TOSEP(*NONE)
ADDPFM  LIB1/FILEX  MBR('MBR'  *CAT  &DATE5)
.
.
.
ENDPGM
```

If the current date is 5 January 1988, the added member would be named MBR88005.

Remember the following when converting dates:

- The length of the value in the DATE parameter and the length of the variable on the TOVAR parameter must be compatible with the date format. The length of the variable on the TOVAR parameter must be at least:
-

1. For Non-Julian Dates possessing 2 digit years
   a. Use 6 characters when using no separators.
      `July 28, 1978 would be written as 072878.`
   b. Use 8 characters when using separators.
      `July 28, 1978 would be written as 07-28-78.`
2. For Non-Julian Dates with 4–digit years
   a. Use 8 characters when using no separators.
      `July 28, 1978 would be written as 07281978.`
   b. Use 10 characters when using separators.
      `July 28, 1978 would be written as 07-28-1978.`
3. For Julian dates with 2–digit years
   a. Use 5 characters when using no separators.
      `December 31, 1996 would be written as 96365.`
   b. Use 6 characters when using separators.
      `December 31, 1996 would be written as 96-365.`
4. For Julian dates with 4–digit years,

a. 7 characters are required when no separators are used.
   `February 4, 1997 would be written as 1997035.`
b. 8 characters are required when separators are used.
   `February 4, 1997 would be written as 1997-035.`

Error messages are sent for converted characters that do not fit in the variable. If the converted date is shorter than the variable, it is padded on the right with blanks.

- In every date format except Julian, the month and day are 2-byte fields no matter what value they contain. The year may be either 2-byte or 4-byte fields. All converted values are right-justified and, when necessary, padded with leading zeros.
- In the Julian format, day is a 3-byte field, and year is a 2-byte or 4-byte field. All converted values are right-justified and, when necessary, padded with leading zeros.

The following is an alternative program that uses the ILE bindable API, Get Current Local Time (CEELOCT), to convert a date to Julian format. To create this program, you must use the Create Bound Control Language Program (CRTBNDCL) command alone, or the Create Control Language Module (CRTCLMOD) command and the Create Program (CRTPGM) command together.

```
PGM
DCL  &LILDATE  *INT   LEN(4)
DCL  &PICTSTR  *CHAR  LEN(5)  VALUE(YYDDD)
DCL  &JULDATE  *CHAR  LEN(5)
DCL  &SECONDS  *CHAR  8       /*  Seconds from CEELOCT            */
DCL  &GREG     *CHAR  23      /*  Gregorian date from CEELOCT     */
                              /*                                  */
CALLPRC  PRC(CEELOCT)         /* Get current date and time        */ +
         PARM(&LILDATE        /* Date in Lilian format            */ +
              &SECONDS        /* Seconds field will not be used   */ +
              &GREG           /* Gregorian field will not be used */ +
              *OMIT)          /* Omit feedback parameter          */ +
                              /*  so exceptions are signalled     */

CALLPRC  PRC(CEEDATE) +
         PARM(&LILDATE        /* Today's date     */ +
              &PICTSTR        /* How to format    */ +
              &JULDATE        /* Julian date      */ +
              *OMIT)

ADDPFM   LIB1/FILEX  MBR('MBR'  *CAT   &JULDATE')

ENDPGM
```

See the **Programming** category of information in the iSeries Information Center for more information on ILE APIs.

Retrieving Configuration Source

Using the Retrieve Configuration Source (RTVCFGSRC) command, you can generate CL command source for creating existing configuration objects and place the source in a source file member. The CL command source generated can be used for the following:

- Moving configurations from system to system
- Maintaining on-site configurations
- Saving configurations (without using SAVSYS)

Retrieving Configuration Status

Using the Retrieve Configuration Status (RTVCFGSTS) command, you can give applications the capability to retrieve configuration status from three configuration objects: line, controller, and device. The RTVCFGSTS command can be used in a CL procedure to check the status of a configuration description.

Retrieving Network Attributes

Using the Retrieve Network Attributes (RTVNETA) command, you can retrieve the network attributes of the system. These attributes can be changed using the Change Network Attributes (CHGNETA) command and displayed using the Display Network Attributes (DSPNETA) command. See the **Systems Management** category of information in the iSeries Information Center for more information about network attributes.

RTVNETA Example

In the following example, the default network output queue and the library that contains it are retrieved, changed to QGPL/QPRINT, and later changed back to the previous value.

```
PGM
DCL  VAR(&OUTQNAME) TYPE(*CHAR) LEN(10)
DCL  VAR(&OUTQLIB) TYPE(*CHAR) LEN(10)
RTVNETA  OUTQ(&OUTQNAME) OUTQLIB(&OUTQLIB)
CHGNETA  OUTQ(QGPL/QPRINT)
.
.
.
CHGNETA  OUTQ(&OUTQLIB/&OUTQNAME)
ENDPGM
```

Retrieving Job Attributes

You can retrieve the job attributes and place their values in a CL variable to control your applications.

Job attributes are retrieved using the Retrieve Job Attribute (RTVJOBA) command. You can retrieve all job attributes, or any combination of them, with the RTVJOBA command.

In the following CL procedure, a RTVJOBA command retrieves the name of the user who called the procedure.

```
        PGM
         /* ORD410C Order entry program  */
        DCL &CLKNAM TYPE(*CHAR) LEN(10)
        DCL &NXTPGM TYPE(*CHAR) LEN(3)
        .
        .
        .
        RTVJOBA USER(&CLKNAM)
BEGIN:  CALL ORD410S2 PARM(&NXTPGM &CLKNAM)
         /* Customer prompt */
        IF (&NXTPGM *EQ 'END') THEN(RETURN)
        .
        .
        .
```

The variable &CLKNAM, in which the user name is to be passed, is first declared using a DCL command. The RTVJOBA command follows the declare commands.

When the program ORD410S2 is called, two variables, &NXTPGM and &CLKNAM, are passed to it. &NXTPGM is passed as blanks but could be changed by ORD410S2.

RTVJOBA Example

Assume in the following CL procedure, an interactive job submits a program including the CL procedure to batch. A Retrieve Job Attributes (RTVJOBA) command retrieves the name of the message queue to which the job's completion message is sent, and uses that message queue to communicate with the user who submitted the job.

```
PGM
DCL &MSGQ *CHAR 10
DCL &MSGQLIB *CHAR 10
DCL &MSGKEY *CHAR 4
DCL &REPLY *CHAR 1
DCL &ACCTNO *CHAR 6
.
.
.
RTVJOBA SBMMSGQ(&MSGQ) SBMMSGQLIB(&MSGQLIB)
IF (&MSGQ *EQ '*NONE') THEN(DO)
   CHGVAR &MSGQ 'QSYSOPR'
   CHGVAR &MSGQLIB 'QSYS'
ENDDO
.
.
.
IF (. . . ) THEN(DO)
 SNDMSG:SNDPGMMSG MSG('Account number ' *CAT &ACCTNO *CAT 'is +
                  not valid. Do you want to cancel the update +
                  (Y or N)?') TOMSGQ(&MSGQLIB/&MSGQ) MSGTYPE(*INQ) +
                  KEYVAR(&MSGKEY)
 RCVMSG MSGQ(*PGMQ) MSGTYPE(*RPY) MSGKEY(&MSGKEY) +
          MSG(&REPLY) WAIT(*MAX)
 IF (&REPLY *EQ 'Y') THEN(RETURN)
 ELSE IF (&REPLY *NE 'N') THEN(GOTO SNDMSG)
ENDDO
.
.
.
```

Two variables, &MSGQ and &MSGQLIB, are declared to receive the name and library of the message queue to be used. The RTVJOBA command is used to retrieve the message queue name and library name. Because it is possible that a message queue is not specified for the job, the message queue name is compared to the value *NONE. If the comparison is equal, no message queue is specified, and the variables are changed so that message queue QSYSOPR in library QSYS is used. Later in the procedure, when an error condition is detected, an inquiry message is sent to the specified message queue and the reply is received and processed. Some of the other possible uses of the RTVJOBA command are:

- Retrieve one or more of the job attributes (such as output queue, library list) so that they can be changed temporarily and later restored to their original values.
- Retrieve one or more of the job attributes for use in the SBMJOB command, so that the submitted job will have the same attributes as the submitting job.

Retrieving Object Descriptions

You can also use the Retrieve Object Description (RTVOBJD) command to return the descriptions of a specific object to a CL procedure. Variables are used to return

the descriptions. You can use these descriptions to help you detect unused objects. For more information about retrieving object descriptions, see "Retrieving Object Descriptions" on page 123.

You can also use the Retrieve Object Description (QUSROBJD) application programming interface (API) to return the description of a specific object to a procedure. The system uses a variable to return the descriptions. For more information, see the *APIs* section of the *Programming* category for the **iSeries Information Center**.

Retrieving User Profile Attributes

Using the Retrieve User Profile Attributes (RTVUSRPRF) command, you can retrieve the attributes of a user profile (except for the password) and place their values in CL variables to control your applications. On this command, you can specify either the 10-character user profile name or *CURRENT.

You can also monitor for escape messages after running the RTVUSRPRF command. See the *CL* section of the *Programming* category in the **iSeries Information Center** for more information.

RTVUSRPRF Example

In the following CL procedure, a RTVUSRPRF command retrieves the name of the user who called the procedure and the name of a message queue to which to send messages for that user:

```
DCL  &USR  *CHAR  10
DCL  &USRMSGQ  *CHAR  10
DCL  &USRMSGQLIB  *CHAR  10
.
.
.
RTVUSRPRF  USRPRF(*CURRENT)  RTNUSRPRF(&USR)  +
           MGSQ(&USRMSGQ)  MSGQLIB(&USRMSGQLIB)
```

The following information is returned to the procedure:

- &USR contains the user profile name of the user who called the program.
- &USRMSGQ contains the name of the message queue specified in the user profile.
- &USRMSGQLIB contains the name of the library containing the message queue associated with the user profile.

Retrieving Member Description Information

Using the Retrieve Member Description (RTVMBRD) command, you can retrieve information about a member of a database file for use in your applications.

RTVMBRD Example

In the following CL procedure, a RTVMBRD command retrieves the description of a specific member. Assume a database file called MFILE exists in the current library (MYLIB) and contains 3 members (AMEMBER, BMEMBER, and CMEMBER).

```
DCL  &LIB      TYPE(*CHAR) LEN(10)
DCL  &MBR      TYPE(*CHAR) LEN(10)
DCL  &SYS      TYPE(*CHAR) LEN(4)
DCL  &MTYPE    TYPE(*CHAR) LEN(5)
DCL  &CRTDATE  TYPE(*CHAR) LEN(13)
DCL  &CHGDATE  TYPE(*CHAR) LEN(13)
DCL  &TEXT     TYPE(*CHAR) LEN(50)
DCL  &NBRRCD   TYPE(*DEC)  LEN(10 0)
```

```
DCL  &SIZE     TYPE(*DEC)  LEN(10 0)
DCL  &USEDATE  TYPE(*CHAR) LEN(13)
DCL  &USECNT   TYPE(*DEC)  LEN(5 0)
DCL  &RESET    TYPE(*CHAR) LEN(13)
.
.
.
RTVMBRD    FILE(*CWeb siteIB/MYFILE) MBR(AMEMBER *NEXT) +
           RTNLIB(&LIB) RTNSYSTEM(&SYS) RTNMBR(&MBR) +
           FILEATR(&MTYPE) CRTDATE(&CRTDATE) TEXT(&TEXT) +
           NBRCURRCD(&NBRRCD) DTASPCSIZ(&SIZE) USEDATE(&USEDATE) +
           USECOUNT(&USECNT) RESETDATE(&RESET)
```

The following information is returned to the procedure:

- The current library name (MYLIB) is placed into the CL variable name &LIB
- The system that MYFILE was found on is placed into the CL variable name &SYS (*LCL means the file was found on the local system, and *RMT means the file was found on a remote system.)
- The member name (BMEMBER), since BMEMBER is the member immediately after AMEMBER in a name ordered member list (*NEXT), is placed into the CL variable named &MBR
- The file attribute of MYFILE is placed into the CL variable named &MTYPE (*DATA means the member is a data member, and *SRC means the file is a source member.)
- The creation date of BMEMBER is placed into the CL variable called &CRTDATE
- The text used to describe BMEMBER is placed into the CL variable called &TEXT
- The current number of records in BMEMBER is placed into the CL variable called &NBRRCD
- The size of BMEMBER's data space (in bytes) is placed into the CL variable called &SIZE
- The date that BMEMBER was last used is placed into the CL variable called &USEDATE
- The number of days that BMEMBER has been used is placed into the CL variable called &USECNT The start date of this count is the value placed into the CL variable called &RESET

Working with CL Procedures

A CL source procedure must be compiled into a module and bound into a program before it can be run.

To create a CL program in one step, you can use the Create Bound Control Language Program (CRTBNDCL) command and create a bound program with one module.

You can also create a module with the Create Control Language Module (CRTCLMOD) command. The module must then be bound into a program or service program using the Create Program (CRTPGM) or Create Service Program (CRTSRVPGM) command.

The following example creates the module ORD040C and places it in library DSTPRODLB:

```
CRTCLMOD     MODULE(DSTPRODLB/ORD040C) SRCFILE(QCLSRC)
             TEXT('Order dept general menu program')
```

The source commands for ORD040C are in the source file QCLSRC, and the source member name is ORD040C. By default, a compiler listing is created.

On the Create Bound Control Language Program (CRTBNDCL) command, you can specify listing options and whether the program should operate under the program owner's user profile.

A program can run using either the owner's user profile or the user's user profile.

CL procedures and programs are created using options on the Programming Development Manager (PDM) menu or the Programmer Menu so the Create Control Language Module (CRTCLMOD) command or CRTBNDCL command does not have to be directly entered.

Logging CL Procedure Commands

You can specify that most CL commands run in a CL procedure be written (logged) to the job log by specifying one of the following values on the LOG parameter on the Create Control Language Module (CRTCLMOD) command or the Create Bound Control Language Program (CRTBNDCL) command when the procedure is compiled:

***JOB** This default value indicates that logging is to occur when the job's logging option is on. The option is initially set for no logging, but it can be changed by the LOGCLPGM parameter on the CHGJOB command. Therefore, if you create the module or program with this value, you can alter the logging option for each job or several times within a job.

***YES** This value indicates that logging is to occur each time the CL procedure is run. It cannot be changed by the CHGJOB command.

***NO** This value indicates that no logging is to occur. It cannot be changed by the CHGJOB command.

Because these values are part of the Create Control Language Module (CRTCLMOD) and the Create Bound Control Language Program (CRTBNDCL) commands, you must recompile the module or program to change them.

When you specify logging, you should use the Remove Message (RMVMSG) command with care in order not to remove any logged commands from the job log. If you specify CLEAR(*ALL) on the RMVMSG command, any commands logged prior to running the RMVMSG command do not appear in the job log. This affects only the CL procedure containing the RMVMSG command and does not affect any logged commands for the preceding or following recursion levels.

Not all commands are logged to the job log. Following is a list of commands that are not logged:

CALLPRC	**CHGVAR**	**DCL**
DCLF	**DO**	**DOFOR**
DOUNTIL	**DOWHILE**	**ELSE**
ENDDO	**ENDPGM**	**ENDSELECT**
GOTO	**IF**	**ITERATE**
LEAVE	**MONMSG**	**OTHERWISE**
PGM	**SELECT**	**WHEN**

If the logging option is on, logging messages are sent to the CL procedure's message queue. If the CL procedure is running interactively, and the message level on the job's LOG parameter is set to 4, you can press F10 (Display detail messages) to view the logging of all commands. You can print the log if the message level is 4 and you specify *PRINT when you sign off.

The log includes the time, program and procedure names, message texts, and command names. Command names are qualified as they are on the original source statement. Command parameters are also logged; if the parameter information is a CL variable, the contents of the variable are printed (except for the RTNVAL parameter).

Logging of commands affects performance.

CL Module Compiler Listings

When you create a CL module, you can create various types of listings using the OPTION and OUTPUT parameters on the Create Control Language Module (CRTCLMOD) command.

The OPTION parameter values and their meanings are:

- *GEN or *NOGEN

  Whether a module is to be created (*GEN is the default).
- *XREF or *NOXREF

  Whether a listing of cross-references to variables and data references in the source input is to be produced (*XREF is the default).

The OUTPUT parameter values and their meanings are:

- *PRINT - print listing
- *NONE - no compiler listing

The listing created by specifying the OUTPUT parameter is called a *compiler listing*. The following shows a sample compiler listing. The callout numbers refer to descriptions following the listing.

```
                                                                                                  1                                  2                    3
5722SS1 V5R3M0  041231                Control Language        MYLIB/DUMPER         SYSNAME  05/06/00 11:12:55       Page 1
 Module  . . . . . . . . . . . . . . . . . . . :   DUMPERR
   Library . . . . . . . . . . . . . . . . . . :     MYLIB
 Source file . . . . . . . . . . . . . . . . . :   QCLSRC
   Library . . . . . . . . . . . . . . . . . . :     MYLIB
 Source member name  . . . . . . . . . . . . . :   DUMPERR   05/06/94 10:42:26  4
 Source printing options . . . . . . . . . . . :   *XREF  *NOSECLVL  *NOEVENTF
 Module logging  . . . . . . . . . . . . . . . :   *JOB
 Replace module object . . . . . . . . . . . . :   *YES
 Target release  . . . . . . . . . . . . . . . :   V5R3M0
 Authority . . . . . . . . . . . . . . . . . . :   *LIBCRTAUT
 Sort sequence . . . . . . . . . . . . . . . . :   *HEX
 Language identifier . . . . . . . . . . . . . :   *JOBRUN
 Text  . . . . . . . . . . . . . . . . . . . . :   Test program
 Optimization  . . . . . . . . . . . . . . . . :   *NONE
 Debugging view  . . . . . . . . . . . . . . . :   *STMT
 Enable performance collection . . . . . . . . :   *PEP
 Compiler  . . . . . . . . . . . . . . . . . . :   IBM iSeries Control Language Compiler  5
 6                      Control Language Source
 SEQNBR  *...+... 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8 ...+... 9 ...+.  DATE   8
   100- PGM                                                                                                  05/06/94
    200- DCL &ABC *CHAR 10 VALUE('THIS')                                                                     05/06/94
    300- DCL &XYZ *CHAR 10 VALUE('THAT')  7                                                                  05/06/94
    400- DCL &MNO *CHAR 10 VALUE('OTHER')                                                                    05/06/94
    500- CRTLIB LB(LARRY)                                                                                    05/06/94
* CPD0043 30  Keyword LB not valid for this command.  9
   600- DLTLIB LIB(MOE                                                                                       05/06/94
* CPD0013 30  A matching parenthesis not found.
    700- MONMSG CPF0000 EXEC(GOTO ERR)                                                                       05/06/94
    800- ERROR:                                                                                              05/06/94
    900- CHGVAR &ABC 'ONE'                                                                                   05/06/94
   1000- CHGVAR &XYZ 'TWO'                                                                                   05/06/94
   1100- CHGVAR &MNO 'THREE'                                                                                 05/06/94
   1200- DMPCLPGM                                                                                            05/06/94
   1300- ENDPGM                                                                                              05/06/94
                              * * * * *  E N D  O F  S O U R C E  * * * * *
 5722SS1 V5R3M0  040201                Control Language        MYLIB/DUMPER         SYSNAME  05/06/00 11:12:55       Page 2
                                        Cross Reference
```

```
 Declared Variables
 Name             Defined     Type            Length      References
 &ABC                200      *CHAR             10           900
 &MNO                400      *CHAR             10          1100
 10
&XYZ                 300      *CHAR             10          1000
 Defined Labels
 Label            Defined     References  11
ERR                ******         700
* CPD0715 30  Label 'ERR       ' does not exist.
 ERROR               800
                     * * * * *  E N D  O F  C R O S S  R E F E R E N C E  * * * * *
 5722SS1 V5R3M0  040201               Control Language         MYLIB/DUMPER          SYSNAME  05/06/04 11:12:55       Page 3
                                       Message Summary
              Severity
 Total        0-9  10-19  20-29  30-39  40-49  50-59  60-69  70-79  80-89  90-99  12
     3          0      0      0      3      0      0      0      0      0      0
 Module DUMPERR not created in library MYLIB. Maximum error severity 30.  13
                              * * * * *  E N D  O F  M E S S A G E  S U M M A R Y  * * * * *
                                * * * * *  E N D  O F  C O M P I L A T I O N  * * * * *
```

Title:

1 The program number, release, modification level and date of OS/400.

2 The date and time of the compiler run.

3 The page number in the listing.

Prolog:

4 The parameter values specified (or defaults if not specified) on the Create Control Language Module (CRTCLMOD) command. If the source is not in a database file, the member name, date, and time are omitted.

5 The name of the compiler.

Source:

6 The sequence numbers of lines (records) in the source. A dash following a sequence number indicates that a source statement begins at that sequence number. The absence of a dash indicates that a statement is the continuation of the previous statement.

Comments between source statements are handled like any other source statement and have sequence numbers.

7 The source statements.

8 The last date the source statement was changed or added. If the source is not in a database file, or the dates have been reset using RGZPFM, the date is omitted.

9 If an error is found during compilation and can be traced to a specific source statement, the error message is printed immediately following the source statement. An asterisk (*) indicates the line contains an error message. The line contains the message identifier, severity, and the text of the message.

For more information about compilation errors, see “Errors Encountered during Compilation” on page 63.

Cross-References:

10 The symbolic variable table is a cross-reference listing of the variables validly declared in the program. The table lists the variable, the sequence number of the statement where the variable is declared, the variable’s attributes, and the sequence numbers of statements that refer to the variable.

11 The label table is a cross-reference listing of the labels validly defined in

the program. The table lists the label, the sequence number of the statement where the label is defined, and the sequence numbers of statements that refer to the label.

Messages:

This section is not included in the sample listing because no general error messages were issued for the sample module. If there were general error messages for this module, this section would contain, for each message, the message identifier, the severity, and the message.

Message Summary:

12 A summary of the number of messages issued during compilation. The total number is given along with totals by severity.

13 A completion message is printed following the message summary.

The title, prologue, source, and message summary sections are always printed for the *SOURCE option. The cross-reference section is printed if the *XREF option is specified. The message section is printed only if general errors are found.

Errors Encountered during Compilation

In the compiler listing of a module, an error condition that relates directly to a specific command is listed after that command. See "CL Module Compiler Listings" on page 61 for an example of these inline messages. Messages that do not relate to a specific command but are more general in nature are listed in a messages section of the listing, not inline with source statements.

The types of errors that are detected at compile time include syntax errors, references to variables and labels not defined, and missing statements. The following types of errors stop the procedure from being created (severity codes are ignored).

- Value errors
- Syntax errors
- Errors related to dependencies between parameters within a command
- Errors detected during validity checking.

Even after an error that stops the procedure from being created is encountered, the compiler continues to check the source for errors. This lets you see and correct as many errors as possible before you try to create the module or program again.

Obtaining a Procedure Dump

You can obtain a CL procedure dump during procedure processing. The CL procedure dump consists of a listing of all messages on the procedure's message queue and the values of all variables used in the procedure. This information may be useful in determining the cause of a problem affecting procedure processing.

To obtain a CL procedure dump, do one of the following:

- Run the Dump CL Program (DMPCLPGM) command. This command can only be used in a CL procedure and does not end the CL procedure.
- Enter D in response to inquiry message CPA0701 or CPA0702. The system sends this message whenever it receives an unmonitored escape message from a CL procedure. If the program is running in an interactive job, the system sends the

message to the job's external message queue. If the program is running as a batch job, the system sends the message to the system operator message queue, QSYSOPR.

- Specify INQMSGRPY(*SYSRPYL) for the job. See the **Systems Management** category of information in the iSeries Information Center for a description of this job attribute. The IBM-supplied system reply list specifies a reply of D for message CPA0702 or CPA0701. The system will print a dump if it receives one of the inquiry messages.
- Change the default reply for message CPA0701 or CPA0702 from C (cancel program) to D (dump procedure). This prints a procedure dump whenever a function check occurs in a CL procedure. To change the default, enter the following command:

  ```
  CHGMSGD MSGID(CPA0702) MSGF(QCPFMSG) DFT(D)
  ```

  Note: The security officer, or another user with update authority to the QCPFMSG file, must enter the CHGMSGD command.

Changing the message default causes a dump to be printed under any of the following conditions:

- The system operator message queue is in default mode and the message is sent from a batch job.
- The display station user presses the Enter key without typing a response, causing the message default to be used.
- INQMSGRPY(*DFT) is specified for the job.

```
                                1
5722SS1 V5R3M0  040201                Control Language        MYLIB/DUMPER         SYSNAME  05/06/00  11:05:03 2 Page 1

 Job name . . . . . :  DSP04 3  User name . . . . . :  SMITH 3  Job number  . . . . . : 01329 3
 Program name . . . . . :  DUMP 4  Library . . . . . :  MYLIB 4  Statement . . . . . : 1200 5
 Module name  . . . . . :  DUMP    Procedure name  . . . :  DUMP

                                                Messages

           Message 6                 Message                       From                      To
 Time      ID                  Sev   Type    Text                  Program         Inst      Program         Inst
 110503    CPC2102             00    COMP    Library LARRY created.  QLICRLIB       *N        DUMP            *N
 110503    CPF2110             40    ESC     Library MOE not found.  QLICLLIB       *N        DUMP            *N

                                              Variables 7

 Variable          Type       Length          Value                        Value in Hexadecimal
                                              *...+....1....+....2....+    * . . . + . . . . 1 . . . . + . . . . 2 . . . . +
 &ABC              *CHAR              10      'ONE       '                 D6D5C540404040404040
 &XYZ              *CHAR              10      'TWO       '                 E3E6D640404040404040

                                    * * * * *  E N D  O F  D U M P  * * * * *
```

1 The program number, release, modification level and date of OS/400.

2 The date and time the dump was printed.

3 The fully qualified name of the job in which the procedure was running.

4 The name and library of the program.

5 The number of the statement running when the dump was taken. If the command is a nested command, the statement number is that of the outer command.

6 Each message on the call message queue, including the time the message was sent, message ID, severity, type, text, sending program and instruction number, and receiving program and instruction number.

7 All variables declared in the procedure, including variable name, type, length, value, and hexadecimal value.

If a decimal variable contains decimal data that is not valid, the character and hexadecimal values are printed as *CHAR variables.

If the value for the variable cannot be located, *NOT ADDRESSABLE is printed. This can occur if the CL procedure is used in a command processing program for a command that has a parameter with either TYPE(*NULL) or PASSVAL(*NULL) specified, or if RTNVAL(*YES) was specified for the parameter and a return variable is not coded on the command.

If a variable is declared as TYPE(*LGL), it is shown on the dump as *CHAR with a length of 1.

Displaying Module Attributes

You can use the Display Module (DSPMOD) command to display the attributes of a module. The information displayed or printed can be used to determine the options specified on the command used to create the module.

For more information on this command, see the *CL* section of the *Programming* category in the **iSeries Information Center**.

Displaying Program Attributes

You can use the Display Program (DSPPGM) command to display the attributes of a program. The information displayed or printed can be used to determine the options specified on the command used to create the program.

For more information on this command, see the *CL* section of the *Programming* category in the **iSeries Information Center**.

Return Code Summary

The return code (RTNCDE) parameter on the RTVJOBA command is a 5-digit decimal value with no decimal positions (12345. for example). The decimal value indicates the status of called programs. CL programs do not set the return code. However, you can retrieve the current value of the return code as set by another program in a CL program. You can do this by using the RTNCDE parameter of the RTVJOBA command.

The following list summarizes the return codes used by languages supported on OS/400:

- RPG IV programs

  The return codes sent by the RPG IV compiler are:

  0 When the program is created

  2 When the program is not created

  The return codes sent by running RPG IV programs are:

  0 When a program is started, or by the CALL operation before a program is called

  1 When a program ends with LR set on

  2 When a program ends with an error (response of C, D, F, or S to an inquiry message)

  3 When a program ends because of a halt indicator (H1-H9)

  RPG IV return codes are tested only after a CALL:

  – 0 or 1 indicate no error

– 3 gives an RPG IV status code of 231
– Any other value gives an RPG IV status code 202 (call ended in error)

The return code cannot be tested directly by the user in the RPG IV program.

- ILE COBOL/400® programs

  The return codes sent by running COBOL/400 programs are:

  0 By each CALL statement before a program is called

  2 When a program receives a function check (CPF9999) or the generic I/O exception handler gets control and there is no applicable USE procedure

  COBOL/400 programs cannot retrieve these return codes. A return code value of 2 sends message CBE9001 and runs a Reclaim Resources (RCLRSC) command with the *ABNORMAL option.

- C/400* programs

  The current value of the integer return code returned by the last C/400® return statement in a C/400 program.

Compiling Source Programs for a Previous Release

The Create Control Language Program (CRTCLPGM) command allows you to compile CL source programs to use on a previous release by using the target release (TGTRLS) parameter. The TGTRLS parameter specifies on which release of the OS/400 licensed program the CL program object created intends to run. You can specify *CURRENT, *PRV, or a specific release level.

A CL program compiled with TGTRLS(*CURRENT) runs only on the current release or later releases of the operating system. A CL program compiled with a specified TGTRLS value other than *CURRENT can run on the specified release value and on later releases.

Previous-Release (*PRV) Libraries

The CL compiler retrieves information about previous-release commands and files from CL previous-release (*PRV) libraries. Two types of libraries contain previous-release support: system libraries and user libraries. The libraries have the names QSYSVxRxMx and QUSRVxRxMx. (VxRxMx represents the version, release, and modification level of the supported previous release). For example, the QUSRV4R5M0 library supports a system that runs Version 4 Release 5 Modification level 0 of the OS/400 licensed program.

When the CL compiler compiles for a supported previous release, it first checks for commands and files in the previous-release libraries. When failing to find the command or file in the previous-release libraries, the system performs a search of the library list (*LIBL) or the qualified library.

QSYSVxRxMx Libraries: The QSYSVxRxMx libraries install at the same time as the CL compiler support for a previous release installs. The QSYSVxRxMx libraries include the command definition objects and output files (*OUTFILE) that are found in library QSYS for that particular previous release.

QUSRVxRxMx Libraries: You can create your own QUSRVxRxMx libraries to hold copies of your commands and files as they existed in the supported previous release. This is especially important if the commands or files have changed on the current release.

When the compiler looks for previous-release commands and files, it checks the QUSRVxRxMx library (if it exists) before checking the QSYSVxRxMx library.

Note: Use the QUSRVxRxMx libraries to hold previous-release user commands and files, instead of the QSYSVxRxMx libraries. When installing future releases of the CL compiler, support for previous releases install as well. Once the previous-release support is installed, the QUSRVxRxMx libraries for releases that are no longer supported can be deleted.

Do not add previous-release libraries to the library list (*LIBL). They contain commands and files that support earlier releases and cannot run on the current system. Only the CL compiler refers to and uses the commands and files in the previous-release libraries. The system commands that are supplied for a previous release are in the primary language for the system. There are no secondary national language versions available.

See the appropriate CL Reference command, (Save Object (SAVOBJ), Save Changed Object (SAVCHGOBJ), or Save Library (SAVLIB)), for how to save objects on a different release.

Note: CL programs that are compiled in the System/38™ environment cannot be saved for a previous release.

Installing CL Compiler Support for a Previous Release

To install the *PRV CL compiler support and QSYSVxRxMx libraries:

1. Enter:

   ```
   GO LICPGM
   ```

   To view the Licensed Program Menu.
2. Select option 11 (Install licensed programs).
3. Select the option that is named *PRV CL Compiler Support. This causes the QSYSVxRxMx libraries to install.

   If you are not using the CL compiler support for a previous release, you can remove this support by entering:

   ```
   DLTLICPGM LICPGM(5769SS1) OPTION(9)
   ```

   When the CL compiler support is removed, the QSYSVxRxMx libraries get removed from the system, but the QUSRVxRxMx libraries do not. If no need exists for the QUSRVxRxMx libraries, you must explicitly delete them using the Delete Library (DLTLIB) command.

Chapter 3. Controlling Flow and Communicating between Programs and Procedures

You can use the Call Program (CALL), Call Bound Procedure (CALLPRC), and Return (RETURN) commands to pass control back and forth between programs and procedures. Each command has slightly different characteristics. Information may be passed to called programs and procedures as parameters when control is passed.

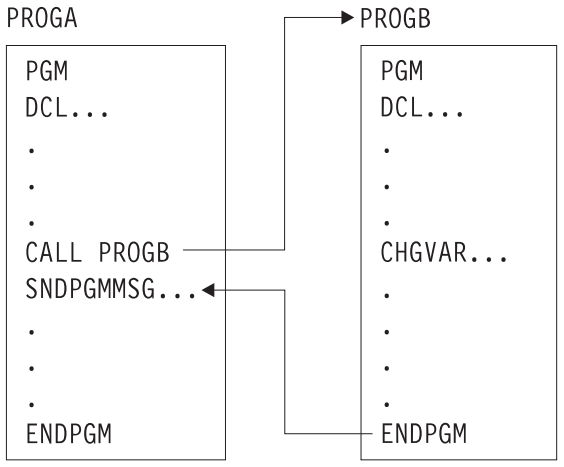
Special attention should be given to programs created with USRPRF(*OWNER) that run CALL or CALLPRC commands. Security characteristics of these commands differ when they are processed in programs running under an owner's user profile. See the Security - Reference book for more information about user profiles.

CALL Command

The Call Program (CALL) command calls a program named on the command, and passes control to it. The CALL command has the following format:

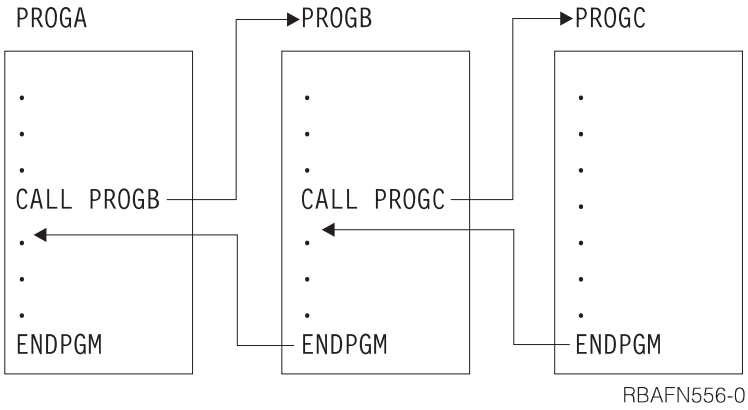
```
CALL PGM(library-name/program-name) PARM(parameter-values)
```

The program name or library name may be a variable. If the called program is in a library that is not on the library list, you must specify the qualified name of the program on the PGM parameter. The PARM parameter is discussed under "Passing Parameters between Programs and Procedures" on page 72. When the called program finishes running, control returns to the next command in the calling program.



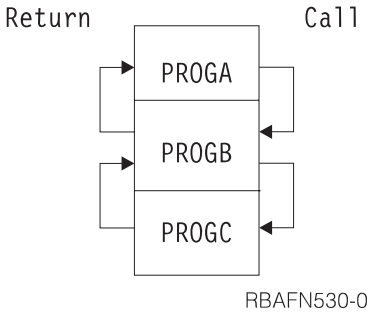
© Copyright IBM Corp. 1997, 2004

The sequence of CALL commands in a set of programs calling each other is the call stack. For example, in this series:



RBAFN556-0

the call stack is:



RBAFN530-0

When PROGC finishes processing, control returns to PROGB at the command after the call to PROGC. Control is thus returned up the call stack. This occurs whether or not PROGC ends with a RETURN or an ENDPGM command.

A CL program can call itself.

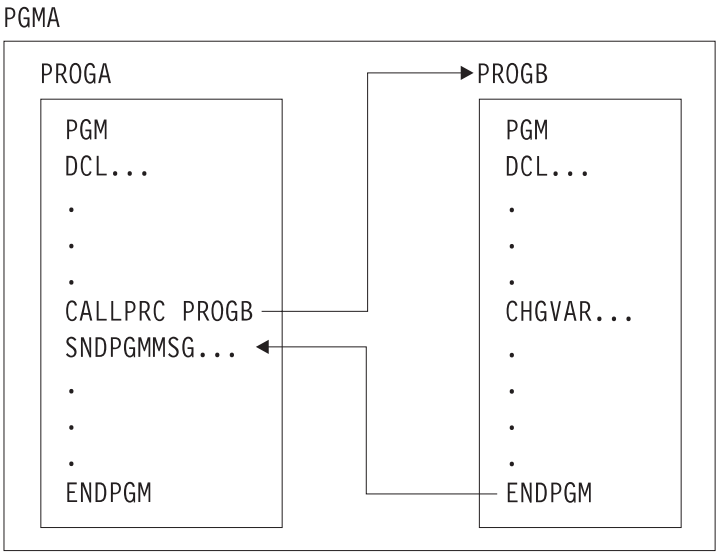
CALLPRC Command

The Call Bound Procedure (CALLPRC) command calls a procedure named on the command, and passes control to it. The CALLPRC command has the following format:

```
CALLPRC PRC(procedure-name) PARM(parameter-values) RTNVAL(return-value-variable)
```

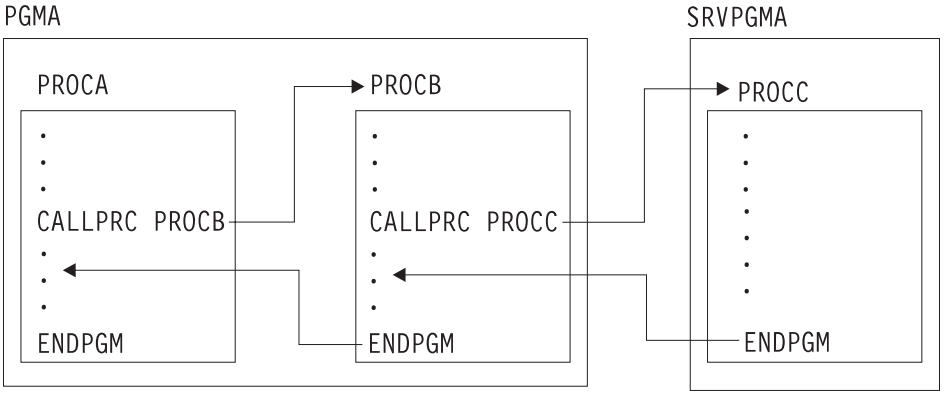
The procedure name may not be a variable. The PARM parameter is discussed under “Passing Parameters between Programs and Procedures” on page 72. When the called procedure finishes running, control returns to the next command in the

calling procedure.



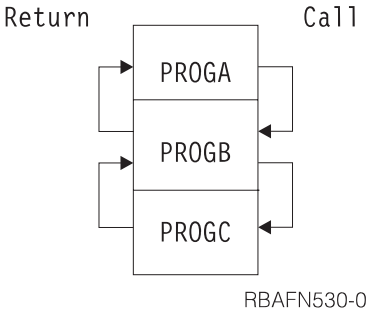
RBAFN539-0

The sequence of CALLPRC commands in a set of procedures calling each other is the call stack. For example, in this series:



RBAGN540-0

the call stack is:



RBAFN530-0

When PROGC finishes processing, control returns to PROGB at the command after the call to PROGC. Control is thus returned up the call stack. This occurs whether or not PROGC ends with a RETURN or an ENDPGM command.

A CL procedure can call itself.

RETURN Command

The Return (RETURN) command in a CL procedure or OPM program removes that procedure or OPM program from the call stack.

If the procedure containing the RETURN command was called by a CALLPRC command, control is returned to the next sequential statement after that CALLPRC command in the calling program.

If a MONMSG command specifies an action that ends with a RETURN command, control is returned to the next sequential statement after the statement that called the procedure or program containing the MONMSG command.

The RETURN command has no parameters.

Note: If you have a RETURN command in an initial program, the command entry display is shown. You may wish to avoid this for security reasons.

Passing Parameters between Programs and Procedures

When you pass control to another program or procedure, you can also pass information to it for modification or use within the receiving program or procedure. See the discussion of this under "Using the CALL Command" on page 75.. You can specify the information to be passed on the PARM parameter on the CALL command or the CALLPRC command. The characteristics and requirements for these commands are slightly different.

For instance, if PROGA contains the following command:

```
CALL PROGB PARM(&AREA)
```

then it calls PROGB and passes the value of &AREA to it. PROGB must start with the PGM command, which also must specify the parameter it is to receive:

```
PGM PARM(&AREA) /* PROGB */
```

For the CALL command or the CALLPRC command, you must specify the parameters passed on the PARM parameter, and you must specify them on the PARM parameter of the PGM command in the receiving program or procedure. Because parameters are passed by position, not name, the position of the value passed in the CALL command or the CALLPRC command must be the same as its position on the receiving PGM command. For example, if PROGA contains the following command:

```
CALL PROGB PARM(&A &B &C ABC)
```

it passes three variables and a character string, and if PROGB starts with:

```
PGM PARM(&C &B &A &D) /*PROGB*/
```

then the value of &A in PROGA is used for &C in PROGB, and so on; &D in PROGB is ABC. The order of the DCL statements in PROGB is unimportant. Only the order in which the parameters are specified on the PGM statement determines what variables are passed.

In addition to the position of the parameters, you must pay careful attention to their length and type. Parameters listed in the receiving procedure or program must be declared as the same length and type as they are in the calling procedure or program. Decimal constants are always passed with a length of (15 5).

When you use the CALLPRC command and pass character string constants, you must specify the exact number of bytes, and pass exactly that number. The called procedure can use the information in the operational descriptor to determine the exact number of bytes passed. You can use the API CEEDOD to access the operational descriptor. See the *APIs* section of the *Programming* category for the **iSeries Information Center** for information on the API CEEDOD.

When you use the CALL command, character string constants of 32 bytes or less are always passed with a length of 32 bytes. If the string is longer than 32, you must specify the exact number of bytes, and pass exactly that number.

The following is an example of a procedure or program that receives the value &VAR1:

```
PGM PARM(&VAR1)  /*PGMA*/
DCL VAR1 *CHAR LEN(36)
.
.
.
ENDPGM
```

The CALL command or CALLPRC command must specify 36 characters:

```
CALLPRC PGMA(ABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJ)
```

The following example specifies the default lengths:

```
PGM PARM(&P1 &P2)
DCL VAR(&P1) TYPE(*CHAR) LEN(32)
DCL VAR(&P2) TYPE(*DEC) LEN(15 5)
IF (&P1 *EQ DATA) THEN(CALL MYPROG &P2)
ENDPGM
```

To call this program, you could specify:

```
CALL PROG (DATA 136)
```

The character string DATA is passed to &P1; the decimal value 136 is passed to &P2

Referring to locally defined variables incurs less overhead than referring to passed variables. Therefore, if the called procedure or program frequently refers to passed variables, performance can be improved by copying the passed values into a local variable and referring to the locally defined value rather than the passed value.

When calling an OPM CL program, the number of parameters that are passed to it must exactly match the number that is expected by the program. The number that is expected is determined at the time the program is created. (The operating system prevents you from calling a program with more or fewer parameters than the program expects). When calling an ILE program or procedure, the operating system does not check the number of parameters that are passed on the call. In addition, the space where the operating system stores the parameters is not reinitialized between procedure calls. Calling a procedure that expects ″n″ parameters with ″n-1″ parameters makes the system use whatever is in the parameter space to access the ″nth″ parameter. The results of this action are very unpredictable. This also applies to procedures written in other ILE languages that call CL procedures or are called by CL procedures.

This also gives you more flexibility when you write ILE CL procedures, because you can write procedures that have variable length parameter lists. For example, based on the value of one parameter, a parameter that is specified later in the list

may not be required. If the controlling parameter indicated an unspecified optional parameter, the called procedure should not attempt to refer to the optional parameter.

You can also specify the special value *OMIT for any parameter that you want to omit from the parameter list. If you specify *OMIT for a parameter, the calling procedure passes a null pointer. The procedure that is called has to be prepared to handle a null pointer if it refers to a parameter that is omitted. In control language (CL), you can check for a null pointer by monitoring for MCH3601 on the first reference to the omittable parameter. The procedure must take appopriate action if it receives a MCH3601.

When calling procedures, you can pass arguments by reference and by value. For more information about passing by value, see the Calls to Procedures and Programs chapter in the ILE Concepts book.

The following example has two CL procedures. The first procedure expects one parameter; if that parameter remains unspecified, results will be unpredictable. The first procedure calls another procedure, PROC1. PROC1 expects one or two parameters. If the value of the first parameter is '1', it expects the second parameter as specified. If the value of the second parameter is '0', it assumes that the second parameter remained unspecified and used a default value instead. PROC1 also uses the CEEDOD API to determine the actual length that is passed for the second parameter.

```
MAIN: PGM   PARM(&TEXT)/* &TEXT must be specified.  Results will be +
                unpredictable if it is omitted.*/
      DCL   VAR(&TEXT) TYPE(*CHAR) LEN(10)
      CALLPRC   PRC(PROC1) PARM('0')
      CALLPRC   PRC(PROC1) PARM('1' &TEXT)
      CALLPRC   PRC(PROC1) PARM('1' 'Goodbye')
      ENDPGM

PROC1: PGM   PARM(&P1 &P2) /* PROC1 - Procedure with optional +
           parameter &P2 */
     DCL  VAR(&P1) TYPE(*LGL) /*Flag which indicates +
         whether or not &P2 will be specified. If +
         value is '1', then &P2 is specified */
     DCL   VAR(&P2) TYPE(*CHAR) LEN(10)
     DCL   VAR(&MSG) TYPE(*CHAR) LEN(10)
     DCL   VAR(&PARMPOS) TYPE(*CHAR) LEN(4) /* +
          Parameter position for CEEDOD*/
     DCL   VAR(&PARMDESC) TYPE(*CHAR) LEN(4) /* +
          Parameter description for CEEDOD*/
     DCL   VAR(&PARMTYPE) TYPE(*CHAR) LEN(4) /* +
          Parameter datatype from CEEDOD*/
     DCL   VAR(&PARMINFO1) TYPE(*CHAR) LEN(4) /* +
          Parameter information from CEEDOD */
     DCL   VAR(&PARMINFO2) TYPE(*CHAR) LEN(4) /* +
          Parameter information from CEEDOD */
     DCL   VAR(&PARMLEN) TYPE(*CHAR) LEN(4) /* +
          Parameter length from CEEDOD*/
     DCL   VAR(&PARMLEND) TYPE(*DEC) LEN(3 0) /* +
          Decimal form of parameter length*/
     IF   COND(&P1) THEN(DO) /* Parm 2 is+
          specified, so use the parm value for the +
          message text*/
     CHGVAR  VAR(%BIN(&PARMPOS 1 4)) VALUE(2) /* Tell +
          CEEDOD that we want the operational +
          descriptor for the second parameter*/
     CALLPRC  PRC(CEEDOD) PARM(&PARMPOS &PARMDESC +
          &PARMTYPE &PARMINFO1 &PARMINFO2 &PARMLEN) +
          /* Call CEEDOD to get the length of data +
          specified for &P2*/
      CHGVAR  VAR(&PARMLEND) VALUE(%BIN(&PARMLEN 1 4)) /* +
              Convert the length returned by CEEDOD to +
              decimal format*/
      CHGVAR  VAR(&MSG) VALUE(%SST(&P2 1 &PARMLEND)) /* +
            Copy the data passed in to a local variable*/
      ENDO
      ELSE   CMD(CHGVAR VAR(%MSG) VALUE('Hello')) /* Use +
             "Hello" for the message text*/
      SNDPGMMSG MSG(&MSG)
      ENDPGM
```

Using the CALL Command

When the CALL command is issued by a CL procedure, each parameter value passed to the called program can be a character string constant, a numeric constant, a logical constant, or a CL variable. A maximum of 255 parameters can be passed to the called program. The values of the parameters are passed in the order in which they appear on the CALL command, and this must match the order in which they appear in the parameter list of the called program. The names of the variables passed do not have to be the same as the names on the receiving parameter list. The names of the variables receiving the values in the called program must be declared to the called program, but the order of the declare commands is not important.

No association exists between the storage in the called program and the variables it receives. Instead, when the calling program passes a variable, the storage for the variable is in the program in which it was originally declared. The system passes variables by address. When passing a constant, the calling program makes a copy of the constant, and passes the address of that copy to the called program.

The result is that when a variable is passed, the called program can change the value of the variable, and the change is reflected in the calling program. The new value does not have to be returned to the calling program for later use; it is already there. Thus no special coding is needed for a variable that is to be returned to the calling program. When a constant is passed, and its value is changed by the called program, the changed value is not known to the calling program. Therefore, if the calling program calls the same program again, it reinitializes the values of constants, but not of variables.

An exception to the previous description is when the CALL command calls an ILE C program. When using the CALL command to call an ILE C program and pass character or logical constants, the system adds a null character (x'00') after the last non-blank character. If the constant is a character string that is enclosed in apostrophes or a hexadecimal constant, the null character is added after the last character that was specified. This preserves the trailing blanks (x '40' characters). Numeric values are not null-terminated.

If a CL program might be called using a CALL command that has not been compiled (an interactive CALL command or through the SBMJOB command), the decimal parameters (*DEC) should be declared with `LEN(15 5)`, and the character parameters (*CHAR) should be declared `LEN(32)` or less in the receiving program.

A CALL command that is not in a CL procedure or program cannot pass variables as arguments. Be careful when specifying the CALL command as a command parameter that is defined as TYPE(*CMDSTR). This converts the contents of any variables that are specified on the PARM parameter to constants. The command (CMD) parameters on the Submit Job (SBMJOB) command, Add Job Schedule Entry (ADDJOBSCDE) command, or Change Job Schedule Entry (CHGJOBSCDE) command are examples. The online help information for the Call (CALL) command describes how to pass parameters when using the CALL command interactively. Refer to the command help or to the command documentation in the *CL* section of the **Programming** category in the **iSeries Information Center**.

Parameters can be passed and received as follows:

- Character string constants of 32 bytes or less are *always* passed with a length of 32 bytes (padded on the right with blanks). If a character constant is longer than 32 bytes, the entire length of the constant is passed. If the parameter is defined to contain more than 32 bytes, the CALL command must pass a constant containing exactly that number of bytes. Constants longer than 32 characters are ***not*** padded to the length expected by the receiving program.

  The receiving program can receive less than the number of bytes passed. For example, if a program specifies that 4 characters are to be received and `ABCDEF` is passed (padded with blanks in 26 positions), only `ABCD` are accepted and used by the program.

  If the receiving program receives more than the number of bytes passed, the results may be unexpected. Numeric values passed as characters must be enclosed in apostrophes.
- Decimal constants are passed in packed form and with a length of `LEN(15 5)`, where the value is 15 digits long, of which 5 digits are decimal positions. Thus,

if a parameter of 12345 is passed, the receiving program must declare the decimal field with a length of LEN(15 5); the parameter is received as 12345.00000.

If you need to pass a numeric constant to a program and the program is expecting a value with a length and precision other than 15 5, the constant can be coded in hexadecimal format. The following CALL command shows how to pass the value 25.5 to a program variable that is declared as LEN(5 2):

```
CALL PGMA PARM(X'02550F')
```

- Logical constants are passed with a length of 32 bytes. The logical value 0 or 1 is in the first byte, and the remaining bytes are blank. If a value other than 0 or 1 is passed to a program that expects a logical value, the results may be unexpected.
- A floating point literal or floating point special value (*NAN, *INF, or *NEGINF) is passed as a double precision value, which occupies 8 bytes. Although a CL program cannot process floating point numbers, it can receive a floating point value into a character variable and pass that variable to an HLL program that can process floating point values.
- The system can pass a variable if the call is made from a CL procedure or program. In this case the receiving program should declare the field to match the variable that is defined in the calling CL procedure or program. For example, assume that a CL procedure or program defines a decimal variable that is named &CHKNUM as LEN(5 0). Then the receiving program should declare the field as packed with 5 digits total, with no decimal positions. When running a CALL command in batch mode by using the SBMJOB command in a CL procedure or program, the system treats any variables that are passed as arguments like constants.
- If either a decimal constant or a program variable can be passed to the called program, the parameter should be defined as LEN(15 5), and any calling program must adhere to that definition. If the type, number, order, and length of the parameters do not match between the calling and receiving programs (other than the length exception noted previously for character constants), results cannot be predicted.
- In order for an integer constant to be passed to an integer variable, the constant must be specified in hexadecimal format. The size of the hexadecimal constant and integer variable must be the same.
- The value *N cannot be used to specify a null value because a null value cannot be passed to another program.

In the following example, program A passes six parameters: one logical constant, three variables, one character constant, and one numeric constant.

```
PGM /* PROGRAM A */
DCL VAR(&B) TYPE(*CHAR)
DCL VAR(&C) TYPE(*DEC) LEN(15 5) VALUE(13.529)
DCL VAR(&D) TYPE(*CHAR) VALUE('1234.56')
CHGVAR VAR(&B) VALUE(ABCDEF)
CALL PGM(B) PARM('1' &B &C &D XYZ 2)  /* Note blanks between parms */
.
.
.
ENDPGM

PGM PARM(&A &B &C &W &V &U) /* PROGRAM B */
DCL VAR(&A) TYPE(*LGL)
DCL VAR(&B) TYPE(*CHAR) LEN(4)
DCL VAR(&C) TYPE(*DEC)
            /* Default length (15 5) matches DCL LEN in program A */
DCL VAR(&W) TYPE(*CHAR)
```

```
DCL VAR(&V) TYPE(*CHAR)
DCL VAR(&U) TYPE(*DEC)
.
.
. ENDPGM
```

Note: If the fifth parameter passed to PGMB was `456` instead of `XYZ` and was intended as alphanumeric data, the value would have been specified as `'456'` in the parameter.

The logical constant '1' does not have to be declared in the calling program. It is declared as type logical and named &A in program B.

Because no length is specified on the DCL command for &B, the default length, which is 32 characters, is passed. Only 6 characters of &B are specified (`ABCDEF`). Because &B is declared with only 4 characters in program B, only those 4 characters are received. If they are changed in program B, those 4 positions for &B will also be changed in program A for the remainder of this call.

The length (LEN) parameter must be specified for &C in program A. If it were not specified, the length would default to the specified value's length, which would be incompatible with the default length expected in program B. &C has a value of `13.52900`.

&W in program B (&D in program A) is received as a character because it is declared as a character. Apostrophes are not necessary to indicate a string if TYPE is *CHAR. In program A, the length defaults to the value's length of 7 (the decimal point is considered a position in a character string). Program B expects a length of 32. The first 7 characters are passed, but the contents past the position 7 cannot be predicted.

The variable &V is a character string `XYZ`, padded with blanks on the right. The variable &U is numeric data, `2.00000`.

For information about default lengths in the DCL commands, refer to the command help or to the command documentation in the *CL* section of the **Programming** category in the **iSeries Information Center**.

Common Errors When Calling Programs and Procedures

The following sections describe the errors encountered most frequently in passing values on a CALL command or a CALLPRC command. Some of these errors can be very difficult to debug, and some have serious consequences for program functions.

Date Type Errors Using the CALL Command

The total length of the command string includes the command name, spaces, parameter names, parentheses, contents of variables and apostrophes used. For most commands, the command string initiates the command processing program as expected. However, for some commands some variables may not be passed as expected. For more information on the topic of variables, see "Working with Variables" on page 21.

When the CALL command is used with the CMD parameter on the SBMJOB command, unexpected results may occur. Syntactically, the CALL command appears the same when used with the CMD parameter as it does when used as the compiler directive for the CALL command. When used with the CMD parameter,

the CALL command is converted to a command string that is run at a later time when the batch subsystem initiates it. When the CALL command is used by itself, the CL compiler generates code to perform the call.

Common problems with decimal constants and character variables often occur. In the following cases, the command string is not constructed as needed:

- When decimal numbers are converted to decimal constants.

  When the command string is run, the decimal constant is passed in a packed form with a length of LEN(15 5). It is not passed in the form specified by the CL variable.
- When a character variable is declared longer than 32 characters.

The contents of the character variable is passed as described previously, usually as a quoted character constant with the trailing blanks removed. As a result, the called program may not be passed enough data.

The following methods can be used to correct errors in constructing command strings:

- Create the CALL command string to be submitted by concatenating the various portions of the command together into one CL variable. Submit the command string using the request data (RQSDTA) parameter of the SBMJOB command.
- For CL character variables larger than 32 characters where trailing blanks are significant, create a variable that is one character larger than needed and substring a non-blank character into the last position. This prevents the significant blanks from being truncated. The called program should ignore the extra character because it is beyond the length expected.
- Create a command that will initiate the program to be called. Submit the new command instead of using the CALL command. The command definition ensures the parameters are passed to the command processing program as expected.

Data Type Errors

When passing a value, the data type (TYPE parameter) must be the same (*CHAR, *DEC, or *LGL) in the calling procedure or program and in the called procedure or program. Errors frequently occur in this area when you attempt to pass a numeric constant. If the numeric constant is enclosed in apostrophes, it is passed as a character string. However, if the constant is not enclosed in apostrophes, it is passed as a packed numeric field with LEN(15 5).

In the following example, a quoted numeric value is passed to a program that expects a decimal value. A decimal data error (escape message MCH1202) occurs when variable &A is referred to in the called program (PGMA):

```
CALL   PGMA PARM('123')  /* CALLING PROGRAM */
PGM    PARM(&A)  /* PGMA */
DCL    &A *DEC LEN(15 5)  /* DEFAULT LENGTH */
.
.
.
IF (&A *GT 0) THEN(...) /* MCH1202 OCCURS HERE */
```

In the following example, a decimal value is passed to a program defining a character variable. Generally, this error does not cause run-time failures, but incorrect results are common:

```
CALL PGMB PARM(12345678)  /* CALLING PROG */

PGM PARM(&A)              /* PGMB */
```

```
DCL &A *CHAR 8
.
.
.
ENDPGM
```

Variable &A in PGMB has a value of hex 001234567800000F.

Generally, data can be passed from a logical (*LGL) variable to a character (*CHAR) variable, and vice versa, without error, so long as the value is expressed as '0' or '1'.

Decimal Length and Precision Errors

If a decimal value is passed with an incorrect decimal length and precision (either too long or too short), a decimal data error (MCH1202) occurs when the variable is referred to. In the following examples, the numeric constant is passed as LEN(15 5), but is declared in the called procedure or program as LEN(5 2). Numeric constants are always passed as packed decimal (15 5).

```
CALL PGMA PARM(123)     /* CALLING PROG */

PGM PARM(&A)                  /* PGMA */
DCL &A *DEC (5 2)
.
.
.
IF (&A *GT 0) THEN(...) /* MCH1202 OCCURS HERE */
```

If a decimal variable had been declared with LEN(5 2) in the calling program or procedure and the value had been passed as a variable instead of as a constant, no error would occur.

If you need to pass a numeric constant to a procedure or program and the procedure or program is expecting a value with a length and precision other than 15 5, the constant can be coded in hexadecimal format. The following CALL command shows how to pass the value 25.5 to a program variable that is declared as LEN(5 2):

```
CALL PGMA PARM(X'02550F')
```

If a decimal value is passed with the correct length but with the wrong precision (number of decimal positions), the receiving procedure or program interprets the value incorrectly. In the following example, the numeric constant value (with length (15 5)) passed to the procedure is handled as 25124.00.

```
CALL PGMA PARM(25.124)  /* CALLING PGM */

PGM PARM(&A)            /* PGMA */
DCL &A *DEC (15 2)      /* LEN SHOULD BE 15 5*/
.
.
.
ENDPGM
```

These errors occur when the variable is first referred to, not when it is passed or declared. In the next example, the called program does not refer to the variable, but instead simply places a value (of the detected wrong length) in the variable returned to the calling program. The error is not detected until the variable is returned to the calling program and first referred to. This kind of error can be especially difficult to detect.

```
PGM           /* PGMA */
DCL &A *DEC (7 2)
CALL PGMB PARM(&A)  /* (7 2) PASSED TO PGMB */
IF (&A *NE 0)  THEN(...)  /* *MCH1202 OCCURS HERE */
.
.
.
ENDPGM
PGM PARM(&A) /* PGMB */
DCL &A *DEC (5 2)        /* WRONG LENGTH */
.
.
.
 CHGVAR &A (&B-&C)      /* VALUE PLACED in &A */
RETURN
When control returns to program PGMA and &A is referred to, the error occurs.
```

Character Length Errors

If you pass a character value longer than the declared character length of the receiving variable, the receiving procedure or program cannot access the excess length. In the following example, PGMB changes the variable that is passed to it to blanks. Because the variable is declared with LEN(5), only 5 characters are changed to blanks in PGMB, but the remaining characters are still part of the value when referred to in PGMA.

```
PGM        /* PGMA */
DCL &A *CHAR 10
CHGVAR &A 'ABCDEFGHIJ'
CALL PGMB PARM(&A)           /* PASS to PGMB */
.
.
.
IF (&A *EQ '  ') THEN(...)  /* THIS TEST FAILS */
ENDPGM

PGM PARM(&A)     /* PGMB */
DCL &A *CHAR 5   /* THIS LEN ERROR*/
CHGVAR &A '  '   /* 5 POSITIONS ONLY; OTHERS UNAFFECTED */
RETURN
```

While this kind of error does not cause an escape message, variables handled this way may function differently than expected.

If the value passed to a procedure or program is shorter than its declared length in the receiving procedure or program, there may be more serious consequences. In this case, the value of the variable in the called procedure or program consists of its values as originally passed, and whatever follows that value in storage, up to the length declared in the called procedure or program. The content of this adopted storage cannot be predicted. If the passed value is a variable, it could be followed by other variables or by internal control structures for the procedure or program. If the passed value is a constant, it could be followed in storage by other constants passed on the CALL or CALLPRC command or by internal control structures.

If the receiving procedure or program changes the value, it operates on the original value and on the adopted storage. The immediate effect of this could be to change other variables or constants, or to change internal structures in such a way that the procedure or program fails. Changes to the adopted storage take effect immediately.

In the following example, two 3-character constants are passed to the called program. Character constants are passed with a minimum of 32 characters for the CALL command. (Normally, the value is passed as 3 characters left-adjusted with trailing blanks.) If the receiving program declares the receiving variable to be longer than 32 positions the extra positions use adopted storage of unknown value. For this example, assume that the two constants are adjacent in storage.

```
CALL PGMA ('ABC' 'DEF')  /* PASSING PROG */

PGM  PARM(&A &B) /* PGMA */
DCL  &A *CHAR 50  /* VALUE:ABC+29' '+DEF+15' ' */
DCL  &B *CHAR 10  /* VALUE:DEF+7' ' */
CHGVAR VAR(&A) (' ')  /* THIS ALSO BLANKS &B */
.
.
.
ENDPGM
```

Values passed as variables behave in exactly the same way.

In the following example, two 3-character constants are passed to the called procedure. Only the number of characters specified are passed for the CALLPRC command. If the receiving program declares the receiving variable to be longer than the length of the passed constant, the extra positions use adopted storage of unknown value.

In the following example, assume the two constants are adjacent in storage.

```
CALLPRC PRCA ('ABC' 'DEF')  /* PASSING PROG */

PGM  PARM(&A &B)       /*  *PRCA  */
DCL  &A *CHAR 5        /*   VALUE:'ABC' + 'DE' */
DCL  &B *CHAR 3        /*   VALUE:'DEF' */
CHGVAR &A ' '          /*   This also blanks the first two bytes of &B */
.
.
.
ENDPGM
```

Using Data Queues to Communicate between Programs and Procedures

Data queues are a type of system object that you can create, to which one HLL procedure or program can send data, and from which another HLL procedure or program can receive data. The receiving program can be already waiting for the data, or can receive the data later.

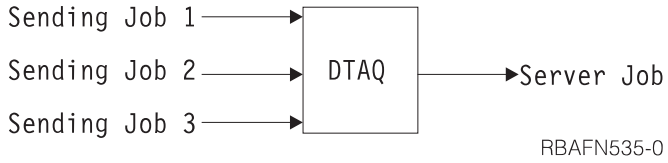
The advantages of using data queues are:

- Using data queues frees a job from performing some work. If the job is an interactive job, this can provide better response time and decrease the size of the interactive program and its process access group (PAG). This, in turn, can help overall system performance. For example, if several workstation users enter a transaction that involves updating and adding to several files, the system can perform better if the interactive jobs submit the request for the transaction to a single batch processing job.
- Data queues are the fastest means of asynchronous communication between two jobs. Using a data queue to send and receive data requires less overhead than using database files, message queues, or data areas to send and receive data.

- You can send to, receive from, and retrieve a description of a data queue in any HLL procedure or program by calling the QSNDDTAQ, QRCVDTAQ, QMHRDQM, QCLRDTAQ, and QMHQRDQD programs without exiting the HLL procedure or program or calling a CL procedure or program to send, receive, clear, or retrieve the description.
- When receiving data from a data queue, you can set a time out such that the job waits until an entry arrives on the data queue. This differs from using the EOFDLY parameter on the OVRDBF command, which causes the job to be activated whenever the delay time ends.
- More than one job can receive data from the same data queue. This has an advantage in certain applications where the number of entries to be processed is greater than one job can handle within the desired performance restraints. For example, if several printers are available to print orders, several interactive jobs could send requests to a single data queue. A separate job for each printer could receive from the data queue, either in first-in-first-out (FIFO), last-in-first-out (LIFO), or in keyed-queue order.
- Data queues have the ability to attach a sender ID to each message being placed on the queue. The sender ID, an attribute of the data queue which is established when the queue is created, contains the qualified job name and current user profile.

In addition to these advantages, you can journal your data queues. This allows you to recover the object to a consistent state, even if the object was in the middle of some change action when the abnormal initial program load (IPL) or crash occurred. Journaling also provides for replication of the data queue journal to a remote system (using remote journal for instance). This lets the system reproduce the actions in a similar environment to replicate the application work. For more information about journaling support on the iSeries server, see the Journal management information in the Information Center.

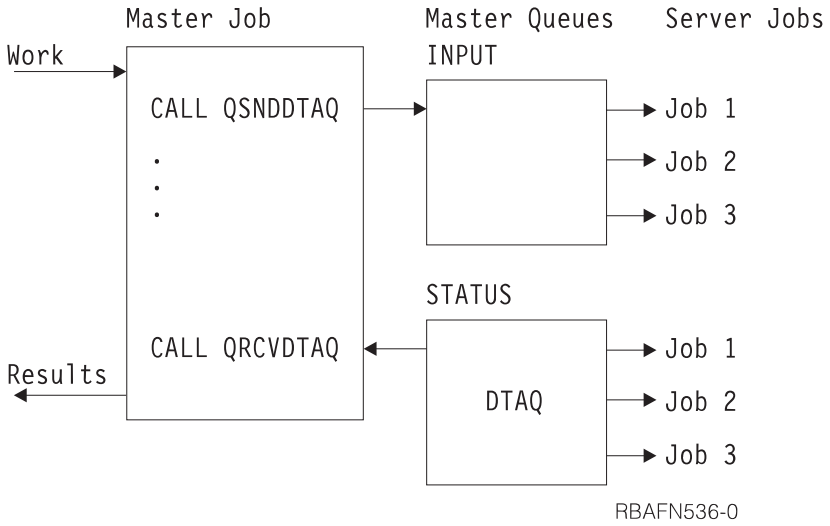
The following is an example showing how data queues work. Several jobs place entries on a data queue. The entries are handled by a server job. This might be used to have jobs send processed orders to a single job that would do the printing. Any number of jobs can send to the same queue.



Another example using data queues follows. A primary job gets the work requests and sends the entries to a data queue (by calling the QSNDDTAQ program). The server jobs receive the entries from the data queue (by calling the QRCVDTAQ program) and process the data. The server jobs can report status back to the primary job using another data queue.

Data queues allow the primary job to route the work to the server jobs. This frees the primary job to receive the next work request. Any number of server jobs can

receive from the same data queue.



When no entries are on a data queue, server jobs have the following options:

- Wait until an entry is placed on the queue
- Wait for a specific period of time; if the entry still has not arrived, then continue processing
- Do not wait, return immediately.

Data queues can also be used when a program needs to wait for input from display files, ICF files, and data queues at the same time. When you specify the DTAQ parameter for the following commands:

- Create Display File (CRTDSPF) command
- Change Display File (CHGDSPF) command
- Override Display File (OVRDSPF) command
- Create ICF File (CRTICFF) command
- Change ICF File (CHGICFF) command
- Override ICF File (OVRICFF) command

you can indicate a data queue that will have entries placed on it when any of the following happens:

- An enabled command key or Enter key is pressed from an invited display device
- Data becomes available from an invited ICF session

Support is available to optionally associate a data queue to an output queue by using the Create Output Queue (CRTOUTQ) or Change Output Queue (CHGOUTQ) command. The system logs entries in the data queue when spooled files are in ready (RDY) status on the output queue. A user program can determine when a spooled file is available on an output queue by using the Receive Data Queue (QRCVDTAQ) API to receive information from a data queue. See the *CL* section of the **Programming** category of the **iSeries Information Center** for details about the Create Output Queue (CRTOUTQ) command. For more information about data queues on output queues see the Printer Device Programming book.

Jobs running on the system can also place entries on the same data queue as the one specified in the DTAQ parameter by using the QSNDDTAQ program.

An application calls the QRCVDTAQ program to receive each entry placed on the data queue and then processes the entry based on whether it was placed there by a display file, an ICF file, or the QSNDDTAQ program. For more information, see “Example 2: Waiting for Input from a Display File and an ICF File” on page 89 and “Example 3: Waiting for Input from a Display File and a Data Queue” on page 91.

Remote Data Queues

You can access remote data queues with Distributed Data Management (DDM) files. DDM files make it possible for a program residing on one server to access a data queue on a remote server to perform any of the following functions:

- send data to a data queue
- receive data from a data queue
- clear data from a data queue

An application program that currently uses a standard data queue can also access a remote DDM data queue without changing or compiling the application again. To ensure the correct data queue is accessed, you may need to do one of the following:

- Delete the standard data queue and create a DDM data queue that has the same name as the original standard data queue.
- Rename the standard data queue.

You can create a DDM data queue with the following command:

```
CRTDTAQ DTAQ(LOCALLIB/DDMDTAQ) TYPE(*DDM)
   RMTDTAQ(REMOTELIB/REMOTEDTAQ) RMTLOCNAME(SYSTEMB)
   TEXT('DDM data queue to access data queue on SYSTEMB')
```

You can also use an expansion of the previous example (″Master Job/Server Job″) to create a DDM data queue to use with remote data queues. The master job resides on SystemA; the data queues and server jobs are moved to SystemB. After creating two DDM data queues (INPUT and STATUS), the master job continues to communicate asynchronously with the server jobs that reside on SystemB. The following example shows how to create a DDM data queue with remote data queues:

```
CRTDTAQ DTAQ(LOCALLIB/INPUT) TYPE(*DDM)
   RMTDTAQ(REMOTELIB/INPUT) RMTLOCNAME(SystemB)
   TEXT('DDM data queue to access INPUT on SYSTEMB')

CRTDTAQ DTAQ(LOCALLIB/STATUS) TYPE(*DDM)
   RMTDTAQ(REMOTELIB/STATUS) RMTLOCNAME(SystemB)
   TEXT('DDM data queue to access STATUS on SYSTEMB')
```

The master job calls QSNDDTAQ, then passes the data queue name of LOCALLIB/INPUT and sends the data to the remote data queue (REMOTELIB/INPUT) on SystemB. To receive data from the remote data queue, (REMOTELIB/STATUS), the master job passes the data queue name of LOCALLIB/STATUS for the call to QRCVDTAQ.

See the *CL* section of the *Programming* category in the **iSeries Information Center** for more information on DDM data queues.

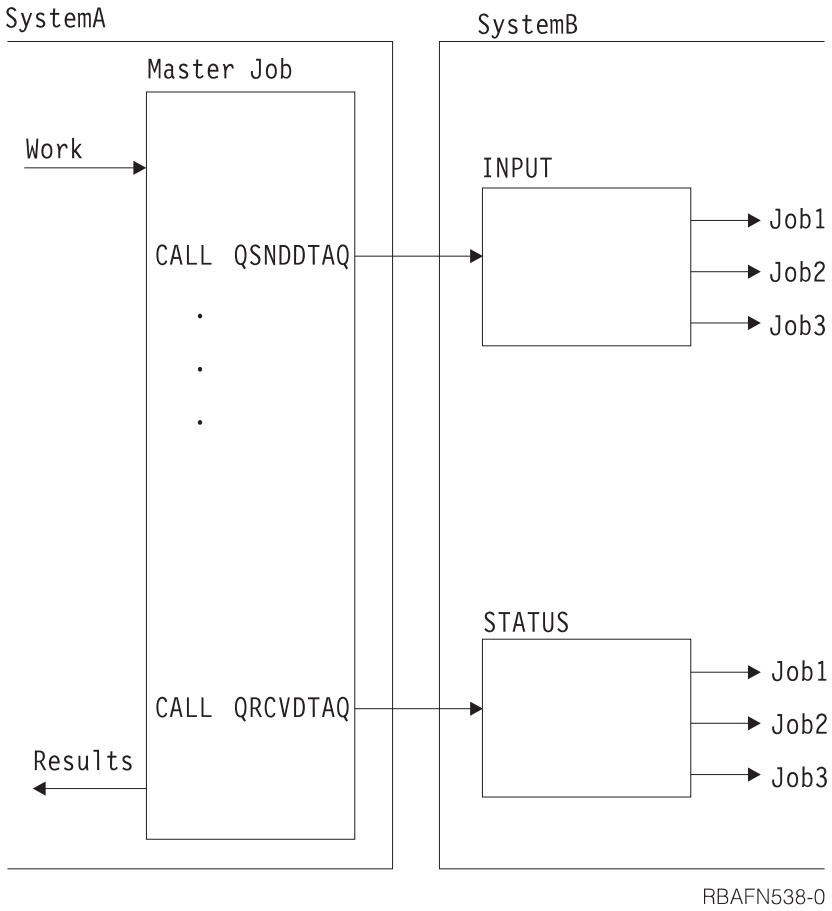


Figure 1. Example of Accessing a Remote Data Queue

Comparisons with Using Database Files as Queues

The following describes the differences between using data queues and database files:

- Data queues have been improved to communicate between active procedures and programs, not to store large volumes of data or large numbers of entries. For these purposes, use database files as queues.
- Data queues should not be used for long-term storage of data. For this purpose, you should use database files.
- When using data queues, you should include abnormal end routines in your programs to recover any entries not yet completely processed before the system is ended.
- It is good practice to periodically (such as once a day) delete and re-create a data queue at a safe point. Performance can be affected if too many entries exist without being removed. Re-creating the data queue periodically will return the data queue to its optimal size. A more efficient approach may be to use the auto reclaim feature, which is described in "Managing the Storage Used by a Data Queue" on page 87.

Similarities to Message Queues

Data queues are similar to message queues, in that procedures and programs can send data to the queue that is received later by another procedure or program.

However, more than one program can have a receive pending on a data queue at the same time, while only one program can have a receive pending on a message queue at the same time. (Only one program receives an entry from a data queue, even if more than one program is waiting.) Entries on a data queue are handled in either first-in-first-out, last-in-first-out, or keyed-queue order. When an entry is received, it is removed from the data queue.

Prerequisites for Using Data Queues

Before using a data queue, you must first create it using the Create Data Queue (CRTDTAQ) command. The following is an example:

```
CRTDTAQ DTAQ(MYLIB/INPUT)  MAXLEN(128)
        TEXT('Sample data queue')
```

The required MAXLEN parameter specifies the maximum length (1 to 64,512 characters) of the entries that are sent to the data queue.

Managing the Storage Used by a Data Queue

Each entry receives a storage allocation when sent to a data queue. The storage allocated will be the value that is specified for the maximum entry length of the data queue that was specified on the Create Data Queue (CRTDTAQ) command. When receiving an entry from a data queue, the data queue removes the entry, but it does not free the auxiliary storage. The system uses the auxiliary storage again when sending a new entry to the data queue. The queue grows larger when not receiving entries that are sent to the queue. Smaller queues that have not been extended past the initial number of entries have better performance. If a data queue has grown too large, delete the data queue by using the Delete Data Queue (DLTDTAQ) command. On completion of the data queue deletion, re-create the queue by using the Create Data Queue (CRTDTAQ) command.

There is another way to manage the size of a data queue on Release V4R5M0 and beyond. This consists of using the SIZE and AUTORCL keywords on the CRTDTAQ command. You can use the SIZE keyword to specify the maximum number of entries and the initial number of entries for the data queue. You can use the AUTORCL keyword for a queue that has been extended to indicate if the data queue should have storage automatically reclaimed when the queue is empty. The amount of storage that remains allocated to the queue equals the initial number of entries specified for the queue when it was created. If AUTORCL contains a value of *NO, which is the default, the system does not automatically reclaim storage from unused space. To reclaim the storage the data queue uses, you would need to delete and re-create it as described in the preceding paragraph. Auto reclaim may be expensive depending on the size of the queue, so the initial number of entries specified on the CRTDTAQ command should be set to the largest typical number of entries that are expected to be on the data queue. If the initial number of entries is set too small, the system will execute the reclaim function more frequently.

Allocating Data Queues

If your application requires that a data queue is not accessed by more than one job at a time, it should be coded to include an Allocate Object (ALCOBJ) command before using a data queue. The data queue should then be deallocated using the Deallocate Object (DLCOBJ) command when the application is finished using it.

The ALCOBJ command does *not*, by itself, restrict another job from sending or receiving data from a data queue or clearing a data queue. However, if all applications are coded to include the ALCOBJ command before any use of a data

queue, the allocation of a data queue already allocated to another job will fail, preventing the data queue from use by more than one job at a time.

When an allocation fails because the data queue is already allocated to another job, the system issues an error message, CPF1002. The Monitor Message (MONMSG) command can be used in the application procedure to monitor for this message and respond to the error message. Possible responses include sending a message to the user and attempting to allocate the data queue again. See "Monitoring for Messages in a CL Program or Procedure" on page 239 for more information.

Examples Using a Data Queue

The following examples explain three methods to process data queue files.

Example 1: Waiting up to 2 Hours to Receive Data from Data Queue

In the following example, program B specifies to wait up to 2 hours (7200 seconds) to receive an entry from the data queue. Program A sends an entry to data queue DTAQ1 in library QGPL. If program A sends an entry within 2 hours, program B receives the entries from this data queue. Processing begins immediately. If 2 hours elapse without procedure A sending an entry, program B processes the time-out condition because the field length returned is 0. Program B continues receiving entries until this time-out condition occurs. The programs are written in CL; however, either program could be written in any high-level language.

The data queue is created with the following command:

```
CRTDTAQ DTAQ(QGPL/DTAQ1) MAXLEN(80)
```

In this example, all data queue entries are 80 bytes long.

In program A, the following statements relate to the data queue:

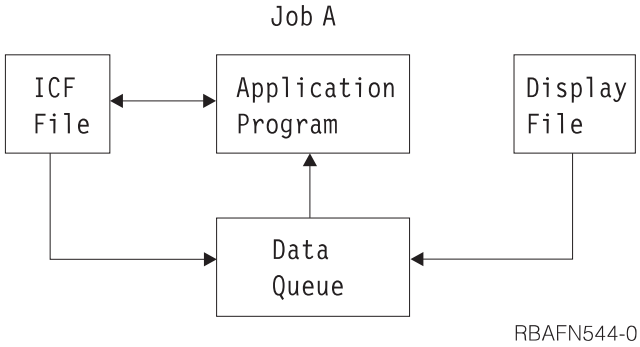
```
PGM
DCL  &FLDLEN  *DEC  LEN(5 0)  VALUE(80)
DCL  &FIELD  *CHAR  LEN(80)
.
.(determine data to be sent to the queue)
.
CALL  QSNDDTAQ  PARM(DTAQ1 QGPL &FLDLEN &FIELD)
.
.
.
```

In program B, the following statements relate to the data queue:

```
        PGM
        DCL  &FLDLEN  *DEC  LEN(5 0)  VALUE(80)
        DCL  &FIELD  *CHAR  LEN(80)
        DCL  &WAIT  *DEC  LEN(5 0)  VALUE(7200)   /* 2 hours */
        .
        .
        .
LOOP:   CALL QRCVDTAQ  PARM(DTAQ1 QGPL &FLDLEN &FIELD &WAIT)
        IF    (&FLDLEN *NE 0) DO   /* Entry received */
               .
               . (process data from data queue)
               .
               GOTO LOOP   /* Get next entry from data queue */
        ENDDO
        .
        . (no entries received for 2 hours; process time-out condition)
        .
```

Example 2: Waiting for Input from a Display File and an ICF File

The following example is different from the usual use of data queues because there is only one job. The data queue serves as a communications object within the job rather than between two jobs.



In this example, a program is waiting for input from a display file and an ICF file. Instead of alternately waiting for one and then the other, a data queue is used to allow the program to wait on one object (the data queue). The program calls QRCVDTAQ and waits for an entry to be placed on the data queue that was specified on the display file and the ICF file. Both files specify the same data queue. Two types of entries are put on the queue by display data management and ICF data management support when the data is available from either file. ICF file entries start with `*ICFF` and display file entries start with `*DSPF`.

The display file or ICF file entry that is put on the data queue is 80 characters in length and contains the field attributes described in the following list. Therefore, the data queue that is specified using the CRTDSPF, CHGDSPF, OVRDSPF, CRTICFF, CHGICFF, and OVRICFF commands must have a length of at least 80 characters.

Position (and Data Type)
Description

1 through 10 (character)
The type of file that placed the entry on the data queue. This field will have one of two values:
`*ICFF` for ICF file
`*DSPF` for display file

If the job receiving the data from the data queue has only one display file or one ICF file open, then this is the only field needed to determine what type of entry has been received from the data queue.

11 through 12 (binary)
The unique identifier for the file. The value of the identifier is the same as the value in the open feedback area for the file. This field should be used by the program receiving the entry from the data queue only if there is more than one file with the same name placing entries on the data queue.

13 through 22 (character)
The name of the display file or ICF file. This is the name of the file actually opened, after all overrides have been processed, and is the same as the file name found in the open feedback area for the file. This field should be used by the program receiving the entry from the data queue only if there is more than one display file or ICF file that is placing entries on the data queue.

23 through 32 (character)
The library where the file is located. This is the name of the library, after all overrides have been processed, and is the same as the library name found in the open feedback area for the file. This field should be used by the program receiving the entry from the data queue only if there is more than one display file or ICF file that is placing entries on the data queue.

33 through 42 (character)
The program device name, after all overrides have been processed. This name is the same as that found in the program device definition list of the open feedback area. For file type *DSPF, this is the name of the display device where the command or Enter key was pressed. For file type *ICFF, this is the name of the program device where data is available. This field should be used by the program receiving the entry from the data queue only if the file that placed the entry on the data queue has more than one device or session invited prior to receiving the data queue entry.

43 through 80 (character)
Reserved.

The following example shows coding logic that the program previously described might use:

```
  .
  .
  .
  .
  OPEN DSPFILE ... /* Open the Display file. DTAQ parameter specified on*/
                   /* CRTDSPF, CHGDSPF, or OVRDSPF for the file.        */

  OPEN ICFFILE ... /* Open the ICF file. DTAQ parameter specified on    */
                   /* CRTICFF, CHGICFF, or OVRICFF for the file.        */

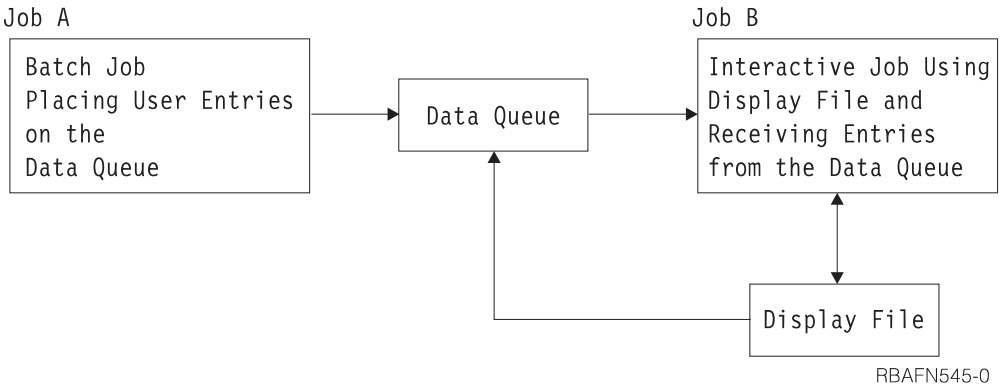
 .
 .
 DO
   WRITE DSPFILE   /* Write with Invite for the Display file            */
   WRITE ICFFILE   /* Write with Invite for the ICF file                */

   CALL QRCVDTAQ   /* Receive an entry from the data queue specified    */
                   /* on the DTAQ parameters for the files. Entries     */
                   /* are placed on the data queue when the data is     */
                   /* available from any invited device or session     */
                   /* on either file.                                   */
                   /* After the entry is received, determine which file */
                   /* has data available, read the data, process it,    */
                   /* invite the file again and return to process the   */
                   /* next entry on the data queue.                     */
   IF 'ENTRY TYPE' FIELD = '*DSPF     ' THEN   /* Entry is from display */
     DO                                       /*  file. Since this entry*/
                                              /* does not contain the   */
                                              /* data received, the data*/
                                              /* must be read from the  */
                                              /* file before it can be  */
       READ DATA FROM DISPLAY FILE            /* processed.             */
       PROCESS INPUT DATA FROM DISPLAY FILE
       WRITE TO DISPLAY FILE                   /* Write with Invite    */
     END
   ELSE                                       /* Entry is from ICF     */
                                              /*  file. Since this entry*/
                                              /* does not contain the   */
                                              /* data received, the data*/
                                              /* must be read from the  */
                                              /* file before it can be  */
                                              /* processed.             */
```

```
      READ DATA FROM ICF  FILE
      PROCESS INPUT DATA FROM ICF FILE
      WRITE TO ICF FILE                      /* Write with Invite     */
  LOOP BACK TO RECEIVE ENTRY FROM DATA QUEUE
.
.
.
END
```

Example 3: Waiting for Input from a Display File and a Data Queue

In the following example, the program in Job B is waiting for input from a display file that it is using and for input to arrive on the data queue from Job A. Instead of alternately waiting for the display file and then the data queue, the program waits for one object, the data queue.



The program calls QRCVDTAQ and waits for the placement of an entry on the data queue that was specified on the display file. Job A is also placing entries on the same data queue. There are two types of entries that are put on this queue, the display file entry, and the user-defined entry. Display data management places the display file entry on the data queue when data is available from the display file. Job A places the user-defined entry on the data queue.

The structure of the display file entry is described in the previous example.

The structure of the entry placed on the queue by Job A is defined by the application programmer.

The following example shows coding logic that the application program in Job B might use:

```
  .
  .
  .
  .
 OPEN DSPFILE ... /* Open the Display file. DTAQ parameter specified on*/
                  /* CRTDSPF, CHGDSPF, or OVRDSPF for the file.       */


.
.
DO
  WRITE DSPFILE   /* Write with Invite for the Display file           */

  CALL QRCVDTAQ   /* Receive an entry from the data queue specified   */
                  /* on the DTAQ parameter for the file. Entries      */
                  /* are placed on the data queue either by Job A or  */
                  /* by display data management when data is          */
                  /* available from any invited device on the display */
```

```
                /* file.                                              */
                /* After the entry is received, determine what type  */
                /* of entry it is, process it, and return to receive */
                /* the next entry on the data queue.                  */
  IF 'ENTRY TYPE' FIELD = '*DSPF     ' THEN   /* Entry is from display */
    DO                                       /*  file. Since this entry*/
                                             /* does not contain the   */
                                             /* data received, the data*/
                                             /* must be read from the  */
                                             /* file before it can be  */
      READ DATA FROM DISPLAY FILE            /* processed.             */
      PROCESS INPUT DATA FROM DISPLAY FILE
      WRITE TO DISPLAY FILE                   /* Write with Invite    */
    END
  ELSE                                       /* Entry is from Job A.  */
                                             /* This entry contains   */
                                             /* the data from Job A,  */
                                             /* so no read is required*/
                                             /* before processing the */
                                             /* data.                 */
      PROCESS DATA QUEUE ENTRY FROM JOB A
  LOOP BACK TO RECEIVE ENTRY FROM DATA QUEUE
.
.
.
END
```

Creating Data Queues Associated with an Output Queue

You can associate a data queue with an output queue. When a spooled file on the output queue goes to a READY status, the data queue entry is sent to the data queue. Use the Create Data Queue (CRTDTAQ) command to create the data queue. Specify the maximum message length (MAXLEN) parameter value as at least 128. The sequence (SEQ) parameter value should be *FIFO or *LIFO.

For information about data queues associated with an output queue, see Data queue support for spooled files in the Printing category in the iSeries Information Center, http://www.iseries.ibm.com/infocenter. This information also includes a sample data queue entry, “Record type 01 data queue entry format”.

Using Data Areas to Communicate between Procedures and Programs

A data area is an object used to hold data for access by any job running on the system. A data area can be used whenever you need to store information of limited size, independent of the existence of procedures or files. Typical uses of data areas are:

- To provide an area (perhaps within each job’s QTEMP library) to pass information within a job.
- To provide a field that is easily and frequently changed to control references within a job, such as:
  - Supplying the next order number to be assigned
  - Supplying the next check number
  - Supplying the next save/restore media volume to be used
- To provide a constant field for use in several jobs, such as a tax rate or distribution list.
- To provide limited access to a larger process that requires the data area. A data area can be locked to a single user, thus preventing other users from processing at the same time.

To create a data area other than a local or group data area, use the Create Data Area (CRTDTAARA) command. By doing this, you create a separate object in a specific library, and you can initialize it to a value. To use the value in a CL procedure or program, use a Retrieve Data Area (RTVDTAARA) command to bring the current value into a variable in your procedure or program. If you change this value in your CL procedure or program and want to return the new value to the data area, use the Change Data Area (CHGDTAARA) command.

To display the current value, use the Display Data Area (DSPDTAARA) command. You can delete a data area using the Delete Data Area (DLTDTAARA) command.

You can journal your data areas. This allows you to recover the object to a consistent state, even if the object was in the middle of some change action when the abnormal IPL or crash occurred. Journaling also provides for replication of the data area journal to a remote system (using remote journal for instance). This lets the system reproduce the actions in a similar environment to replicate the application work. For more information about journaling support on iSeries servers, see the Backup and Recovery book.

Local Data Area

A local data area is created for each job in the system, including autostart jobs, jobs started on the system by a reader, and subsystem monitor jobs.

The system creates a local data area, which is initially filled with blanks, with a length of 1024 and type *CHAR. When you submit a job using the SBMJOB command, the value of the submitting job's local data area is copied into the submitted job's local data area. You can refer to your job's local data area by specifying *LDA for the DTAARA keyword on the CHGDTAARA, RTVDTAARA, and DSPDTAARA commands or *LDA for the substring built-in function (%SST).

The following is true of a local data area:

- The local data area cannot be referred to from any other job.
- You cannot create, delete, or allocate a local data area.
- No library is associated with the local data area.
- You cannot change the local data area in a secondary thread.
- The ILE CL compiler generates code to ensure that a procedure running in a secondary thread cannot access the local data area while a procedure running in the initial thread is changing it.

The local data area contents exist across routing step boundaries. Therefore, using a Transfer Job (TFRJOB), Transfer Batch Job (TFRBCHJOB), Reroute Job (RRTJOB), or Return (RETURN) command does not affect the contents of the local data area.

You can use the local data area to:

- Pass information to a procedure or program without the use of a parameter list.
- Pass information to a submitted job by loading your information into the local data area and submitting the job. Then, you can access the data from within your submitted job.
- Improve performance over other types of data area accesses from a CL procedure or program.
- Store information without the overhead of creating and deleting a data area yourself.

Most high-level languages can also use the local data area. The SBMxxxJOB and STRxxxRDR commands cause jobs to start with a local data area initialized to blanks. Only the SBMJOB command allows the contents of the submitting job's local data area to be passed to the new job.

Group Data Area

The system creates a group data area when an interactive job becomes a group job (using the Change Group Attributes [CHGGRPA] command). Only one group data area can exist for a group. The group data area is deleted when the last job in the group is ended (with the ENDJOB, SIGNOFF, or ENDGRPJOB command, or with an abnormal end), or when the job is no longer part of the group job (using the CHGGRPA command with GRPJOB(*NONE) specified).

A group data area, which is initially filled with blanks, has a length of 512 and type *CHAR. You can use a group data area from within a group job by specifying *GDA for the DTAARA parameter on the CHGDTAARA, RTVDTAARA, and DSPDTAARA commands. A group data area is accessible to all of the jobs in the group.

The following are true for a group data area:

- You cannot use the group data area as a substitute for a character variable on the substring built-in function (%SUBSTRING or %SST). (You can, however, move a 512-byte character variable used by the substring function into or out of the group data area.)
- A group data area cannot be referred to by jobs outside the group.
- You cannot create, delete, or allocate a group data area.
- No library is associated with a group data area.

The contents of a group data area are unchanged by the Transfer to Group Job (TFRGRPJOB) command.

In addition to using the group data area as you use other data areas, you can use the group data area to communicate information between group jobs in the same group. For example, after issuing the Change Group Job Attributes (CHGGRPA) command, the following command can be used to set the value of the group data area:

```
CHGDTAARA  DTAARA(*GDA)  VALUE('January1988')
```

This command can be run from a program or can be issued by the workstation user.

Any other CL procedure or program in the group can retrieve the value of the group data area with the following CL command:

```
RTVDTAARA  DTAARA(*GDA)  RTNVAR(&GRPARA)
```

This command places the value of the group data area (January1988) into CL variable &GRPARA.

Program Initialization Parameter (PIP) Data Area

A PIP data area (PDA) is created for each prestart job when the job is started. The object sub-type of the PDA is different than a regular data area. The PDA can only be referred to by the special value name *PDA. The size of the PDA is 2000 bytes but the number of parameters contained in it is not restricted.

The RTVDTAARA, CHGDTAARA, and DSPDTAARA CL commands and the RTVDTAARA and CHGDTAARA macro instructions support the special value *PDA for the data area name parameter.

Remote Data Areas

You can access remote data areas by using Distributed Data Management (DDM). You do not need to change or recompile an application program that resides on one server when it retrieves data that resides on a remote server. To ensure that you are accessing the correct data area, you may need to do one of the following:

- Delete the standard data area and create a DDM data area that has the same name as the original standard data area
- Rename the standard data area

You can create a DDM data area by doing the following:

```
CRTDTAARA DTAARA(LOCALLIB/DDMDTAARA) TYPE(*DDM)
   RMTDTAARA(REMOTELIB/RMTDTAARA) RMTLOCNAME(SYSTEMB)
   TEXT('DDM data area to access data area on SYSTEMB')
```

To use a value from a data area on a remote server in a CL program, use the Retrieve Data Area (RTVDTAARA) command. Specify the name of a DDM data area to bring the current value into a variable in your program. If you change this value in your CL program and want to return the new value to the remote data area, use the Change Data Area (CHGDTAARA) command and specify the same DDM data area.

If you specify the name of a DDM data area when using the Display Data Area (DSPDTAARA) command, the value of the DDM data area is displayed, rather than the value of the remote data area. You can delete a DDM data area using the Delete Data Area (DLTDTAARA) command.

See the *CL* section of the *Programming* category in the **iSeries Information Center** for more information on DDM data areas.

Creating a Data Area

Unlike variables, data areas are objects and must be created before they can be used. A data area can be created as:

- A character string that can be as long as 2000 characters.
- A decimal value with different attributes, depending on whether it is used only in a CL program or procedure or also with other high-level language programs or procedures. For CL procedures and programs, the data area can have as many as 15 digits to the left of the decimal point and as many as 9 digits to the right, but only 15 digits total. For other languages, the data area can have as many as 15 digits to the left of the decimal point and as many as 9 to the right, for a total of up to 24 digits.
- A logical value '0' or '1', where '0' can mean off, false, or no; and '1' can mean on, true, or yes.

When you create a data area, you can also specify an initial value for the data area. If you do not specify one, the following is assumed:

- 0 for decimal.
- Blanks for character.
- '0' for logical.

To create a data area, use the Create Data Area (CRTDTAARA) command. In the following example, a data area is created to pass a customer number from one program to another:

```
CRTDTAARA  DTAARA(CUST) TYPE(*DEC) +
           LEN(5 0) TEXT('Next customer number')
```

Data Area Locking and Allocation

The CHGDTAARA command uses a *SHRUPD (shared for update) lock on the data area during command processing. The RTVDTAARA and DSPDTAARA commands use a *SHRRD (shared for read) lock on the data area during command processing. If you are performing more than one operation on a data area, you may want to use the Allocate Object (ALCOBJ) command to prevent other users from accessing the data area until your operations are completed. For example, if the data area contains a value that is read and incremented by jobs running at the same time, the ALCOBJ command can be used to protect the value in both the read and update operations. See Chapter 4 for how to allocate objects.

For information on handling data areas in other (non-CL) languages, refer to the appropriate HLL reference manual.

Displaying a Data Area

You can display the attributes (name, library, type, length, data area text description), and the value of a data area. See the *CL* section of the *Programming* category in the **iSeries Information Center** for a detailed description of the Display Data Area (DSPDTAARA) command.

The display uses the 24-digit format with leading zeros suppressed.

Changing a Data Area

The Change Data Area (CHGDTAARA) command changes all or part of the value of a specified data area. It does not change any other attributes of the data area. The new value can be a constant or a CL variable. If the command is in a CL procedure, the data area does not need to exist when the program is created.

Retrieving a Data Area

The Retrieve Data Area (RTVDTAARA) command retrieves all or part of a specified data area and copies it into a CL variable. The data area does not need to exist at compilation time, and the CL variable need not have the same name as the data area. Note that this command retrieves, but does not alter, the contents of the specified data area.

Retrieve Data Area Examples

Example 1

Assume that you are using a data area named ORDINFO to track the status of an order file. This data area is designed so that:

- Position 1 contains an O (open), a P (processing), or a C (complete).
- Position 2 contains an I (in-stock) or an O (out-of-stock).
- Positions 3 through 5 contain the initials of the order clerk.

You would declare these fields in your procedure as follows:

```
DCL VAR(&ORDSTAT) TYPE(*CHAR) LEN(1)
DCL VAR(&STOCKC) TYPE(*CHAR) LEN(1)
DCL VAR(&CLERK) TYPE(*CHAR) LEN(3)
```

To retrieve the order status into &ORDSTAT, you would enter the following:

```
RTVDTAARA DTAARA(ORDINFO (1  1))  RTNVAR(&ORDSTAT)
```

To retrieve the stock condition into &STOCK, you would enter the following:

```
RTVDTAARA DTAARA(ORDINFO (2  1)) RTNVAR(&STOCKC)
```

To retrieve the clerk's initials into &CLERK, you would enter the following:

```
RTVDTAARA DTAARA(ORDINFO (3  3)) RTNVAR(&CLERK)
```

Each use of the RTVDTAARA command requires the data area to be accessed. If you are retrieving many subfields, it is more efficient to retrieve the entire data area into a variable, then use the substring built-in function to extract the subfields.

Example 2

The following example of the RTVDTAARA command places the specified contents of a 5-character data area into a 3-character variable. This example:

- Creates a 5-character data area named DA1 (in library MYLIB) with the initial value of `'ABCDE'`
- Declares a 3-character variable named &CLVAR1
- Copies the contents of the last three positions of DA1 into &CLVAR1

To do this, the following commands would be entered:

```
CRTDTAARA DTAARA(MYLIB/DA1) TYPE(*CHAR) LEN(5) VALUE(ABCDE)
.
.
.
DCL VAR(&CLVAR1) TYPE(*CHAR) LEN(3)
RTVDTAARA DTAARA(MYLIB/DA1 (3 3)) RTNVAR(&CLVAR1)
```

&CLVAR1 now contains `'CDE'`.

Example 3

The following example of the RTVDTAARA command places the contents of a 5-digit decimal data area into a 5-digit decimal digit variable. This example:

- Creates a 5-digit data area named DA2 (in library MYLIB) with two decimal positions and the initial value of `12.39`
- Declares a 5-digit variable named &CLVAR2 with one decimal position
- Copies the contents of DA2 into &CLVAR2

To do this, the following commands would be entered:

```
CRTDTAARA DTAARA(MYLIB/DA2) TYPE(*DEC) LEN(5 2) VALUE(12.39)
.
.
.
DCL  VAR(&CLVAR2) TYPE(*DEC) LEN(5 1)
RTVDTAARA DTAARA(MYLIB/DA2) RTNVAR(&CLVAR2)
```

&CLVAR2 now contains `0012.3` (fractional truncation occurred).

Changing and Retrieving a Data Area Example

The following is an example of using the CHGDTAARA and RTVDTAARA commands for character substring operations.

This example:

- Creates a 10-character data area named DA1 (in library MYLIB) with initial value ABCD5678IJ
- Declares a 5-character variable named &CLVAR1
- Changes the contents of data area DA1 (starting at position 5 for length 4) to the value EFG padding after the G with 1 blank)
- Retrieves the contents of data area DA1 (starting at position 5 for length 5) into the CL variable &CLVAR1

To do this, the following commands would be entered:

```
DCL VAR(&CLVAR1) TYPE(*CHAR) LEN(5)
.
CRTDTAARA DTAARA(MYLIB/DA1) TYPE(*CHAR) LEN(10) +
     VALUE('ABCD5678IJ')
.
.
.
CHGDTAARA DTAARA((MYLIB/DA1) (5 4)) VALUE('EFG')
RTVDTAARA DTAARA((MYLIB/DA1) (5 5)) RTNVAR(&CLVAR1)
```

The variable &CLVAR1 now contains 'EFG I'.

Chapter 4. Objects and Libraries

Objects are the basic units on which commands perform operations. For example, programs and files are objects. Through objects you can find, maintain, and process your data on the iSeries server. You need only know what object and what function (command) you want to use; you do not need to know the storage address of your data to use it.

Note: Objects can reside in both libraries and directories. (Previously, an object could reside only in a library.) This chapter contains information only about objects residing in libraries. See the *Integrated File System* topics in the **Database and File Systems** category of information of the iSeries Information Center for information on directories.

Object Types and Common Attributes

Each type of object on the server has a unique purpose within the system and has an associated set of commands which process that type of object. For a complete list of the types of objects, the abbreviations used as parameter values for object type parameters, and the definition of the object belonging to that type, refer to the *CL* section of the **Programming** category in the **iSeries Information Center**.

Each object type has a set of common attributes that describes the object. These common attributes are listed in Table 3 on page 120. The online help information for the Display Object Description (DSPOBJD) command describes these attributes. Refer to the command documentation in the *CL* section of the **Programming** category in the **iSeries Information Center**

Functions Performed on Objects

Many functions can be performed on objects. Some functions the system performs automatically and others you request through commands.

Functions the System Performs Automatically

The functions performed automatically ensure that objects are processed in a consistent, secure, and correct way. These functions are:

- Object type verification. The system checks the type of object and the type of function being performed on the object to verify that function can be performed on that type of object. For example, if the object specified in a CALL command is not a program, the call function cannot be performed.
- Object authority verification. The system checks the object, the function, and the user to verify that user can perform that function on that object. For example, if USERA is not authorized to use OBJB in any way, he cannot request that any functions be performed on it.
- Object lock enforcement. The system ensures that the integrity of objects is preserved when two or more users try to use an object at the same time. Simultaneous changes to an object are locked out; users cannot use an object while it is being changed.
- Object damage detection and notification. The system monitors for errors during the processing of objects and communicates to you unplanned failures that result from the unrecognizable contents of objects. These failures are communicated to

© Copyright IBM Corp. 1997, 2004

you through standard messages that indicate object damage. The system is designed so that these failures are rare, and monitoring and communicating these failures provide integrity.

Functions You Can Perform Using Commands

The functions you can request through commands are of two types:

- Specific functions for each object type. For example, create, change, and display are specific functions. The specific functions are described in other sections of this manual that describe the object type.
- Some common functions that apply to objects in general are explained in this guide:

Table 2. Common Functions for Objects

Function	Page
Searching for multiple objects or a single object in a library	110
Specifying authority for objects in a library	111
Placing objects in libraries	115
Describing objects	119
Displaying object descriptions	119
Retrieving object descriptions	123
Detecting unused objects on the system	125
Moving objects between libraries	131
Creating duplicate objects	133
Renaming objects	135
Deleting objects	139
Allocating and deallocating objects	139
Displaying the lock states on objects	142
Checking for object existence	147

Libraries

On the iSeries server, objects are grouped in special objects called *libraries*. Objects are found using libraries. To access an object in a library, you must be authorized to the library and to the object. See “Security Considerations for Objects” on page 113 and “Specifying Authority for Libraries” on page 111 for more information.

If you specify a library name in the same parameter as the object name, the object name is called a *qualified name*.

When a library is created you can specify into which user Auxiliary storage pool (ASP) the library should be created. A library can be created into a basic user ASP or an Independent ASP. See the Independent ASPs article for details about the independent ASPs. All objects created into the library are created into the same ASP as the library.

If you are entering a command in which you must specify a qualified name, for example, the object name could be:

```
DISTLIB/ORD040C
```

The order entry program ORD040C is in the library DISTLIB.

If you are using prompting during command entry and you are prompted for a qualified name, you receive prompts for both the object name and the library name. On most commands, you can specify a particular library name, specify *CURLIB (the current library for the job), or use a library list. Library lists are described in the following section.

Library Lists

For commands in which a qualified name can be specified, you can omit specifying the library name. If you do so, either of the following happens:

- For a create command, the object is created and placed in the user's current library, *CURLIB, or in a system library, depending on the object type. For example, programs are created and placed in *CURLIB; authorization lists are created and placed in the system library, QSYS.
- For commands other than a create command, the system normally uses a library list to find the object.

Library lists used by OS/400 consist of the following four parts.

System part
The system part of the library list contains objects needed by the system.

Product libraries
Two product libraries may be included in the library list. The system uses product libraries to support languages and utilities that are dependent on libraries other than QSYS to process their commands.

User commands and menus can also specify a product library on the PRDLIB parameter on the Create Command (CRTCMD) and Create Menu (CRTMNU) commands to ensure that dependent objects can be found.

The product libraries are managed by the system, which automatically places product libraries (such as QRPG) into the reserved product library position in the library list when needed. A product library may be a duplicate of the current library or of a library in the user part of the library list.

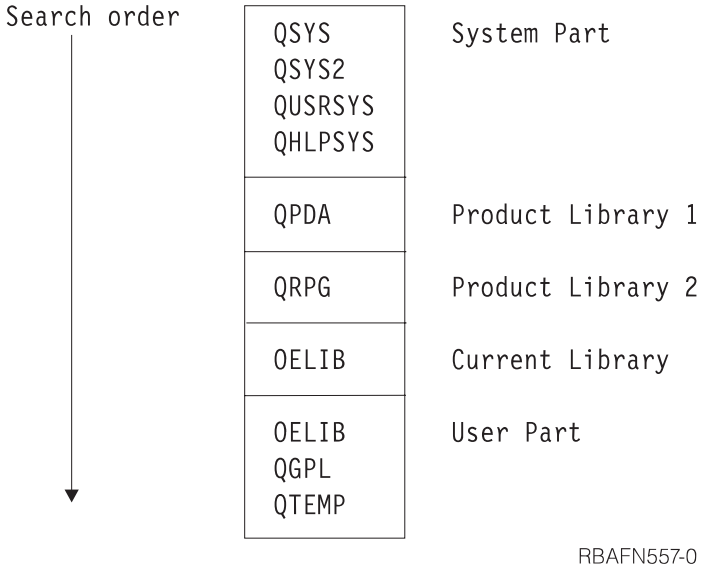
For example, assume that there is a product library in the library list when a command or menu that has a product library starts. The system will replace the product library in the library list with the new product library until the new command ends or the user leaves the new menu.

Current library
The current library can be, but does not have to be, a duplicate of any library in the library list. The value *CURLIB (current library) may be used on most commands as a library name to represent whatever library has been specified as the current library for the job. If no current library exists in the library list and *CURLIB is specified as the library, QGPL is used. You can change the current library for a job by using the Change Current Library (CHGCURLIB) or Change Library List (CHGLIBL) command.

User part
The user part of the library list contains those libraries referred to by the system's users and applications. The user part, and the product and current libraries, may be different for each job on the system. There is a limit of 250 libraries.

For a list of the libraries shipped with the system or optionally installable on the system, see Appendix C, "IBM-Supplied Libraries in Licensed Programs (LP)," on page 413.

The following diagram shows an example of the structure of the library list:



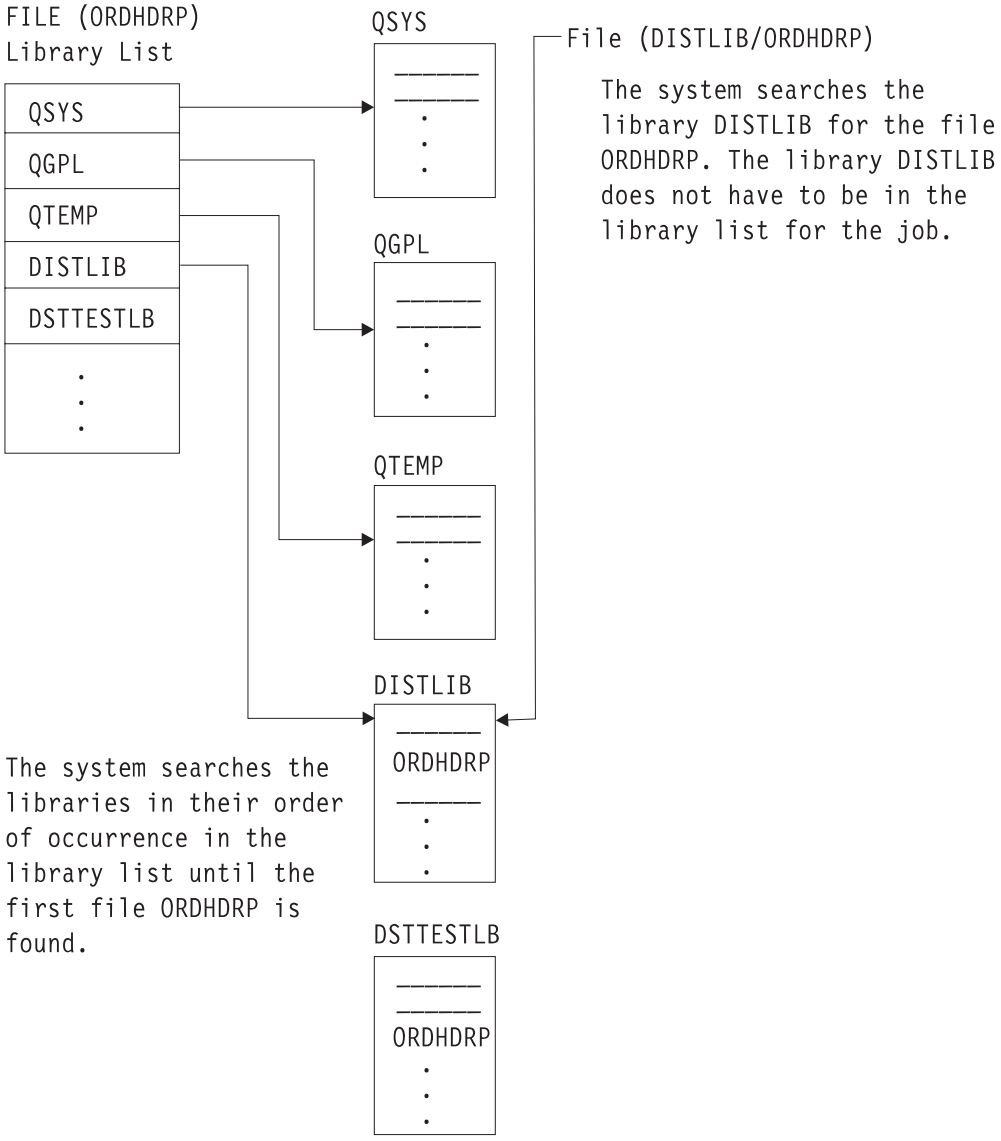
Note: The system places library QPDA in product library 1 when the source entry utility (SEU) is used. When SEU is being used to syntax check source code, a second product library can be added to product library 2. For example, if you are syntax checking RPG source, then QPDA is product library 1 and QRPG is product library 2. In most other system functions, product library 2 is not used.

Using a library list simplifies finding objects on the system. Each job has a library list associated with it. When a library list is used to find an object, each library in the list is searched in the order of its occurrence in the list until an object of the specified name and type is found. If two or more objects of the same type and name exist in the list, you get the object from the library that appears first in the library list. The following diagram shows the searches made for an object both when the library list (*LIBL) is used and when a library name is specified:

Note: Alternatively, use *NLVLIBL instead of *LIBL to qualify any command. Enter the command from a CL program, on a command line, or anywhere you normally enter a command. The system uses *NLVLIBL to determine which libraries to search for the *CMD object. You search only the national language support libraries in the library list by specifying *NLVLIBL.

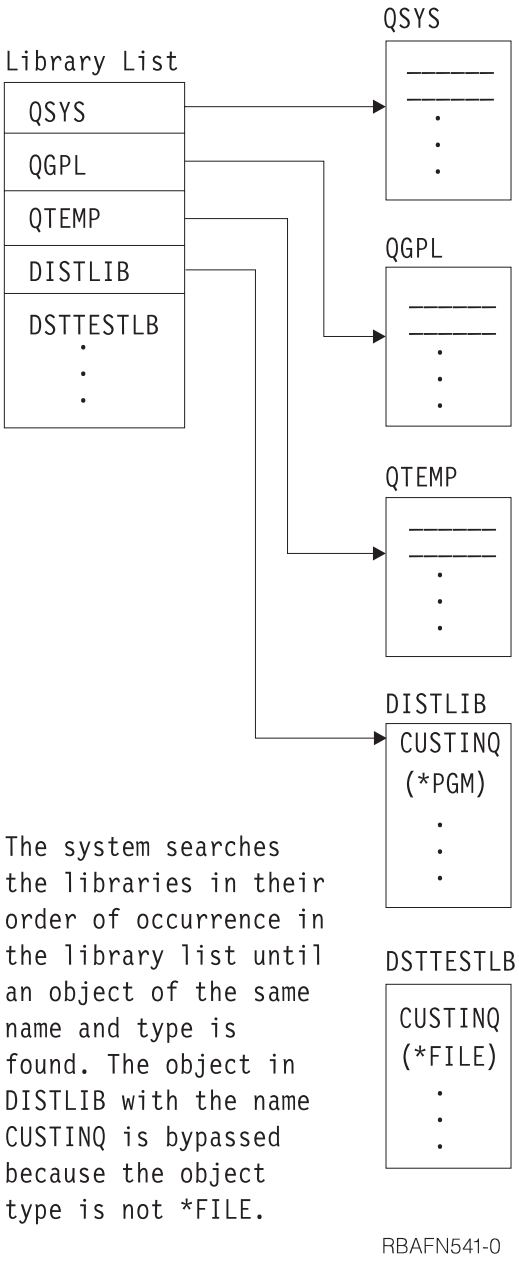
For more information on signing and verification of *CMD objects, see the Object signing and signature verification article in the **Security** category of

information of the iSeries Information Center.



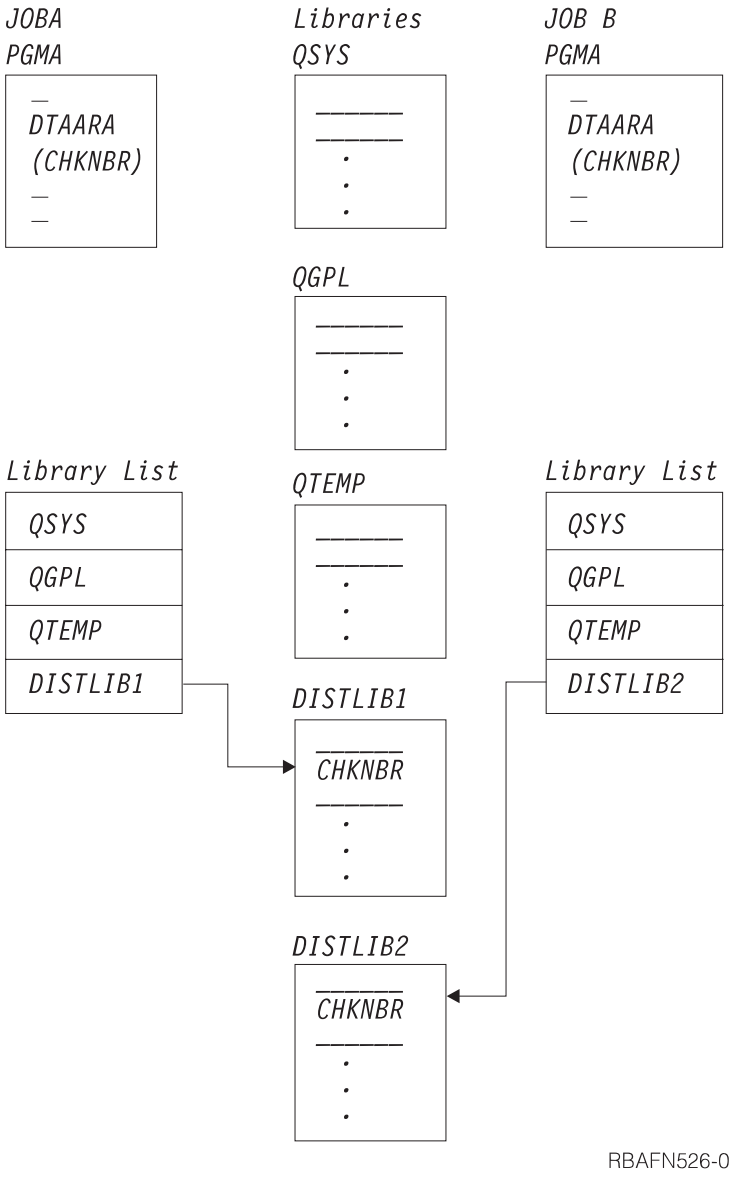
The following diagram shows what happens when two objects of the same name but different types are in the library list. The system will search for CUSTINQ *FILE in the library list by specifying:

```
DSPOBJD OBJ(*LIBL/CUSTINQ) OBJTYPE(*FILE)
```



Generally, a library list is more flexible and easier to use than qualified names. More important than the advantage of not entering the library name, is the advantage of performing functions in an application on different data simply by using a different library list without having to change the application. For example, a CL program PGMA updates a data area CHKNBR. If the library name is not specified, the program can update the data area named CHKNBR in different libraries depending on the use of the library list. For example, assume that JOBA

and JOBB both call PGMA as shown in the following illustration:

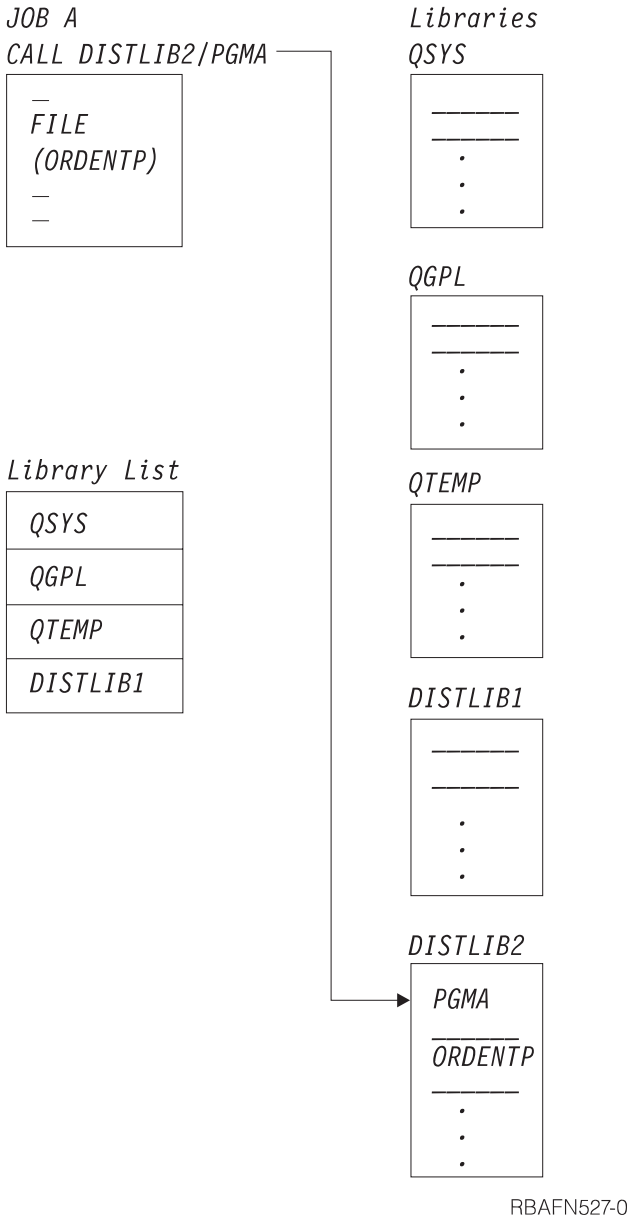


However, the use of a qualified name is advantageous in any of the following situations:

- When the object you are using is not in the library list for the job
- When there is more than one object of the same name in the library list and you want one in a specific library
- When you want to ensure that a specific library is used for security reasons.

If, however, you call a program using a qualified name and the program attempts to open files whose names are not qualified, the files are not opened if they are not

in the library list, as shown in the following example:



The call to PGMA is successful because the program name is qualified on the CALL command. However, when the program attempts to open file ORDENTP, the open operation fails because the file is not in one of the libraries in the library list, and its name is not qualified. If library DISTLIB2 was added to the library list or a qualified file name was used, the program could open the file. Some high-level languages do not allow a qualified file name to be specified. By using an Override (OVRxxx) command, a qualified name can be specified.

A Job's Library List

Each job's library list consists of up to four parts: a system part, a user part, and the current and product libraries. Only the system part *will always* be included in the library list.

When the system is shipped, the system value QSYSLIBL contains the names of the libraries to become the system part of the library list. The shipped values are

QSYS, QSYS2, QHLPSYS, and QUSRSYS. The system value QUSRLIBL contains the names of the libraries to become the user part of the library list.

QSYSLIBL can contain 15 library names, and QUSRLIBL can contain 25 library names. To change the system portion of a job's library list, use the Change System Library List (CHGSYSLIBL) command. To change the value of either QSYSLIBL or QUSRLIBL, use the Change System Value (CHGSYSVAL) command. A change to these system values takes effect on new jobs that are started after the system values are changed.

Changing the Library List

For a running job, you can add entries to or remove entries from the library list by using the Add Library List Entry (ADDLIBLE) command or the Remove Library List Entry (RMVLIBLE) command, or you can change the libraries in the library list by using the CHGLIBL command or the EDTLIBL command. These commands change the user part of the library list, not the system part.

The current library may be added or changed using the Change Current Library (CHGCURLIB) or CHGLIBL command. The current library can also be changed in the user's user profile, at sign-on, or on the Submit Job (SBMJOB) command. The product libraries cannot be added using a CL command; these libraries are added by the system when a command or menu using them is run. The product libraries cannot be changed with a CL command; however, they can be changed with the Change Library List (QLICHGLL) API.

When you use these commands, the change to the library list affects only the job in which the command is run, and the change is effective only as long as the job is running, or until you change the job's library list again. When the library list is changed through the use of these commands, the libraries must exist when the command is run. A library cannot be deleted if it exists in your job's library list. If it exists in another job's library list, it cannot be deleted only if the system value QLIBLCKLVL is set to lock libraries in the library search list.

When a job is started, the user portion of the library list is determined by the values contained in the job description or by values specified on the SBMJOB command. A value of *SYSVAL can be specified, which causes the libraries specified by the system value QUSRLIBL to become the user portion of the library list. If you have specified library names in both the job description and the Batch Job (BCHJOB) or SBMJOB command, the library names specified in the BCHJOB or SBMJOB command override both the libraries specified in the job description and the system value QUSRLIBL.

The following shows the order in which the user part of the library list specified in QUSRLIBL is overridden by commands for individual jobs:

- A library list can be specified in the job description that, when the job is run, overrides the library list specified in QUSRLIBL. (See the *Work Management* topic of the **Systems Management** category in the iSeries Information Center for more information on job descriptions.)
- When a job is submitted either through a BCHJOB command or a SBMJOB command, a library list can be specified on the command. This list overrides the library list specified in the job description or in the system value QUSRLIBL.
- When a job is submitted using the SBMJOB command, *CURRENT (the default) can be specified for the library list. *CURRENT indicates that the library list of the job issuing the SBMJOB command is used.

- Within a job, an ADDLIBLE, RMVLIBLE, or CHGLIBL command can be used. These commands override any previous library list specifications.
- The current library for the job can be changed using the CHGCURLIB or CHGLIBL command.

Instead of entering the CHGLIBL command each time you want to change the library list, you can place the command in a CL program:

```
PGM  /* SETLIBL - Set library list */
CHGLIBL LIBL(APPDEVLIB QGPL QTEMP)
ENDPGM
```

If you normally work with this library list, you could set up an initial program to establish the library list instead of calling the program each time:

```
PGM  /* Initial program for QPGMR */
CHGLIBL LIBL(APPDEVLIB QGPL QTEMP)
TFRCTL PGM(QPGMMENU)
ENDPGM
```

This program must be created and the user profile to which it will apply changed to specify the new initial program. Control then transfers from this program to the QPGMMENU program, which displays the Programmer Menu.

If you occasionally need to add a library to the library list specified in your initial program, you can use the ADDLIBLE command to add that library to the library list. For example, the following command adds the library JONES to the end of the library list:

```
ADDLIBLE LIB(JONES) POSITION(*LAST)
```

If part of your job requires a different library list, you can write a CL program that saves the current library list and later restores it, such as the following program.

```
      PGM
      DCL &LIBL *CHAR 2750
      DCL &CMD *CHAR 2760
(1)   RTVJOBA USRLIBL(&LIBL)
(2)   CHGLIBL (QGPL QTEMP)
      .
      .
      .
(3)   CHGVAR &CMD ('CHGLIBL (' *CAT &LIBL *TCAT ')')
(4)   CALL QCMDEXC (&CMD 2760)
      .
      .
      .
      ENDPGM
```

(1) Command to save the library list. The library list is stored into variable &LIBL. Each library name occupies 10 bytes (padded on the right with blanks if necessary), and one blank is between each library name.

(2) This command changes the library list as required by the following function.

(3) The Change Variable (CHGVAR) command builds a CHGLIBL command in variable &CMD.

(4) QCMDEXC is called to process the command string in variable &CMD. The CHGVAR command is required before the call to QCMDEXC because concatenation cannot be done on the CALL command.

Considerations for Setting Up a Library List

You should consider the following when setting up a library list and using it:

- The libraries in a library list must exist on the system. The system values QSYSLIBL and QUSRLIBL are accessed when OS/400 is started. If a library in either of these values does not exist on the system, a message is sent to the system operator's message queue (QSYSOPR), the library is ignored, and OS/400 is started without the library. Once OS/400 is started, no libraries in the library list of any active job can be deleted. You cannot delete a library in your job's library list. For another job, you cannot delete a library in the library list if the QLIBLCKLVL system value is set to lock libraries in the library search list. If any library in the library list specified in the job description or in a Batch Job (BCHJOB) or Submit Job (SBMJOB) command does not exist or is not available, the job is not started.
- The libraries in a library list must be authorized to all users who need to use them. To initialize a library list (for example, in a Submit Job [SBMJOB], Job [JOB], or Create Job Description [CRTJOBD] command), a user must have object operational authority for the libraries or the job is not started. A user must also have *USE authority to libraries added to the library list using the Add Library List Entry (ADDLIBLE) or Change Library List (CHGLIBL) command.
- When a program running under an adopted user profile adds a library to the library list that the current user is not authorized to and does not remove the library from the library list before ending the program, the user keeps (*USE authority) access to the library after the program exits. This only occurs when *LIBL is specified to access the objects.
- System performance is better when the library list is kept as short as possible.

Displaying a Library List

You can use the Display Library List (DSPLIBL) command to display the library list for a job currently running. The display contains a list of all the libraries in the library list in the same order that they appear in the library list.

You can also display the library list for an active job using the Display Job (DSPJOB) command and selecting option 13 from the Display Job menu.

Using Generic Object Names

Sometimes you may want to search for more than one object (even though only one might be found) when the object names start with the same characters. This type of search is called a *generic search* and can be used on several commands.

To use a generic search, specify a generic name in place of the object name on the command. A generic name consists of a set of characters common to all the object names that identifies a group of objects and ends with an * (asterisk). All objects whose names begin with the specified characters and to which you are authorized have the requested function performed on them. For example, if you entered the Display Object Description (DSPOBJD) command using the generic name ORD*, object descriptions for the objects beginning with ORD are shown.

A generic search can be limited by the following library qualifiers on the generic name (the library name parameter value is given in parentheses, if applicable):

- A specified library. The operation you requested is performed on the generically named objects in the specified library only.

- The library list for the job (*LIBL). The libraries are searched in the order they are listed in the library list. The operation you requested is performed on the generically named objects in the libraries specified in the library list for the job.
- The current library for the job (*CURLIB). The current library for the job is searched. If no current library exists, QGPL is used.
- All libraries in the user part of the library list for the job (*USRLIBL). The libraries are searched in the order they are listed in the library list, including the current library (*CURLIB). The operation you requested is performed on the generically named objects in the libraries specified in the user portion of the library list for the job.
- All user libraries for which you are authorized (*ALLUSR) and libraries beginning with the letter Q as listed in Generic library names in the *APIs* section of the **Programming** category in the **iSeries Information Center** .
- The libraries are searched in alphanumeric order. The following S/36 environment libraries that begin with # are not searched with *ALLUSR specified: #CGULIB, #COBLIB, #DFULIB, #DSULIB, #RPGLIB, #SDALIB, and #SEULIB. The operation you requested is performed on the generically named objects in all the user libraries for which you are authorized.
- All libraries on the system for which you are authorized (*ALL). The libraries are searched in alphanumeric order. The operation you requested is performed on the generically named objects in all the libraries on the system for which you are authorized.

IBM provides information on operations that use generic functions. Refer to the *CL* section of the **Programming** category in the **iSeries Information Center**.

Searching for Multiple Objects or a Single Object

In all commands for which you can specify a generic name, you can specify an object name (no asterisk is specified) and you can search for multiple objects. If you specify an object name and *ALL or *ALLUSR for the library name, the system searches for multiple objects, and the search returns objects of the indicated name and type for which you are authorized. If you specify a generic name, or if you specify *ALL, *ALLUSR, or a library with an object name, you can specify all supported object types (or *ALL object types).

Using Libraries

A library is an object used to group related objects and to find objects by name. Thus, a library is a directory to a group of objects.

You can use libraries to:

- Group certain objects for individual users. This helps you manage the objects on your system. For example, you might place all the files that a user JOE can use in a library JOELIB.
- Group all objects used for an individual application. For example, you might place all your order entry files and programs into an order entry library DISTLIB. You need only add one library to the library list to ensure that all your order entry files and programs are in the list. This is advantageous if you do not want to specify a library name every time you use an order entry file or program.
- Ensure security. For example, you can specify which users have authority to use the library and what they are allowed to do with the library.

- Simplify security by having automatic authorization list and public authority assignment for newly created objects based on the CRTAUT parameter value of the library. Auditing attributes for newly created objects can be set based on the Create Object Auditing (CRTOBJAUD) parameter value.
- Simplify save/restore operations by grouping objects that are saved and restored at the same time into the same library. You can use a Save Library (SAVLIB) command instead of saving objects individually using the Save Object (SAVOBJ) command.
- Use multiple libraries for testing. Refer to "Debugging OPM Programs" on page 378 for more information.
- Use multiple production libraries. For example, you can use one production library for source files and for the creation of objects, one for the application programs and files, one for objects that are infrequently saved, and one for objects that are frequently saved.

Multiple libraries make it easier to use objects. For example, you can have two files with the same name but in different libraries so that one can be used for testing and the other for normal processing. As long as you do not specify the library name in your program, the file name in the program does not have to be changed for testing or normal processing. You control which library is used by using the library list. (Objects of the same type can have the same names only if they are in different libraries.)

The two types of libraries are production and test. A production library is for normal processing. In debug mode, you can protect database files in production libraries from being updated. While in debug mode, any files in test libraries can be updated without any unique specifications. (See "Debugging OPM Programs" on page 378 for more information on using test libraries.)

Creating a Library

To create a library, use the Create Library (CRTLIB) command. For example, the following CRTLIB command creates a library to be used to contain order entry files and programs. The library is named DISTLIB and is a production library. The default authority given to the public prevents a user from accessing the library. Any object created into the library is given the default public authority of *CHANGE based on the CRTAUT value.

```
CRTLIB   LIB(DISTLIB) TYPE(*PROD) CRTAUT(*CHANGE) CRTOBJAUD(*USRPRF) +
         ASP(1) ASPDEV(*ASP) AUT(*EXCLUDE) TEXT('Distribution library')
```

You should not create a library with a name that begins with the letter Q. During a generic search, the system assumes that most libraries with names that begin with the letter Q (such as QRPG or QPDA) are system libraries. See "Using Generic Object Names" on page 109 for more information.

Specifying Authority for Libraries

The following describes each of the authorities that can be given to users for libraries. See the Security - Reference book for more information.

Object Authority

Object operational authority for a library gives the user authority to display the description of a library.

Object management authority for a library includes authority to:

- Grant and revoke authority. You can only grant and revoke authorities that you have. Only the object owner or a user with *ALLOBJ authority can grant object management authority for a library.
- Rename the library.

Object existence authority and **use authority** gives the user authority to delete a library.

Object existence authority and **object operational authority** gives the user authority to transfer ownership of the library.

Data Authority

Add authority and **read authority** for a library allows a user to create a new object in the library or to move an object into the library.

Update authority and **execute authority** for a library allow a user to change the name of an object in the library, provided the user is also authorized to the object.

Delete authority allows the user to remove entries from an object. Delete authority for a library does not allow a user to delete objects in the library. Authority for the object in the library is used to determine if the object can be deleted.

Execute authority allows the user to search the library for an object.

Combined Authority

***USE authority** for a library (consisting of object operational authority, read authority, and execute authority) includes authority to:

- Use a library to find an object
- Display library contents
- Place a library in the library list
- Save a library (if sufficient authority to the object)
- Delete objects from the library (if the user is authorized to the object in the library)

***CHANGE authority** for a library (consisting of object operational authority and all data authorities to the library) includes authority to:

- Use a library to find an object
- Display library contents
- Place a library in the library list
- Save a library (if sufficient authority to the object)
- Delete objects from the library (if the user is authorized to the object in the library)
- Add objects to the library.

***ALL authority** provides all object authorities and data authorities. The user can delete the library, specify the security for the library, change the library, and display the library's description and contents.

***EXCLUDE authority** prevents users from accessing an object.

To display the authority associated with your library, you may use the Display Object Authority (DSPOBJAUT) command.

Security Considerations for Objects

When the system accesses an object that you refer to, it checks to determine if you are authorized to use the object and to use it in the way you are requesting. Generally, you must be authorized at two levels:

- You must be authorized to use the object on which you have requested a function to be performed.
- You must be authorized to the library containing the object. If a library list is used, you must be authorized to the libraries in the list.

Object authority is controlled by the system's security functions, which include the following:

- An object owner and users with *ALLOBJ special authority have all authority for an object, and can grant and revoke authority to and from other users.
- Users have public authority when private authority has not been granted to them for the object.

The Security - Reference book explains in detail the types of authority that can be granted for an object and what authority a user needs to perform a function on that object. Authority that can be granted for libraries is discussed under "Specifying Authority for Libraries" on page 111.

Special considerations apply when writing a program that must be secure (for example, a program that adopts the security officer's user profile). See the Security - Reference book for information about writing these programs.

Display Audit Journal Entries (DSPAUDJRNE) Command

The Display Audit Journal Entries (DSPAUDJRNE) command allows you to generate security journal audit reports. The reports are based on the audit entry types and the user profile that are specified on the command. You can limit reports to specific time frames, and you can search detached journal receivers. You can direct these reports to the active display or an output queue.

RESTRICTIONS: You must have *ALLOBJ and *AUDIT special authorities to use this command.

Refer to online help to see the parameter and value descriptions for this command.

Default Public Authority for Newly Created Objects

When objects are created in a library, the public authority for the object will, by default, be set by using the CRTAUT value of the library.

By specifying:

```
CRTLIB LIB(TESTLIB) CRTAUT(*USE) AUT(*LIBCRTAUT)
```

The library TESTLIB is created. All objects created into library TESTLIB will, by default, have public authority of *USE. The public authority for library TESTLIB is determined by the CRTAUT value of library QSYS.

By specifying:

```
CRTDTAARA DTAARA(TESTLIB/DTA1) TYPE(*CHAR) +
          AUT(*LIBCRTAUT)

CRTDTAARA DTAARA(TESTLIB/DTA2) TYPE(*CHAR) +
          AUT(*EXCLUDE)
```

Data area DTA1 is created into library TESTLIB. The public authority of DTA1 is *USE based on the CRTAUT value of library TESTLIB.

Data area DTA2 is created into library TESTLIB. The public authority of DTA2 is *EXCLUDE. *EXCLUDE was specified on the AUT parameter of the Create Data Area (CRTDTAARA) command.

An authorization list can also be used to secure an object when it is created into a library.

By specifying:

```
CRTAUTL AUTL(PAYROLL)
CRTLIB  LIB(PAYLIB) CRTAUT(PAYROLL) +
        AUT(*EXCLUDE)
```

An authorization list called PAYROLL is created. Library PAYLIB is created with the public authority of *EXCLUDE. By default, an object created into library PAYLIB is secured by authorization list PAYROLL.

By specifying:

```
CRTPF FILE(PAYLIB/PAYFILE) +
      AUT(*LIBCRTAUT)

CRTPF FILE(PAYLIB/PAYACC) +
      AUT(*CHANGE)
```

File PAYFILE is created into library PAYLIB. File PAYFILE is secured by authorization list PAYROLL. The public authority of file PAYFILE is set to *AUTL as part of the Create Physical File (CRTPF) command. *AUTL indicates that the public authority for file PAYFILE is taken from the authorization list securing file PAYFILE, which is authorization list PAYROLL.

File PAYACC is created into library PAYLIB. The public authority for file PAYACC is *CHANGE since it was specified on the AUT parameter of the CRTPF command.

Note: The *LIBCRTAUT value of the AUT parameter that exists on most CRT commands indicates that the public authority for the object is set to the CRTAUT value of the library that the object is being created into.

The CRTAUT value on the library specifies the default authority for public use of the objects created into the library. These possible values are:

***SYSVAL**
The public authority for the object being created is the value specified in system value QCRTAUT

***ALL** All public authorities

***CHANGE**
Change authority

***USE** Use authority

***EXCLUDE**
Exclude authority

authorization list name
The authorization list secures the object

Default Auditing Attribute for Newly Created Objects

When objects are created in a library, the auditing attribute of the object will, by default, be set by using the CRTOBJAUD value of the library.

By specifying:

```
CRTLIB LIB(PAYROLL) AUT(*EXCLUDE) CRTAUT(*EXCLUDE) CRTOBJAUD(*ALL)
```

all objects created into the payroll library are audited for both read and change access. See the Security - Reference book for details on auditing.

Placing Objects in Libraries

When you create an object, it is placed in a library. If you do not specify a library, the object is placed in the current library for the job (*CURLIB) or, if there is no current library for the job, in QGPL. When a library is created, you can specify the public authority for objects created in the library by using the CRTAUT parameter on the Create Library (CRTLIB) command. All objects placed in that library will assume the specified public authority on the CRTAUT value of the library. To specify a library, you specify a qualified name; that is, a library name and an object name. For example, the following Create Physical File (CRTPF) command creates an order entry physical file ORDHDRP to be placed in DISTLIB.

```
CRTPF FILE(DISTLIB/ORDHDRP)
```

To place an object in a library, you must have read and add authorities for the library.

More than one object of the same type cannot have the same name and be in the same library. For example, two files with the name ORDHDRP cannot both be in the library DISTLIB. If you try to place into a library an object of the same name and type as an object already in the library, the system rejects the request and sends you a message indicating the reason.

Note: Use the QSYS library for system objects only. Do not restore other licensed programs to the QSYS library because changes are lost when installing a new release of OS/400.

Deleting and Clearing Libraries

When you delete a library with the Delete Library (DLTLIB) command, you delete the objects in the library as well as the library itself. When you clear a library with the Clear Library (CLRLIB) command, you delete objects in the library without deleting the library. To delete or clear a library, all you need to specify is the library name. For example:

```
DLTLIB LIB(DISTLIB)
```

or:

```
CLRLIB LIB(DISTLIB)
```

To delete a library, you must have object existence authority for both the library and the objects within the library, and use authority for the library. If you try to

delete a library but do not have object existence authority for all the objects in the library, the library and all objects for which you do not have authority are not deleted. All objects for which you have authority are deleted. If you try to delete a library but do not have object existence authority for the library, not only is the library not deleted, but none of the objects in the library are deleted. If you want to delete a specific object (for which you have object existence authority), you can use a delete command for that type of object, such as the Delete Program (DLTPGM) command.

You cannot delete a library in an active job's library list. You must wait until the end of the job before the deletion of the library is allowed. Because of this, you must delete the library before the next routing step begins. When you delete a library, you must be sure no one else needs the library or the objects within the library.

If a library is part of the initial library list defined by the system values QSYSLIBL and QUSRLIBL, the following steps should be followed to delete the library:

1. Use the Change System Value (CHGSYSVAL) command to remove the library from the system value it is contained in. (The changed system value does not affect the library list of any jobs running.)
2. Use the Change Library List (CHGLIBL) command to change the job's library list.

   The Change System Library List (CHGSYSLIBL), Add Library List Entry (ADDLIBLE), Edit Library List (EDTLIBL), and Remove Library List Entry (RMVLIBLE) commands are also used to change the library list.
3. Use the DLTLIB command to delete the library and the objects in the library.

Note: You cannot delete the library QSYS and should not delete any objects in it. You may cause the system to end because the system needs objects that are in QSYS to operate properly. You should not delete the library QGPL because it also contains some objects that are necessary for the system to be able to perform effectively. You should not use the library QRECOVERY because it is intended for system use only. The library QRECOVERY contains objects that the system needs to operate properly.

For concerns about deleting objects other than libraries, see "Deleting Objects" on page 139.

To clear a library, you must have object existence authority for the objects within the library and use authority for the library. If you try to clear a library but do not have object existence authority for all the objects in the library, the objects you do not have authority for are not deleted from the library. If an object is allocated to someone else, it is not deleted.

Displaying Library Names and Contents

You can use the Display Library (DSPLIB) or Work with Libraries (WRKLIB) command to display or print all the libraries you have authority to and find basic information on each object within the libraries.

The object information includes:

- The name and type of the object
- The attributes of the object
- The size of the object

- The description entered for the object when it was created

On the DSPLIB command, you can also specify a specific library name or names, in which case you bypass the library selection display. In this list, the objects are grouped by library; within each library, they are grouped by object type; within each type, they are listed in alphanumeric order. The order of the libraries is one of the following:

- If libraries are specified on the DSPLIB command, the libraries are displayed in the order they are specified in the display command.
- If *LIBL or *USRLIBL is specified on the DSPLIB command, the order of the libraries matches the order of the libraries in the library list for the job.
- If *ALL or *ALLUSR is specified on the DSPLIB command, the order of the libraries is in alphanumeric order. The user must have read authority for the library to be displayed.

For example, the following DSPLIB command displays a list of the objects contained in DISTLIB:

```
DSPLIB  LIB(DISTLIB)  OUTPUT(*)
```

The asterisk (*) for the OUTPUT parameter means that the libraries are to be shown at the display station if in interactive processing and printed if in batch processing. To print a list when in interactive processing, specify *PRINT instead of taking the default *.

See the *CL* section of the **Programming** category in the **iSeries Information Center** for more information and sample displays for the DSPLIB command.

Displaying and Retrieving Library Descriptions

You can use the Display Library Description (DSPLIBD) and Retrieve Library Description (RTVLIBD) commands to display and retrieve the description of libraries.

The library description information includes:

- Type of library (either PROD or TEST)
- Auxiliary storage pool number of the library
- Auxiliary storage pool device name of the library
- Create authority of the library
- Create object auditing of the library
- Text description of the library

OS/400 Globalization

The OS/400 licensed program supports different national languages on the same system. This allows information in one national language to be presented to one user while information in a different national language is presented to another user.

The language used for user-readable information (displays, messages, printed output, and online help information) is controlled by the library list for the job. By adding a national language library to the system portion of the library list, different national language versions of information can be presented. For the

primary language, a **national language version** is the running code and textual data for each licensed program entered. For the secondary language, it is the textual data for all licensed programs.

The language information for the primary language of the system is stored in the same libraries as the programs for IBM licensed programs. For example, if the primary national language of the system is English, then libraries such as QSYS, QHLPSYS, and QSSP contain information in English. Libraries QSYS and QHLPSYS are on the system portion of the library list. Libraries for other licensed programs (such as QRPGLE for ILE RPG for OS/400*) are added to the library list by the system when they are needed.

National language versions other than the system primary language are installed in secondary national language libraries. Each secondary language library contains a single national language version of the displays, messages, commands prompts, and help for *all* IBM licensed programs. The name of a secondary language library is in the form `QSYSnnnn`, where `nnnn` is a language feature code. For example, the feature code for French is 2928, so the secondary national language library name for French is QSYS2928.

If a user wants information presented in the primary national language of the system, no special action is required. To present information in a national language different from the primary national language of the system, the user must change the library list so that the desired national language library is positioned before all other libraries in the library list that contains national language information. You can use any of the following options to position the desired national language library first:

- You can use the SYSLIBLE parameter on the CRTSBSD or CHGSBSD to present displays, messages, and so on for a specific language. For example:
  ```
  CRTSBSD SBSD(QSBSD 2928) POOLS((1 *NOTSG)) SYSLIBLE(QSYS2928)
  ```
- You can use the LIB parameter on the CHGSYSLIBL command to specify the desired national language library at the top of the library list. For example:
  ```
  CHGSYSLIBL  LIB(QSYS2928)
  ```
- You can set up an initial program in the user profile to specify the desired national library at the top of the library list for an interactive job. This is a good option if the user does not want to run the CHGSYSLIBL command at every sign-on. The initial program uses the Change System Library List (CHGSYSLIBL) command to add the desired national language library to the top of the library list.

  Note: The authority shipped with the CHGSYSLIBL command does not allow all users to run the command.

To enable a user to run the CHGSYSLIBL command without granting the user rights to the command, you can write a CL program containing the CHGSYSLIBL command. The program is owned by the security officer, and adopts the security officer's authority when created. Any user with authority to run the program can use it to change the system part of the library list in the user's job. The following is an example of a program to set the library list for a French user.

```
PGM
  CHGSYSLIBL  LIB(QSYS2928)  /* Use French information */
ENDPGM
```

Describing Objects

Whenever you use a create command to create an object, you can describe the object in a 50-character field on the TEXT parameter of the create command. Some commands allow a default of *SRCMBRTXT which indicates the text for the object being created is to be taken from the text of the source member from which the object is being created. This is valid only for objects created from source in database source files.

If the source input for the create command is a device or inline file, or if source is not used, the default value is blank. This text becomes part of the object description and can be displayed using the Display Object Description (DSPOBJD) or Display Library (DSPLIB) command. The text can be changed using the Change Object Description (CHGOBJD) command or many of the Change (CHGxxx) commands that are specific to each object type.

Displaying Object Descriptions

You can use the Display Object Description (DSPOBJD) or Work with Objects (WRKOBJ) command to display descriptions of objects. These descriptions are helpful for determining if objects exist on the system but are not being used. If you are using batch processing, the descriptions can be printed or written to a database file. If you are using interactive processing, the descriptions can be displayed, printed, or written to a database file.

You can display basic, full, or service attributes for object descriptions. These object descriptions are found in the following table:

Table 3. Attributes Displayed for Object Descriptions

Basic Attributes	Full Attributes	Service Attributes (see Notes)
• Object name • Library name • Library ASP device • Object type • Extended attribute • Object size • Text description (partial)	• Object name • Library name • Library ASP device • Object type • Owner • Primary Group • Extended attribute • User-defined attribute • Text description • Creation date and time • User who created object • System object created on • Object domain • Change date and time • Whether or not usage data collected • Last used date • Days used count • Days used count reset date • Allow change by program • Object auditing value • Digitally signed • Digitally signed system-trusted source • Digitally signed multiple signatures • Object size • Offline size • Associated space size • Optimum space alignment • Freed status • Compression status • Object ASP number • Object overflowed • Object ASP device • Journaling status • Current or last journal • Journal images • Journal entries omitted • Journal start date and time • Save operation date and time • Restore operation date and time • Save command • Device type	• Object name • Library name • Library ASP device • Object type • Source file and library • Member name • Extended attribute • User-defined attribute • Freed status • Object size • Creation date and time • Date and time member in source file was last updated • System level • Compiler • Object control level • Changed by program • Whether or not changed by user • Licensed program • PTF number • APAR ID • Text description of object or object status conditions

Notes:

1. The service information is used by programming support personnel to determine the level of the system on which an object was created and whether or not the object has been changed since it was shipped. Some of this information may be helpful to you because it indicates the source member used to create an object and the last date of change to that source from which the object was created.
2. Library objects contain only the *names* of the objects included in the library. If DSPOBJD for object type *LIB is used, the object size information refers to the size of the library object only, not the total size of the objects included in the library.

   You can use either the Retrieve Library Description API (QLIRLIBD) or the command DSPLIB OUTPUT(*PRINT) to find the total size of the library.

Using the DSPOBJD or WRKOBJ command, you can list the objects in a library for which you are authorized by:

- Name
- Generic name
- Type
- Name or generic name within object type

The objects are listed by library; within a library, they are listed by type. Within object type, the objects are listed in alphanumeric order.

You may want to use the DSPOBJD command in a batch job if you want to display many objects with the *FULL or *SERVICE option. The output can go to a spooled printer file and be printed instead of being shown at the display station, or the output can go to a database file. If you direct the output to a database file, all the attributes of the object are written to the file. Use the Display File Field Description (DSPFFD) command for file QADSPOBJ, in library QSYS, to view the record format for this file.

The following command displays the descriptions of the order entry files (that is, the files in DISTLIB) whose names begin with ORD. ORD* is the generic name.

```
DSPOBJD   OBJ(DISTLIB/ORD*)  OBJTYPE(*FILE) +
          DETAIL(*BASIC)  OUTPUT(*)
```

The resulting basic display is:

```
                    Display Object Description - Basic
                                                                Library 1 of 1
Library . . . . . :    DISTLIB         Library ASP device . :    *SYSBAS

Type options, press Enter.
  5=Display full attributes   8=Display service attributes

Opt  Object      Type      Attribute              Size   Text
 _   ORDDTLP     *FILE        PF                  8192   Order detail
 _   ORDHDRP     *FILE        PF                  8192   Order header

                                                                      Bottom
F3=Exit   F12=Cancel   F17=Top   F18=Bottom
```

If you specify *FULL instead of *BASIC or if you enter a 5 in front of ORDDTLP on the basic display, the resulting full display is:

```
                    Display Object Description - Full
                                                                  Library 1 of 1
Object . . . . . :   ORDDTLP         Attribute  . . . . . :   PF
  Library  . . . . :     DISTLIB       Owner  . . . . . . :   QSECOFR
Library ASP device . :   *SYSBAS         Primary group  . . . :   *NONE
Type . . . . . :   *FILE

User-defined information:
  Attribute  . . . . . . . . . . . . :
  Text . . . . . . . . . . . . . . . :

Creation information:
  Creation date/time . . . . . . . . :   06/08/89  10:17:03
  Created by user  . . . . . . . . . :   QSECOFR
  System created on  . . . . . . . . :   SYSTEM01
  Object domain  . . . . . . . . . . :   *SYSTEM

                                                                        More...
Press Enter to continue.

F3=Exit   F12=Cancel
```

```
                    Display Object Description - Full
                                                                  Library 1 of 1
Object . . . . . :   ORDDTLP         Attribute  . . . . . :   PF
  Library  . . . . :     DISTLIB       Owner  . . . . . . :   QSECOFR
Library ASP device . :   *SYSBAS         Primary group  . . . :   *NONE
Type . . . . . :   *FILE

Change/Usage information:
  Change date/time . . . . . . . . . :   05/11/90  10:03:02
  Usage data collected . . . . . . . :   YES
  Last used date . . . . . . . . . . :   05/11/90
  Days used count  . . . . . . . . . :   20
    Reset date . . . . . . . . . . . :   03/10/90
  Allow change by program  . . . . . :   YES
Auditing/Integrity information:
  Object auditing value  . . . . . . :   *NONE
  Digitally signed . . . . . . . . . :   NO

                                                                        More...
Press Enter to continue.

F3=Exit   F12=Cancel
```

```
                    Display Object Description - Full
                                                                  Library 1 of 1
Object . . . . . :   ORDDTLP         Attribute  . . . . . :   PF
  Library  . . . . :     DISTLIB       Owner  . . . . . . :   QSECOFR
Library ASP device . :   *SYSBAS         Primary group  . . . :   *NONE
Type . . . . . :   *FILE

Storage information:
  Size . . . . . . . . . . . . . . . :   32768
  Offline size . . . . . . . . . . . :   0
  Associated space size  . . . . . . :   3840
  Optimum space alignment  . . . . . :   NO
  Freed  . . . . . . . . . . . . . . :   NO
  Compressed . . . . . . . . . . . . :   INELIGIBLE
  Object ASP number  . . . . . . . . :   1
    Object overflowed  . . . . . . . :   NO
  Object ASP device  . . . . . . . . :   *SYSBAS
Journaling information:
  Currently journaled  . . . . . . . :   NO

                                                                        More...
Press Enter to continue.

F3=Exit   F12=Cancel
```

```
                    Display Object Description - Full
                                                                  Library 1 of 1
Object . . . . . :   ORDDTLP         Attribute  . . . . . :   PF
  Library  . . . . :     DISTLIB      Owner  . . . . . . . :   QSECOFR
Library ASP device . :   *SYSBAS        Primary group  . . . :   *NONE
Type . . . . . :   *FILE

Save/Restore information:
  Save date/time . . . . . :
  Restore date/time  . . . . . :
  Save command . . . . . :
  Device type  . . . . . :
                                                                        Bottom
Press Enter to continue.

F3=Exit   F12=Cancel
```

Retrieving Object Descriptions

You can use the Retrieve Object Description (RTVOBJD) command to return the descriptions of a specific object to a CL procedure. Variables are used to return the descriptions. You can use these descriptions to help you detect unused objects on the system.

The command can return the following descriptions as variables for an object:

- The name of the library that contains the object
- Any extended attribute of an object (such as program or file type)
- User-defined attribute
- Text description of the object
- Name of the object owner's user profile
- Name of the primary group for the object
- Object ASP number
- Library ASP number
- Object ASP device
- Library ASP device
- Indication of whether or not the object overflowed the ASP in which it resides
- Date and time the object was created
- Date and time the object was last changed
- Date and time the object was last saved
- Date and time the object was last saved during a SAVACT (*LIB, *SYSDFN, or *YES) save operation
- Date and time the object was last restored
- Name of the object creator's user profile
- System the object was created on
- Object domain
- Whether or not usage data was collected
- Date the object was last used
- Count (number) of days the object was used
- Date the use count was last reset
- Storage status of the object data
- Compression status of the object

- Size of the object in bytes
- Size of the object in bytes of storage at the time of the last save
- Command used to save the object
- Tape sequence number generated when the object was saved on tape
- Tape or diskette volumes used for saving the object
- Type of the device the object was last saved to
- Name of the save file if the object was saved to a save file
- Name of the library that contains the save file if the object was saved to a save file
- File label used when the object was saved
- Name of the source file that was used to create the object
- Name of the library that contains the source file that was used to create the object
- Name of the member in the source file
- Date and time the member in the source file was last updated
- Level of the operating system when the object was created
- Licensed program identifier, release level, and modification level of the compiler
- Object control level for the created object
- Information about whether or not the object can be changed by the Change Object Description (QLICOBJDD) API
- Indication of whether or not the object has been modified with the Change Object Description (QLICOBJD) API
- Information about whether or not the program was changed by the user
- Name, release level, and modification level of the licensed program if the retrieved object is part of a licensed program
- Program Temporary Fix (PTF) number that resulted in the creation of the retrieved object
- Authorized Program Analysis Report (APAR) identification
- Type of auditing for the object
- Whether or not the object is digitally signed
- Digitally signed system-trusted source
- Digitally signed multiple signatures
- Current journal status for the object
- Current or last journal
- Journal image information
- Journal entries to be omitted information
- The date and time that journaling was last started

RTVOBJD Example

In the following CL procedure, a RTVOBJD command retrieves the description of a specific object. Assume an object called MOBJ exists in the current library (MYLIB).

```
DCL  &LIB      TYPE(*CHAR) LEN(10)
DCL  &CRTDATE  TYPE(*CHAR) LEN(13)
DCL  &USEDATE  TYPE(*CHAR) LEN(13)
DCL  &USECNT   TYPE(*DEC)  LEN(5 0)
DCL  &RESET    TYPE(*CHAR) LEN(13)
.
.
.
```

```
RTVOBJD    OBJ(MYLIB/MOBJ) OBJTYPE(*FILE) RTNLIB(&LIB)
           CRTDATE(&CRTDATE) USEDATE(&USEDATE)
           USECOUNT(&USECNT) RESETDATE(&RESET)
```

The following information is returned to the program:

- The current library name (MYLIB) is placed into the CL variable name &LIB.
- The creation date of MOBJ is placed into the CL variable called &CRTDATE.
- The date that MOBJ was last used is placed into the CL variable called &USEDATE.
- The number of days that MOBJ has been used is placed into the CL variable called &USECNT. The start date of this count is the value placed into the CL variable called &RESET.

Creation Information for Objects

The following information is provided in the object description and is set when the object is created. It is useful for object management and maintenance.

- Creator of the object
  - The creator of the object is the user profile that is performing the create operation. This is true even if the user profile has a group profile and the group profile owns the object.
  - The creator of the object does not change when the ownership changes.
  - The creator is the creator of the object on the media when an object is restored.
  - The creator of the object is the user running the command when an object is duplicated using the Create Duplicate Object (CRTDUPOBJ) command.
  - The creator is `*IBM` for IBM-supplied objects.
  - The creator of the object is blank for user objects that already existed on the system before Version 1, Release 3.0.
- System on which the object was created
  - When an object is restored, the system created on is the system the object on the media was created on.
  - For IBM-supplied objects, the system created on is `00000000`.
  - For objects that already existed on the system before Version 1, Release 3.0, the system that is created is blank.

Detecting Unused Objects on the System

Information provided in the object description can help you detect and manage unused objects on the system.

To detect an unused object, look at both the last-used date and the last-changed date. Change commands do not update the last-used date unless the commands cause the object to be deleted and created again, or the change operation causes the object to be read as a part of the change.

- Date and time of last change
  - When an object is created or changed, the system time stamps the object, indicating the date and time the change occurred.
- Date of last use
  - The date of last use is only updated once per day (the first time an object is used in a day). The system date is used.

- An unsuccessful attempt to use an object does not update the last used date. For example, if a user tries to use an object for which the user is not authorized, the date of last use does not change.
- The date of last use is blank for new objects.
- When an object that already exists on the system is restored, the date of last use comes from the object on the system. If it does not already exist when restored, the date is blank.
- Objects that are deleted and re-created during the restore operation lose the date of last use.
- The last used date for a database file is **not** updated when the number of members in the file is zero. For example, if you use the CRTDUPOBJ to copy objects and there are no members in the database file, the last used date is not updated.
- The last used date for a database file is the last used date of the file member with the most current last used date.
- For logical files, the last used date is the last time a logical member (or cursor) was used.
- For physical files, the last used date is the last time the data in the data space was used through a physical or logical access.

Table 4 contains additional information about operations that cause the last-used date to be updated for various object types.

Table 4. Updating Usage Information

Type of Object	Commands and Operations
All object types	Create Duplicate Object (CRTDUPOBJ) command and other commands, such as the Copy Library (CPYLIB) command, that use CRTDUPOBJ to copy objects. Grant Object Authority (GRTOBJAUT) command (for referenced objects)
Binding directory	When bound with another module or binding directory to create a bound program (CRTPGM command) or bound service program (CRTSRVPGM command). When updated on the Update Program UPDPGM command or Update Service Program (UPDSRVPGM command).
Change Request Description	Change Command Change Request Activity (CHGCMDCRQA)
Chart format	Display Chart (DSPCHT) command
C locale description	Retrieve C Locale Description Source (RTVCLDSRC) command or when referred to in a C program
Class	When used to start a job
Command	When run When compiled in a CL program When prompted during entry of source entry utility (SEU) source When calling the system in check mode **Note:** Prompting from the command line and then pressing F3 is not counted as a use of a command.

Table 4. Updating Usage Information (continued)

Type of Object	Commands and Operations
Communications side information (CSI)	When the CPI-Communications Initialize Conversation (CMINIT) call is used to initialize values for various conversation characteristics from the side information object.
Connection list	When the connection list goes beyond status of vary on pending
Cross system product map	When referred to in a CSP application
Cross system product table	When referred to in a CSP application
Controller description	When the controller goes beyond status of vary on pending
Device description	When the device goes beyond status of vary on pending
Data area	Retrieve Data Area (RTVDTAARA) command Display Data Area (DSPDTAARA) command
Data queue	Usage information for the following APIs is updated only once per job (the first time one of the APIs is initiated). Send Data Queue (QSNDDTAQ) API Receive Data Queue (QRCVDTAQ) API Retrieve Data Queue (QMHQRDQD) API Read Data Queue (QMHRDQM) API
File (database file only unless specified otherwise)	When closed (other files, such as device and save files, also updated when closed) When cleared When initialized When reorganized Commands: • Apply Journaled Changes (APYJRNCHG) command • Remove Journaled Changes (RMVJRNCHG) command
Font resource	When referred to during a print operation
Form definition	When referred to during a print operation
Graphics symbol set	When referred to by a GDDM* or PGR graphics application program When loaded internally or using GSLSS
Job description	When used to establish a job
Job schedule	When the system submits a job for a job schedule entry When the user takes 'Option 10 = submit immediately' from the WRKJOBSCDE panel
Job queue	When an entry is placed on or removed from the queue

Table 4. Updating Usage Information (continued)

Type of Object	Commands and Operations
Line description	When the line goes beyond status of vary on pending
Locale	Retrieve locale API QLGRTVLC When a job starts if the user profile LOCALE value contains a path name to a valid *LOCALE object.
Management collection	Only updated by commands and operations that affect all object types.
Media definition	The SAVLIB, SAVOBJ, RSTLIB, RSTOBJ, SAVCHGOBJ commands; as well as, the BRMS and QSRSAVO API.
Menu	When a menu is displayed using the GO command
Message files	When a message is retrieved from a message file other than QCPFMSG, ##MSG1, ##MSG2, or QSSPMSG (such as when a job log is built, a message queue is displayed, help is requested on a message in the QHST log, or a program receives a message other than a mark message) Merge Message File (MRGMSGF) command except when the message file is QCPFMSG, ##MSG1, ##MSG2, or QSSPMSG
Message queue	When a message is sent to, received from, or listed message queue other than QSYSOPR and QHST
Module	When bound with another module or binding directory to create a bound program (CRTPGM command) or bound service program (CRTSRVPGM command). When updated on the Update Program UPDPGM command or Update Service Program (UPDSRVPGM command).
Network interface description	When the network interface description goes beyond status of vary on pending
Node List	Only updated by commands and operations that affect all object types
Output queue	When an entry is placed on or removed from the queue
Overlay	When referred to during a print operation
Page definition	When referred to during a print operation
Page segment	When referred to during a print operation
Panel group	When the Help key is used to request help information for a specific prompt or panel, the date of usage is updated When a panel is displayed or printed from a panel group
PDF map	Add PDF Map Entry (QPQAPME) API Work with PDF Map Entries (WRKPDFMAPE) command
Print descriptor group	When referred to during a print operation
Product Availability	Only updated by commands and operations that affect all object types

Table 4. Updating Usage Information (continued)

Type of Object	Commands and Operations
Product Load	Only updated by commands and operations that affect all object types
Program	Retrieve CL Source (RTVCLSRC) command When run and not a system program
PSF Configuration	When referred to during a print operation
Query definition	When used to generate a report When extracted or exported
Query manager form	When used to generate a report When extracted or exported
Query manager query	When used to generate a report When extracted or exported
Search index	When the F11 key is used through the online help information When the Start Search Index (STRSCHIDX) command is used
Server storage	Vary Configuration (VRYCFG) is run against a network server description object
Service program	When a bound service program is activated
SQL Package	Only updated by commands and operations that affect all object types
Subsystem description	When subsystem is started
Spelling aid dictionary	When used to create another dictionary When retrieved When a word is found in the dictionary during a spell check and the dictionary is not an IBM-supplied spelling aid dictionary
Table	When used by a program for translation
Time zone description	• Starting a job with the time zone description. • Referencing the time zone description in order to calculate a new current offset when a DST (Daylight Savings Time) boundary is crossed.
User profile	When a job is initiated for the profile When the profile is a group profile and a job is started using a member of the group Grant User Authority (GRTUSRAUT) command (for referenced profile)
Workstation User Customization	Only updated by commands and operations that affect all object types

The following is additional object usage information provided in the object description:

- Counter of number of days used

- The count is increased when the date of last use is updated.
- When an object that already exists on the system is restored, the number of days used comes from the object on the system. If it does not already exist when restored, the count is zero.
- Objects that are deleted and re-created during the restore operation lose the days used count.
- The days used count is zero for new objects.

  Note: The iSeries server *cannot* determine the difference between old and new device files. If you restore a device file on to the system and a device file of that same name already exists, delete the existing file if you want the days used count to be reset to zero. If the file is not deleted, the system will interpret this as a restore operation of an old object and retain the days used count.
- The days used count for a database file is the sum of the days used counts for all file members. If there is an overflow on the sum, the maximum value (of the days used counts field) is shown.

- Date days used count was reset
  - When the days used count is reset using the Change Object Description (CHGOBJD) command or the Change Object Description (QLICOBJD) API, the date is recorded. The user then knows how long the days used count has been active.
  - If the days used count is reset for a file, all of the members have their days used count reset.

Common situations that can delete the days used count and the last used date are as follows:

- Restoring damaged objects on the system.
- Restoring programs when the system is not in a restricted state.

The Display Object Description (DSPOBJD) command can be used to display a full description of an object. You can use the same command to write the description to an output file. To retrieve the descriptions, use the Retrieve Object Description (RTVOBJD) command.

Note: The application programming interface (API), QUSROBJD, provides the same information as the Retrieve Object Description command. For more information, see the *APIs* section of the **Programming** category for the **iSeries Information Center**.

The Retrieve Member Description (RTVMBRD) command and Display File Description (DSPFD) command provide similar information for members in a file.

Object usage information is not updated for the following object types:

- Alert table (*ALRTBL)
- Authorization list (*AUTL)
- Configuration list (*CFGL)
- Class-of-service description (*COSD)
- Data Dictionary (*DTADCT)
- Double-byte character set dictionary (*IGCDCT)
- Double-byte character set sort (*IGCSRT)
- Double-byte character set table (*IGCTBL)

- Edit description (*EDTD)
- Exit Registration (*EXITRG)
- Filter (*FTR)
- Forms control table (*FCT)
- Folder (*FLR)
- Internet Packet Exchange Description (*IPXD)
- Journal (*JRN)
- Journal receiver (*JRNRCV)
- Library (*LIB)
- Mode description (*MODD)
- Network Server Description (*NWSD)
- NetBIOS Description (*NTBD)
- Product definition (*PRDDFN)
- Reference code translation table (*RCT)
- Session description (*SSND)
- S/36 machine description (*S36)
- User-defined SQL type (*SQLUDT)
- User queue (*USRQ)

Moving Objects from One Library to Another

You can use the Move Object (MOVOBJ) command to move objects between libraries. Moving objects from one library to another is useful in that you make an object temporarily unavailable and it lets you replace an out-of-date version of an object with a new version. For example, a new primary file can be created to be temporarily placed in a library other than the one containing the old primary file. Because the data in the old primary file is normally copied to the new primary file, the old primary file cannot be deleted until the new primary file has been created. Then, the old primary file can be deleted and the new primary file can be moved to the library that contained the old primary file.

You can only move an object if you have object management authority for the object, delete and execute authority for the library the object is being moved from, and add and read authority to the library the object is being moved to.

You can move an object out of the temporary library, QTEMP, but you cannot move an object into QTEMP. Also, you cannot move an output queue unless it is empty.

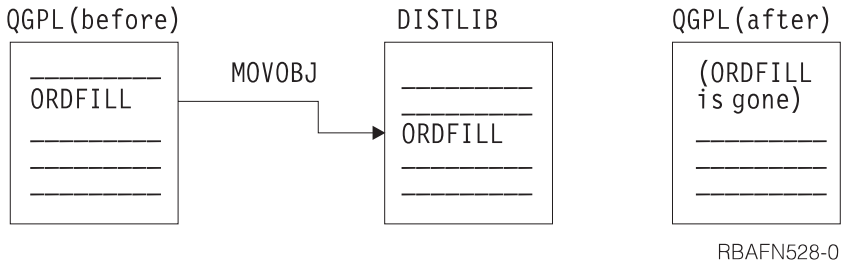
Moving journals and journal receivers is limited to moving these object types back into the library in which they were originally created. If the journal objects have been placed into QRCL by a Reclaim Storage (RCLSTG) command, they must be moved back into their original library to be made operational.

The following is a list of objects that *cannot* be moved:

- Authorization lists (*AUTL)
- Class-of-service descriptions (*COSD)
- Cluster resource group (*CRG)
- Configuration lists (*CFGL)
- Connection lists (*CNNL)

- Controller descriptions (*CTLD)
- Data dictionaries (*DTADCT)
- Device descriptions (*DEVD)
- Display station message queues (*MSGQ)
- Documents (*DOC)
- Edit descriptions (*EDTD)
- Exit registration (*EXITRG)
- Folders (*FLR)
- Double-Byte Character Set (DBCS) font tables (*IGCTBL)
- Image catalog (*IMGCLG)
- Internet Packet Exchange Description (*IPXD)
- Job schedules (*JOBSCD)
- Libraries (*LIB)
- Line descriptions (*LIND)
- Mode descriptions (*MODD)
- NetBIOS description (*NTBD)
- Network interface descriptions (*NWID)
- Structured Query Language (SQL) packages (*SQLPKG)
- System/36™ machine descriptions (*S36)
- The system history log (QHST)
- The system operator message queue (QSYSOPR)
- Time zone description (*TIMZON)
- User-defined SQL type (*SQLUDT)
- User profiles (*USRPRF)

In the following example, a file from QGPL (where it was placed when it was created) is moved to the order entry library DISTLIB so that it is grouped with other order entry files.



To move the object, you must specify the to-library (TOLIB) as well as the object type (OBJTYPE):

```
MOVOBJ  OBJ(QGPL/ORDFILL)  OBJTYPE(*FILE)  TOLIB(DISTLIB)
```

When you move objects, you should be careful not to move objects that other objects depend on. For example, CL procedures may depend on the command definitions of the commands used in the procedure to be in the same library at run time as they were at module creation time. At compile time and at run time, the command definitions are found either in the specified library or in a library in the library list if *LIBL is specified. If a library name is specified, the command definitions must be in the same library at run time as they were at compile time. If *LIBL is specified, the command definitions can be moved between compile time

and program run time as long as they are moved to a library in the library list. Similarly, any application program you write can depend on certain objects being in specific libraries.

An object referring to another object may be dependent on the location of that object (even though *LIBL can be specified for the location of the object). Therefore, if you move an object, you should change any references to it in other objects. The following lists examples of objects that refer to other objects:

- Subsystem descriptions refer to job queues, classes, message queues, and programs.
- Command definitions refer to programs, message files, help panel groups, and source files that are containing REXX procedures.
- Device files refer to output queues.
- Device descriptions refer to translation tables.
- Job descriptions refer to job queues and output queues.
- Database files refer to other database files.
- Logical files refer to physical files or format selections.
- User profiles refer to programs, menus, job descriptions, message queues, and output queues.
- CL programs refer to display files, data areas, and other programs.
- Display files refer to database files.
- Printer files refer to output queues.

Note: You should be careful when moving objects from the system library QSYS. These objects are necessary for the system to perform effectively and the system must be able to find the objects. This is also true for some of the objects in the general-purpose library QGPL, particularly for job and output queues.

The MOVOBJ command moves only one object at a time.

Creating Duplicate Objects

You can use the Create Duplicate Object (CRTDUPOBJ) command to create a copy of an existing object. The duplicate object has the same object type and authorization as the original object and is created into the same auxiliary storage pool (ASP) as the original object. The user who issues the command owns the duplicate object.

Notes:

1. If you create a duplicate object of a journaled file, the duplicate object (file) will not have journaling active. However, you can select this object for journaling later. If you create a duplicate object and the object (file) has no members, the last used date field is blank and the count for number of days used is zero.

You can duplicate an object if you have object management and use authority for the object, use and add authority for the library in which the duplicate object is to be placed, use authority for the library in which the original object exists, and add authority for the process user profile.

To duplicate an authorization list, you must have authorization list management authority for the object and both add and object operational authority for library QSYS.

Only the definitions of job queues, message queues, output queues and data queues are duplicated. Job queues and output queues cannot be duplicated into the temporary library (QTEMP). For a physical file or a save file, you can specify whether the data in the file is also to be duplicated.

The following objects *cannot* be duplicated:

- Class-of-service descriptions (*COSD)
- Cluster resource group (*CRG)
- Configuration lists (*CFGL)
- Connection lists (*CNNL)
- Controller descriptions (*CTLD)
- Data dictionaries (*DTADCT)
- Device descriptions (*DEVD)
- Data queues (*DTAQ)
- Documents (*DOC)
- Edit descriptions (*EDTD)
- Exit registration (*EXITRG)
- Folders (*FLR)
- DBCS font tables (*IGCTBL)
- Image catalog (*IMGCLG)
- Internet Packet Exchange Description (*IPXD)
- Job schedules (*JOBSCD)
- Journals (*JRN)
- Journal receivers (*JRNRCV)
- Libraries (*LIB)
- Line descriptions (*LIND)
- Mode descriptions (*MODD)
- Network interface descriptions (*NWID)
- Network server descriptions (*NWSD)
- Reference code translation tables (*RCT)
- Server storage (*SVTSTG)
- Spelling aid dictionaries (*SPADCT)
- SQL packages (*SQLPKG)
- System/36 machine descriptions (*S36)
- System operator message queue (QSYSOPR)
- System history log (QHST)
- Time zone description (*TIMZON)
- User-defined SQL type (*SQLUDT)
- User profiles (*USRPRF)
- User queues. (*USRQ)

In some cases, you may want to duplicate only some of the data in a file by following the CRTDUPOBJ command with a CPYF command that specifies the selection values.

The following command creates a duplicate copy of the order header physical file, and duplicates the data in the file:

```
CRTDUPOBJ OBJ(ORDHDRP) FROMLIB(DSTPRODLIB) OBJTYPE(*FILE) +
          TOLIB(DISTLIB2) NEWOBJ(*SAME) DATA(*YES)
```

When you create a duplicate object, you should consider the consequences of creating a duplicate of an object that refers to another object. Many objects refer to other objects by name, and many of these references are qualified by a specific library name. Therefore, the duplicate object could contain a reference to an object that exists in a library different from the one in which the duplicate object resides. For all object types other than files, references to other objects are duplicated in the duplicate object. For files, the duplicate objects share the formats of the original file.

Any physical files which exist in the from-library, and on which a logical file is based, must also exist in the to-library. The record format name and record level ID of the physical files in the to- and from-libraries are compared; if the physical files do not match, the logical file is not duplicated.

If a logical file uses a format selection that exists in the from-library, it is assumed that the format selection also exists in the to-library.

Renaming Objects

You can use the Rename Object (RNMOBJ) command to rename objects. However, you can rename an object only if you have object management authority for the object and update and execute authority for the library containing the object.

To rename an authorization list, you must have authorization list management authority, and both update and read authority for library QSYS.

The following objects *cannot* be renamed:

- Class-of-service descriptions (*COSD)
- Cluster resource group (*CRG)
- Data dictionaries (*DTADCT)
- DBCS font tables (*IGCTBL)
- Display station message queues (*MSGQ)
- Documents (*DOC)
- Exit Registration (*EXITRG)
- Folders (*FLR)
- Job schedules (*JOBSCD)
- Journals (*JRN)
- Journal receivers (*JRNRCV)
- Mode descriptions (*MODD)
- Network Server Description (*NWSD)
- SQL packages (*SQLPKG)
- System/36 machine descriptions (*S36)
- The system history log (QHST)
- The system library, QSYS, and the temporary library, QTEMP
- The system operator message queue (QSYSOPR)
- Time zone description (*TIMZON)
- User-defined SQL type (*SQLUDT)
- User profiles (*USRPRF)

Also, you cannot rename an output queue unless it is empty. You should not rename IBM-supplied commands because the licensed programs also use IBM-supplied commands.

To rename an object, you must specify the current name of the object, the name to which the object is to be renamed, and the object type.

The following RNMOBJ command renames the object ORDERL to ORDFILL:

```
RNMOBJ   OBJ(QGPL/ORDERL)  OBJTYPE(*FILE)   NEWOBJ(ORDFILL)
```

You cannot specify a qualified name for the new object name because the object remains in the same library. If the object you want to rename is in use when you issue the RNMOBJ command, the command runs, but does not rename the object. As a result, the system sends you a message.

When you rename objects, you should be careful not to rename objects that other objects depend on. For example, CL programs depend on the command definitions of the commands used in the program to be named the same at run time as they were at compile time. Therefore, if the command definition is renamed in between these two times, the program cannot be run because the commands will not be found. Similarly, any application program you write depends on certain objects being named the same at both times.

You cannot rename a library that contains a journal, journal receiver, data dictionary, cluster resource group, or SQL package.

An object referring to another object may be dependent on the object and library names (even though *LIBL can be specified for the library name). Therefore, if you rename an object, you should change any references to it in other objects. See "Moving Objects from One Library to Another" on page 131 for a list of objects that refer to other objects.

If you rename a physical or logical file, the members in the file are not renamed. However, you can use the Rename Member (RNMM) command to rename a physical or logical file member.

Note: You should be careful when renaming objects in the system library QSYS. These objects are necessary for the system to perform effectively and the system must be able to find the objects. This is also true for some of the objects in the general-purpose library QGPL.

Compressing or Decompressing Objects

You can use the Compress Object (CPROBJ) command to compress selected objects in order to save disk space on the system or you can use the Decompress Object (DCPOBJ) command to decompress objects that have been compressed. The object types that are supported for compression and decompression are *PGM, *SRVPGM, *MODULE, *PNLGRP, *MENU (only UIM menus), and *FILE (only display files or print files). Database files are not allowed to be compressed. Customer objects, as well as OS/400-supplied objects, may be compressed or decompressed. To see or retrieve the compression status of an object, use the Display Object Description (DSPOBJD) command (*FULL display), or the Retrieve Object Description (RTVOBJD) command.

Compression of Objects

Object types, *PGM, *SRVPGM, *MODULE, *PNLGRP, *MENU, and *FILE (display and print files only) can be compressed or decompressed using the CPROBJ or DCPOBJ commands. Objects can be compressed only when both of the following are true:

- If the system can obtain an exclusive lock on the object.
- When the compressed size saves disk space.

The following restrictions apply to the compression of objects:

- Programs created before Version 1 Release 3 of the operating system cannot be compressed.
- Programs, service programs, or modules created before Version 3 Release 6 of the operating system that have not been translated again cannot be compressed.
- Programs in IBM-supplied libraries QSYS and QSSP cannot be compressed unless the paging pool value of the program is *BASE. Use the Display Program (DSPPGM) command to see the paging pool value of a program. Programs in libraries other than QSYS and QSSP can be compressed regardless of their paging pool value.
- Only menus with the attribute UIM can be compressed.
- Only files with attributes DSPF and PRTF can be compressed.
- The system must be in restricted state (all subsystems ended) in order to compress program objects in system libraries.
- The program must not be running in the system when it is compressed, or the program will end abnormally.

Compression runs much faster if you use multiple jobs in nonrestricted state as shown in the following table:

Table 5. Compressing Objects using Multiple Jobs

Object Type	IBM-supplied	User-supplied
*FILE	Job 3: QSYS	Job 7: USRLIB1
*MENU	Job 2: QSYS	Job 8: USRLIB1
*MODULE	Not applicable	Job 10: USRLIB1
*PGM	Restricted State Only	Job 5: USRLIB1
*PNLGRP	Job 1: QSYS Job 4: QHLPSYS	Job 6: USRLIB1
*SRVPGM	Job 11: QSYS	Job 9: USRLIB1

Temporarily Decompressed Objects

Compressed objects are temporarily decompressed automatically by the system when used. A temporarily decompressed object will remain temporarily decompressed until:

- An IPL of the system. This causes the temporarily decompressed object to be deleted (the compressed object remains).
- A Reclaim Temporary Storage (RCLTMPSTG) command is used to reclaim temporarily decompressed objects. This causes temporarily decompressed objects to be deleted (the compressed objects remain) if the objects have not been used for a specified number of days.
- The temporarily decompressed object is used more than 2 days or more than 5 times on the same IPL, in which case it is permanently decompressed.

- A DCPOBJ command is used to decompress the object, in which case it is permanently decompressed.
- The system has an exclusive lock on the object.

Notes:

1. Objects of the type *PGM, *SRVPGM, or *MODULE cannot be temporarily decompressed. If you call a compressed program or debug the program, it is automatically permanently decompressed.
2. Compressed file objects, when opened, are automatically decompressed.
3. If the description of a compressed file is retrieved, the file is temporarily decompressed. Two examples of retrieving a file are:
   - Using the Display File Field Description (DSPFFD) command to display field level information of a file.
   - Using the Declare File (DCLF) command to declare a file.

Automatic Decompression of Objects

Compressed objects shipped in the OS/400 or other IBM licensed programs are decompressed by the system *after the licensed programs are installed*. The decompression occurs only when sufficient storage is available on the system.

System jobs called QDCPOBJx are automatically started by the system to decompress objects.

The number of QDCPOBJ jobs is based on number of processors + 1. The jobs are system jobs running at priority 60 which can't be changed, ended or held by the user. A QDCPOBJx job may be in one of the following statuses, which are from the Work Active Job (WRKACTJOB) command:

- RUN (running): The job is actively decompressing objects.
- EVTW (event wait): The job is not actively decompressing objects. The job is active in case more objects need to decompressed (i.e. additional licensed programs are installed).
- DLYW (delay wait): The job is temporarily halted. The following situations could cause the QDCPOBJx jobs to halt:
  - The system is running in restricted state (i.e. ENDSYS or ENDSBS *ALL was executed)
  - A licensed program was just installed from the ″Work with Licensed Programs″ display. The job is in a delay wait state for a maximum of 15 minutes prior to starting to decompress objects.
- LCKW (lock wait): The job is waiting for an internal lock. Typically, this occurs when one QDCPOBJ job is in DLYW state.

The following storage requirements apply if the operating system was installed over an existing operating system:

- The system must have greater than 250 megabytes of unused storage for the QDCPOBJx jobs to start.
- On a system with available storage of greater than 750MB, the jobs are submitted to decompress all system objects just installed.
- On a system with available storage of less than 250MB, jobs are not submitted, and the objects are decompressed as they are used.
- On a system with available storage between 250MB and 750MB, only frequently-used objects are automatically decompressed.

Frequently-used objects are objects that have been used at least five times and the last use was within the last 14 days. The remaining low-use objects remain compressed.

The system must have greater than 1000MB of unused storage if the operating system is installed on a system that has been initialized using options 2, Install Licensed Internal Code and Initialize the system, from the Install Licensed Internal Code (LIC) display.

If QDCPOBJx jobs are active at the last system termination, the jobs are started again at the time of the next IPL.

Deleting Objects

To delete an object, you can use a delete (DLTxxx) command for that type of object or you can use the delete option on the Work with Objects display (shown from the Work with Libraries (WRKLIB) display). To delete an object, you must have object existence authority to the object and execute authority to the library. Only the owner of an authorization list, or a user with *ALLOBJ special authority, can delete the authorization list.

When you delete an object, you must be sure no one else needs the object or is using the object. Generally, if someone is using an object, it cannot be deleted. However, programs can be deleted unless you use the Allocate Object (ALCOBJ) command to allocate the program before it is called.

Some create commands, such as commands that are used to create programs, commands, and device files, have a REPLACE option. This option allows users to continue using the old version of a previously replaced object. The system stores the old versions of these re-created objects in library QRPLOBJ.

You should be careful of deleting objects that exist in the system libraries. These objects are necessary for the system to perform properly.

On most delete commands, you can specify a generic name in place of an object name. Before using a generic delete, you may want to specify the generic name by using the DSPOBJD command to verify that the generic delete will delete *only* the objects you want to delete. See “Using Generic Object Names” on page 109 for more information on specifying objects generically.

For information about deleting libraries, see “Deleting and Clearing Libraries” on page 115.

Allocating Resources

Objects are allocated on the system to guarantee integrity and to promote the highest possible degree of concurrency. An object is protected even though several operations may be performed on it at the same time. For example, an object is allocated so that two users can read the object at the same time or one user can only read the object while another can read and update the same object.

OS/400 allocates objects by the function being performed on the object. For example:

- If a user is displaying or dumping an object, another user can read the object.

- If a user is changing, deleting, renaming, or moving an object, no one else can use the object.
- If a user is saving an object, someone else can read the object, but not update or delete it; if a user is restoring the object, no one else can read or update the object.
- If a user is opening a database file for input, another user can read the file. If a user is opening a database file for output, another user can update the file.
- If a user is opening a device file, another user can only read the file.

Generally, objects are allocated on demand; that is, when a job step needs an object, it allocates the object, uses the object, and deallocates the object so another job can use it. The first job that requests the object is allocated the object. In your program, you can handle the exceptions that occur if an object cannot be allocated by your request. (See Chapter 7 and Chapter 8 for more information on monitoring for messages or your high-level language reference manual for information on handling exceptions.)

Sometimes you want to allocate an object for a job before the job needs the object, to ensure its availability so a function that has only partially completed would not have to wait for an object. This is called preallocating an object. You can preallocate objects using the Allocate Object (ALCOBJ) command.

Objects are allocated on the basis of their intended use (read or update) and whether they can be shared (used by more than one job). The file and member are always allocated *SHRRD and the file data is allocated with the level of lock specified with the lock state. A lock state identifies the use of the object and whether it is shared. The five lock states are (parameter values given in parentheses):

- Exclusive (*EXCL). The object is reserved for the exclusive use of the requesting job; no other jobs can use the object. However, if the object is already allocated to another job, your job cannot get exclusive use of the object. This lock state is appropriate when a user does not want any other user to have access to the object until the function being performed is complete.
- Exclusive allow read (*EXCLRD). The object is allocated to the job that requested it, but other jobs can read the object. This lock is appropriate when a user wants to prevent other users from performing any operation other than a read.
- Shared for update (*SHRUPD). The object can be shared either for update or read with another job. That is, another user can request either a shared-for-read lock state or a shared-for-update lock state for the same object. This lock state is appropriate when a user intends to change an object but wants to allow other users to read or change the same object.
- Shared no update (*SHRNUP). The object can be shared with another job if the job requests either a shared-no-update lock state, or a shared-for-read lock state. This lock state is appropriate when a user does not intend to change an object but wants to ensure that no other user changes the object.
- Shared for read (*SHRRD). The object can be shared with another job if the user does not request exclusive use of the object. That is, another user can request an exclusive-allow-read, shared-for-update, shared-for-read, or shared-no-update lock state.

Note: The allocation of a library does not restrict the operations that can be performed on the objects within the library. That is, if one job places an exclusive-allow-read or shared-for-update lock state on a library, other jobs

can no longer place objects in or remove objects from the library; however, the other jobs can still update objects within the library.

The following table shows the valid lock state combinations for an object:

Table 6. Valid Lock State Combinations

If One Job Obtains This Lock State:	Another Job Can Obtain This Lock State:
*EXCL	None
*EXCLRD	*SHRRD
*SHRUPD	*SHRUPD or *SHRRD
*SHRNUP	*SHRNUP or *SHRRD
*SHRRD	*EXCLRD, *SHRUPD, *SHRNUP, or *SHRRD

You can specify all five lock states (*EXCL, *EXCLRD, SHRUPD, SHRNUP, and SHRRD) for most object types. this does not apply to *all* object types. Object types that **cannot** have all five lock states specified are listed in the following table with valid lock states for the object type:

Table 7. Valid Lock States for Specific Object Types

Object Type	*EXCL	*EXCLRD	*SHRUPD	*SHRNUP	*SHRRD
Device description		x			
Library		x	x	x	x
Message queue	x				x
Panel group	x	x			
Program	x	x			x
Subsystem description	x				

To allocate an object, you must have object existence authority, object management authority, or operational authority for the object. Allocated objects are automatically deallocated at the end of a routing step. To deallocate an object at any other time, use the Deallocate Object (DLCOBJ) command.

You can allocate a program before it is called to protect it from being deleted. To prevent a program from running in different jobs at the same time, an exclusive lock must be placed on the program in each job before the program is called in any job.

You cannot use the ALCOBJ or DLCOBJ commands to allocate an APPC device description.

The following example is a batch job that needs two files members for updating. Members from either file can be read by another program while being updated, but no other programs can update these members while this job is running. The first member of each file is preallocated with an exclusive-allow-read lock state.

```
//JOB  JOBD(ORDER)
  ALCOBJ  OBJ((FILEA *FILE *EXCLRD) (FILEB *FILE *EXCLRD))
  CALL  PROGX
//ENDJOB
```

Objects that are allocated to you should be deallocated as soon as you are finished using them because other users may need those objects. However, allocated objects are automatically deallocated at the end of the routing step.

If the first members of FILEA and FILEB had not been preallocated, the exclusive-allow-read restriction would not have been in effect. When you are using files, you may want to preallocate them so that you are assured they are not changing while you are using them.

Note: If a single object has been allocated more than once (by more than one allocate command), a single DLCOBJ command will not completely deallocate that object. One deallocate command is required for each allocate command.

It is not an error if the DLCOBJ command is issued against an object where you do not have a lock or do not have the specific lock state requested to be allocated.

You can change the lock state of an object, as the following example shows:

```
PGM
ALCOBJ  OBJ((FILEX *FILE *EXCL)) WAIT(0)
CALL PGMA
ALCOBJ  OBJ((FILEX *FILE *EXCLRD))
DLCOBJ  OBJ((FILEX *FILE *EXCL))
CALL PGMB
DLCOBJ  OBJ((FILEX *FILE *EXCLRD))
ENDPGM
```

File FILEX is allocated exclusively for PGMA, but FILEX is allocated as exclusive-allow-read for PGMB.

You can use record locks to allocate data records within a file. You can also use the WAITFILE parameter on a Create File command to specify how long your program is to wait for that file before a time-out occurs.

The WAITRCD parameter on a Create File command specifies how long to wait for a record lock. The DFTWAIT parameter on the Create Class (CRTCLS) command specifies how long to wait for other objects. For a discussion of the WAITRCD parameter, see the Backup and Recovery book.

Displaying the Lock States for Objects

You can use the Work with Object Locks (WRKOBJLCK) command or the Work with Job (WRKJOB) command to display the lock states for objects.

The WRKOBJLCK command displays all the lock state requests in the system for a specified object. It displays both the held locks and the locks being waited for. For a database file, the WRKOBJLCK command displays the locks at the file level (the object level) but not at the record level. For example, if a database file is open for update, the lock on the file is displayed, but the lock on any records within the file is not. Locks on database file members can also be displayed using the WRKOBJLCK command.

If you use the WRKJOB command, you can select the locks option on the Display Job menu. This option displays all the lock state requests outstanding for the specified active job, the locks being held by the job, and the locks for which the job is waiting. However, if a job is waiting for a database record lock, this does not appear on the object locks display.

The following command displays all the lock state requests in the system for the logical file ORDFILL:

```
WRKOBJLCK  OBJ(QGPL/ORDFILL)  OBJTYPE(*FILE)
```

The resulting display is:

```
                          Work with Object Locks
                                                              System:   SYSTEM01
Object:   ORDFILL        Library:   QGPL           Type:   *FILE-LGL

Type options, press Enter.
  4=End job   5=Work with job   8=Work with job locks

Opt     Job            User           Lock        Status     Scope      Thread
 _      WORKST04       QSECOFR        *SHRRD       HELD      *JOB
                                      *SHRRD       HELD      *JOB
                                      *SHRRD       HELD      *JOB
                                      *SHRRD       HELD      *JOB
                                      *SHRRD       HELD      *JOB
                                      *SHRRD       HELD      *JOB
                                      *SHRRD       HELD      *JOB
                                      *SHRRD       HELD      *JOB
                                      *SHRRD       HELD      *JOB
                                      *SHRRD       HELD      *JOB
                                      *SHRRD       HELD      *JOB
                                      *SHRRD       HELD      *JOB
                                      *SHRRD       HELD      *JOB
                                                                        More...
F3=Exit   F5=Refresh   F6=Work with member locks   F12=Cancel
```

Chapter 5. Working with Objects in CL Procedures and Programs

Accessing Objects in CL Programs

Rules that refer to objects in CL program commands and procedures are the same as objects in commands that are processed individually (not within a program). Object names can be either qualified or unqualified. Locate an unqualified object name through a search of the library list.

Most objects referred to in CL procedures and programs are not accessed until the command referring to them is run. To qualify the name (*library/name*)of an object, it must be in the specified library when the command that refers to it runs. However, the object does not have to be in that library at the creation of the program. This means that most objects can be qualified in CL source statements that are simply based only on their run–time location. "Exceptions: Accessing Command Definitions, Files, and Procedures" on page 146 discusses the exceptions.

You can avoid this run–time consideration for all objects if you do not qualify object names on CL source statements, but refer to the library list (*LIBL/name) instead. If you refer to the library list at compile time, the object can be in any library on the library list at command run time. This is possible providing you do not have duplicate-name objects in different libraries. If you use the library list, you can move the object to a different library between procedure creation and command processing.

Objects do not need to exist until the command that refers to them runs. Because of this, the CL program successfully compiles even though program PAYROLL does not exist at compile time:

```
PGM /*TEST*/
DCL...
MONMSG...
.
.
.
CALL PGM(QGPL/PAYROLL)
.
.
.
ENDPGM
```

In fact, PAYROLL does not have to exist when activating the program TEST, but only when running the CALL command. This creates the called program within the calling program immediately prior to the CALL command:

```
PGM /*TEST*/
DCL...
.
.
.
MONMSG
.
.
.
CRTCLPGM PGM(QGPL/PAYROLL)
CALL PGM(QGPL/PAYROLL)
```

© Copyright IBM Corp. 1997, 2004

```
.
.
.
ENDPGM
```

Note that for create commands, such as (CRTCLPGM) or (CRTDTAARA), the object that is accessed at compile or run time is the create command definition, not the created object. If you are using a create command, the create command definition must be in the library that is used to qualify the command at compile time. (Alternately, it must be in a library on the library list if you used *LIBL.)

Exceptions: Accessing Command Definitions, Files, and Procedures

Two requirements exist for creating a CL program from source statements that refer to command definitions or files.

- The objects must exist at creation time of the program.
- The objects must exist when the command that refers to them runs.

This means that if you use the Declare File (DCLF) command, you must create the file before creating a program that refers to the file.

Accessing Command Definitions

Access to the command definitions occurs during program creation time and at command run time. To allow for syntax checking, the command must exist during the creation of a program that uses it. If it is qualified at creation time, the command needs to exist in the library referred to during creation, and in the same library when processed. If it is not library-qualified, it must be in some library on the library list during creation time and at run time.

The command name should be qualified in the program:

- When the command's definition will not be accessible through the library list while the program is running.
- When multiple command definitions exist with the same name if expecting a specific instance of the command at run time.

The name of the command must be the same when the program runs as when the system created it. An error occurs if the command name changes after creating a program that refers to that command. This is because the program cannot find the command when it runs. However, if a default changes for a parameter on a command, the new default is used when that command runs. For more detail on attributes that you may change on a command without having to re-create the command, see the Change Command (CHGCMD) command description in the *CL* topic of the **Programming** category in the iSeries Information Center.

Accessing Files

The compiler accesses files when compiling a program module that has a Declare File (DCLF) command. The file must exist when compiling a CL module or OPM program that uses it. The file does not have to exist when creating a program or service program that uses the module.

Enter Data Description Specifications (DDS) into a source file before creating it. The DDS describes the record formats and the fields within the records. Additionally, the system compiles this information to create the file object through the Create Display File (CRTDSPF) command.

Note: You can create other types of files from DDS, and each type has its own command: Create Physical File (CRTPF) and Create Logical File (CRTLF) are two that create files that you can use in CL programs and procedures.

The fields that are described in the DDS can be input or output fields (or both). The system declares the fields in the CL program or procedure as variables when it compiles a program or module. The program manipulates data from display through these variables.

If you do not use DDS to create a physical file, the system declares a CL variable to contain the entire record. This variable has the same name as the file, and its length is the same as the record length of the file.

CL programs and procedures cannot manipulate data in any types of files other than display files and database files, except with specific CL commands.

Deletion of the DDS after creating the file is possible but not recommended. You can delete the file after the system compiles the CL program or module that refers to the file. This is true provided the file exists when the command referring to it, such as a Receive File (RCVF), is processed in the program.

The rules on qualified names that are described here for command definitions also apply to files. For more details on files, see "Working with Files in CL Procedures" on page 148.

Accessing Procedures

A procedure that is specified by Call Bound Procedure (CALLPRC), does not have to exist at the time a module that refers to it is created. The system requires the existence of the procedure in order to create a program or service program that uses the procedure. The called procedure may be:

- In a module that is specified on the MODULE parameter on the Create Program (CRTPGM) or CRTSRVPGM command.
- In a service program that is specified on the BNDSRVPGM parameter. The service program must be available at run time.
- In a service program or module that is listed in a binding directory that is specified on the BNDDIR parameter of the CRTPGM command or CRTSRVPGM command. The binding directory and modules do not have to be available at run time.

Checking for the Existence of an Object

Before attempting to use an object in a program, check to determine if the object exists and if you have the authority to use it. This is useful when a function uses more than one object at one time.

To check for the existence of an object, use the Check Object (CHKOBJ) command. You can use this command at any place in a procedure or program. The CHKOBJ command has the following format:

```
CHKOBJ  OBJ(library-name/object-name)  OBJTYPE(object-type)
```

Other optional parameters allow object authorization verification. If you are checking for authorization and intend to open a file, you should check for both operational and data authority.

When this command runs, the system sends messages to the program or procedure to report the result of the object check. You can monitor for these messages and handle them as you wish. For example:

```
          CHKOBJ  OBJ(OELIB/PGMA)  OBJTYPE(*PGM)
          MONMSG  MSGID(CPF9801)  EXEC(GOTO NOTFOUND)
          CALL OELIB/PGMA
          .
          .
          .
NOTFOUND: CALL FIX001 /*PGMA Not Found Routine*/
          ENDPGM
```

In this example, the MONMSG command checks only for the object-not-found escape message. For a list of all the messages which the CHKOBJ command may send see the online help information for the CHKOBJ command. "Using the Monitor Message (MONMSG) Command" on page 50, Chapter 7, and Chapter 8 contain additional information about monitoring for messages.

The CHKOBJ command does not allocate an object. For many application uses the check for existence is not an adequate function, the application should allocate the object. The Allocate Object (ALCOBJ) command provides both an existence check and allocation.

Use the Check Tape (CHKTAP) or Check Diskette (CHKDKT) command to ensure that a specific tape or diskette is placed on the drive and ready. These commands also provide an escape message that you can monitor for in your CL program.

Working with Files in CL Procedures

Two types of files are supported in CL procedures and programs, display files, and database files. You can send a display to a workstation and receive input from the workstation for use in the procedure or program, or you can read data from a database file for use in the procedure or program.

Note: Database files are made available for use within the CL procedure or program through the DCLF and RCVF commands.

To use a file in a CL procedure or program, you must:

- Format the display or database record, identifying fields and conditions which you enter as DDS source. The use of DDS is not required for a database file.
- Create the file using the Create Display File (CRTDSPF) command, Create Physical File (CRTPF) command, or Create Logical File (CRTLF) command. Subfiles (except for message subfiles) are not supported by CL procedures and programs.
- For database files, add a member to the file using the Add Physical File Member (ADDPFM) command or Add Logical File Member (ADDLFM) command. This is not required if a member was added by the CRTPF or CRTLF commands. The file must have a member when the procedure or program is processed, but does not need to have a member when the procedure or program is created.
- Refer to the file in the CL procedure using the DCLF command, and refer to the record format on the appropriate data manipulation CL commands in your CL source.
- Create the CL module.
- Create the program or service program.

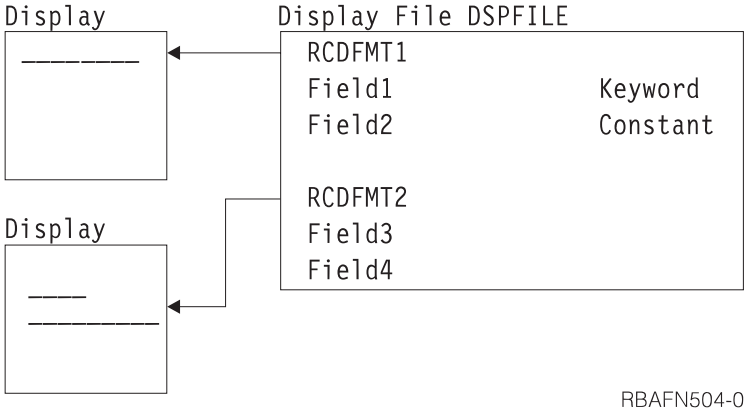
Up to five display or database files can be referred to in a CL procedure. The support for database files and display files is similar as the same commands are used. However, there are a few differences, which are described here.

- The following statements apply only to database files used with CL procedures and programs:
  - Only database files with a single record format may be used by a CL procedure or program.
  - The files may be either physical or logical files, and a logical file may be defined over multiple physical file members.
  - Only input operations, with the RCVF command, are allowed. The WAIT and DEV parameters on the RCVF command are not allowed for database files. In addition, the SNDF, SNDRCVF, and ENDRCV commands are not allowed for database files.
  - DDS is not required to create a physical file which is referred to in a CL procedure or program. If DDS is not used to create a physical file, the file has a record format with the same name as the file, and there is one field in the record format with the same name as the file, and with the same length as the record length of the file (RCDLEN parameter of the CRTPF command).
  - The file need not have a member when it is created for the module or program. It must, however, have a member when the file is processed by the program.
  - The file is opened for input only when the first RCVF command is processed. The file must exist and have a member at that time.
  - The file remains open until the procedure or OPM program returns or when the end of file is reached. When end of file is reached, message CPF0864 is sent to the CL procedure or program, and additional operations are not allowed for the file. The procedure or program should monitor for this message and take appropriate action when end of file is reached.
- The following statements apply only to display files used with CL procedures and programs:
  - Display files may have up to 99 record formats.
  - All data manipulation commands (SNDF, SNDRCVF, RCVF, ENDRCV and WAIT) are allowed for display files.
  - The display file must be defined with the DDS.
  - The display file is opened for both input and output when the first SNDF, SNDRCVF, or RCVF command is processed. The file remains open until the procedure or OPM program returns.

Note: The open does not occur for both types of files until the first send or receive occurs. Because of this, the file to be used can be created during the procedure or program and an override can be performed before the first send or receive. However, the file must exist before the module or program is compiled.

The format for the display is identified as a record format in DDS. Each record format may contain fields (input, output, and input/output), conditions/indicators, and constants. Several record formats can be entered in one display file. The display file name, record format name, and field names should be unique, because

other HLLs may require it, even though CL procedures and programs do not.

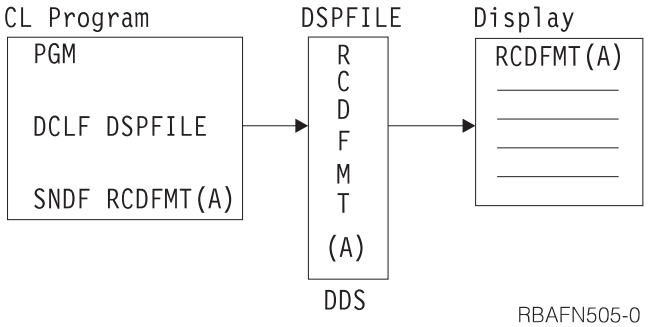


You can use the methods discussed in the Application Display Programming book or the Screen Design Aid (SDA) to enter DDS source for records and fields in the display file. See the

A CL procedure or program can use several commands, called data manipulation commands. These commands let you refer to a display file to send data to and receive data from device displays. These commands also allows you to refer to a database file to read records from a database file. These commands are:

- Declare File (DCLF). Defines a display or database file to be used in a procedure or program. The fields in the file are automatically declared as variables for use in the procedure or program.
- Send File (SNDF). Sends data to the display.
- Receive File (RCVF). Receives data from the display or database.
- Send/Receive File (SNDRCVF). Sends data to the display; then asks for input and, optionally, receives data from the display.
- Override with Display File (OVRDSPF). Allows a run-time override of a file used by a procedure or program with a display file.
- Override with Database File (OVRDBF). Allows a run-time override of a file used by a procedure or program with a database file.

These commands let a running program communicate with a device display using the display functions provided by DDS, and to read records from a database file. DDS provides functions for writing menus and performing basic application-oriented data requests that are characteristic of many CL applications.



The fields on the display or in the record are identified in the DDS for the file. In order for the CL procedure or program to use the fields, the file must be referred to in the CL procedure or program by the DCLF command. This reference causes the fields and indicators in the file to be declared automatically in your procedure

or program as variables. You can use these variables in any way in CL commands; however, their primary purpose is to send information to and receive information from a display. The DCLF command is not used at run time.

The format of the display and the options for the fields are specified in the device file and controlled through the use of indicators. Up to 99 indicator values can be used with DDS and CL support. Indicator variables are declared in your CL procedure or program in the form of logical variables with names &IN01 through &IN99 for each indicator that appears in the device file record formats referred to on the DCLF command. Indicators let you display fields and control data management display functions, and provide response information to your procedure or program from the device display. Indicators are not used with database files.

Referring to Files in a CL Procedure

Files are accessed during compiling of DCLF commands when CL modules and programs are created so that variables can be declared for each field in the file.

If you have qualified the name of the file at compile time, the file must be in that library at run time. If you have used the library list at compile time, the file must be in a library on the library list at run time.

Opening and Closing Files in a CL Procedure

When you use CL support, the file referred to is implicitly opened when you do your first send, receive, or send/receive operation. An opened display file remains open until the procedure or OPM program in which it was opened returns or transfers control. An opened database file is closed when end of file is reached, or when the procedure or OPM program in which it was opened returns or transfers control. Once a database file has been closed, it cannot be opened again during the same call of the procedure or OPM program.

When a database file opens, the first member in the file will open, unless you previously used an OVRDBF command to specify a different member (MBR parameter). If a procedure or OPM program ends because of an error, the files close. A file remains open until the procedure or OPM program in which that file was opened ends. Because of this, you have an easy way to share open data paths between running procedures and programs. You can open a file in one procedure or program. Then the file can share its open data path with another procedure or program under either of the following conditions:

- The file was created with or has been changed to have the SHARE(*YES) attribute.
- An override for that file by specifying SHARE(*YES) is in effect.

You can share files in this way between any two procedures or programs. Use online help for a detailed description of the function available when the system shares open data paths. Additionally, IBM provides a description of the SHARE parameter on the CRTDSPF, CRTPF, and CRTLF commands online. Refer to the *CL* section of the **Programming** category in the iSeries Information Center. A display file opened in a CL procedure or OPM program always opens for both input and output. A database file opened in a CL procedure or OPM program opens for input only.

Do not specify LVL(*CALLER) on the Reclaim Resources (RCLRSC) command in CL procedures and programs using files. If you specified LVL(*CALLER), all files

opened by the procedure or OPM program would be immediately closed, and any attempt to access the file would end abnormally.

Declaring a File

The Declare File (DCLF) command is used to declare a display or database file to your CL procedure or program. The DCLF command cannot be used to declare files such as tape, printer, and mixed files. You can declare up to five files in a CL procedure or OPM program. The DCLF command has the following parameters:

```
DCLF   FILE(library-name/file-name)
       RCDFMT(record-format-names)
       OPNID(open_id_name)
```

Note that the file must exist before the module or program is compiled.

If you are using a display file in your procedure or program, you may have to specify input and output fields in your DDS. These fields are handled as variables in the procedure or program. When processing a DCLF command, the CL compiler declares CL variables for each field and option indicator in each record format in the file. For a field, the CL variable name is the field name preceded by an ampersand (&). For an option indicator, the CL variable name is the indicator that is preceded by &IN.

You use the open file identifier (OPNID) parameter to uniquely identify an instance of a declared file so that multiple files can be declared. If you use the OPNID parameter, then for a field, the CL variable name is the field name preceded by an ampersand (&), the OPNID value, and an underscore (_). For an option indicator, the CL variable name is the indicator preceded by an ampersand (&), the OPNID value, an underscore, and “IN”.

For example, if a field named INPUT and indicator 10 are defined in DDS, the DCLF command automatically declares them as &INPUT and &IN10. This declaration is performed when the CL module or program is compiled. Up to 50 record format names can be specified on one command, but none can be variables. Only one record format may be specified for a database file.

If the following DDS were used to create display file CNTRLDSP in library MCGANN:

```
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
     A          R MASTER
     A                                        CA01(01 'F1 RESPONSE')
     A            TEXT         300      2  4
     A            RESPONSE      15   1  8  4 BLINK
     A
     A
```

Three variables, &IN01, &TEXT, and &RESPONSE, would be available from the display file. In a CL procedure referring to this display file, you would enter only the DCLF source statement:

```
DCLF MCGANN/CNTRLDSP
```

The compiler will expand this statement to individually declare all display file variables. The expanded declaration in the compiler list looks like this:

```
  •
  •
  •
  00500- DCLF(MCGANN/CNTRLDSP)
 04/02/03
         QUALIFIED FILE NAME - MCGANN/CNTRLDSP
            RECORD FORMAT NAME - MASTER
              CL VARIABLE               TYPE     LENGTH     PRECISION (IF *DEC)
               &IN01                    *LGL        1
               &TEXT                    *CHAR     300
               &RESPONSE                *CHAR      15
  •
  •
  •
```

If the DCLF source statement includes an OPNID parameter:

```
DCLF MCGANN/CNTRLDSP OPNID(OPENID1)
```

The expanded declaration in the compiler list will look like this:

```
  •
  •
  •
  00500- DCLF FILE(MCGANN/CNTRLDSP) OPNID(OPENID1)                   04/02/03
         QUALIFIED FILE NAME - MCGANN/CNTRLDSP
            RECORD FORMAT NAME - MASTER
              CL VARIABLE               TYPE     LENGTH     PRECISION     TEXT
               &OPENID1_IN01            *LGL        1
               &OPENID1_TEXT            *CHAR     300
               &OPENID1_RESPONSE        *CHAR      15
  •
  •
  •
```

Sending and Receiving Data with a Display File

The only commands you can use with a display file to send or receive data in CL procedures and programs are the SNDF, RCVF, and SNDRCVF commands.

The system formats the content of the variables associated with the output or output/input fields in the record format when you run a SNDF command. Additionally the system sends it to the display device. This is similar to when you run a RCVF command. The values of the fields associated with input or output/input fields in the record format on the display are placed in the corresponding CL variables.

The SNDRCVF command sends the contents of the CL variables to the display. The command then performs the equivalent of a RCVF command to obtain the updated fields from the display. Note that CL does not support zoned decimal numbers. Consequently, fields in the display file that are defined as zoned decimal, cause *DEC fields to be defined in the CL procedure or program. *DEC fields are internally supported as packed decimal, and the CL commands convert the packed and zoned data types as required. Fields that overlap in the display file because of coincident display positions result in separately defined CL variables that do not overlap. You cannot use record formats that contain floating point data in a CL procedure or program.

Note: If a SNDRCVF or RCVF command for a workstation indicates WAIT(*NO), then the system uses the WAIT command to receive data. The same is true if a SNDF command is issued using a record format containing the INVITE DDS keyword.

Except for message subfiles, any attempt to send or receive subfile records causes run-time errors. Most other functions specified for display files in DDS are available; some functions (such as using variable starting line numbers) are not. For more information on messages and subfiles in CL procedures and programs, see Chapter 8.

The following example shows the steps required to create a typical operator menu and to send and receive data using the SNDRCVF command. The menu looks like this:

```
Operator Menu

  1. Accounts Payable
  2. Accounts Receivable
  90. Signoff

Option:
```

First, enter the following DDS source. The record format is MENU, and OPTION is an input-capable field. The OPTION field uses DSPATR(MDT). This causes the system to check this field for valid values even if the operator does not enter anything.

```
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
     A          R MENU
     A                                  1  2'Operator Menu'
     A                                  3  4'1. Accounts Payable'
     A                                  5  4'2. Accounts Receivable'
     A                                  5  4'90. Signoff'
     A                                  7  2'Option'
     A            OPTION         2Y 01   + 2VALUES(1 2 90) DSPATR(MDT)
     A
     A
```

Enter the CRTDSPF command to create the display file. In CL programming, the display file name (INTMENU) can be the same as the record format name (MENU), though this is not true for some other languages, like RPG for OS/400.

The display file could also be created using the Screen Design Aid (SDA) utility.

Next, enter the CL source to run the menu.

The CL source for this menu is:

```
         PGM /* OPERATOR MENU */
         DCLF INTMENU
BEGIN:   SNDRCVF RCDFMT(MENU)
         IF COND(&OPTION *EQ 1) THEN(CALL ACTSPAYMNU)
         IF COND(&OPTION *EQ 2) THEN(CALL ACTSRCVMNU)
         IF COND(&OPTION *EQ 90) THEN(SIGNOFF)
         GOTO BEGIN
         ENDPGM
```

When this source is compiled, the DCLF command automatically declares the input field OPTION in the procedure as a CL variable.

The SNDRCVF command defaults to WAIT(*YES); that is, the program waits until input is received by the program.

Writing a CL Program to Control a Menu

The following example shows how a CL procedure can be written to display and control a menu. See the Application Display Programming book for another method of creating and controlling menus.

This example shows a CL procedure, ORD040C, that controls the displaying of the order department general menu and determines which HLL procedure to call based on the option selected from the menu. The procedure shows the menu at the display station.

The order department general menu looks like this:

```
 Order Dept General Menu

  1 Inquire into customer file
  2 Inquire into item file
  3 Customer name search
  4 Inquire into orders for a customer
  5 Inquire into an existing order
  6 Order entry
 98 End of menu

Option:
```

The DDS for the display file ORD040C looks like this:

```
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
     A* MENU ORDO4OCD ORDER DEPT GENERAL MENU
     A
     A          R MENU                      TEXT('General Menu')
     A                                    1  2'Order Dept General Menu'
     A                                    3  3'1 Inquire into customer file'
     A                                    4  3'2 Inquire into item file'
     A                                    5  3'3 Customer name search'
     A                                    6  3'4 Inquire into orders for a custom+
     A                                       er'
     A                                    7  3'5 Inquire into existing order'
     A                                    8  3'6 Order Entry'
     A                                    9  2'98 End of menu'
     A                                   11  2'Option'
     A            RESP           2Y001 11 10VALUES(1 2 3 4 5 6 98)
     A                                        DSPATR(MDT)
     A
     A
```

The source procedure for ORD040C looks like this:

```
|
|       PGM /* ORD040C Order Dept General Menu */
|       DCLF FILE(ORD040CD)
| START: SNDRCVF RCDFMT(MENU)
|       SELECT
```

```
|                   WHEN (&RESP=1) THEN(CALLPRC CUS210)  /* Customer inquiry */
|                   WHEN (&RESP=2) THEN(CALLPRC ITM210)  /* Item inquiry     */
|                   WHEN (&RESP=3) THEN(CALLPRC CUS220)  /* Cust name search */
|                   WHEN (&RESP=4) THEN(CALLPRC ORD215)  /* Orders by cust   */
|                   WHEN (&RESP=5) THEN(CALLPRC ORD220)  /* Existing order   */
|                   WHEN (&RESP=6) THEN(CALLPRC ORD410C) /* Order entry      */
|                   WHEN (&RESP=98) THEN(RETURN)         /* End of Menu      */
|                 ENDSELECT
|                 GOTO START
|                 ENDPGM
```

The DCLF command indicates which file contains the field attributes the system needs to format the order department general menu when the SNDRCVF command is processed. The system automatically declares a variable for each field in the record format in the specified file if that record format is used in an SNDF, RCVF, or SNDRCVF command. The variable name for each field automatically declared is an ampersand (&) followed by the field name. For example, the variable name of the response field RESP in ORD040C is &RESP.

Other notes on the operation of this menu:

The SNDRCVF command is used to send the menu to the display and to receive the option selected from the display.

If the option selected from the menu is 98, ORD040C returns to the procedure that called it.

The ELSE statements are necessary to process the responses as mutually exclusive alternatives.

Note: This menu is run using the CALL command. See the Application Display Programming book for a discussion of those menus run using the GO command.

Overriding Display Files in a CL Procedure

You can use the Override with Display File (OVRDSPF) command to replace the display file named in a CL procedure or program or to change certain parameters of the existing display file. This may be especially useful for files that have been renamed or moved since the module or program was compiled.

The initial parameters of the OVRDSPF command are:

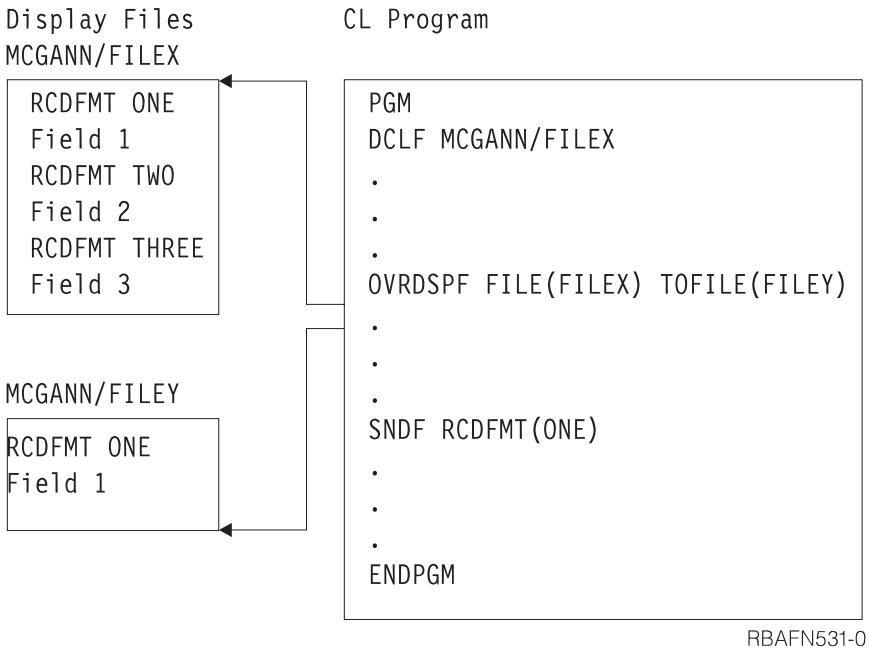
```
OVRDSPF   FILE(overridden-file-name) TOFILE(new-file-name)
          DEV(device-name)
```

The OVRDSPF command is valid for a file referred to by a CL procedure or program only if the file specified on the DCLF command was a display file when the module or program was created. The file used when the program is run must be of the same type as the file referred to when the module or program was created.

You must run the OVRDSPF command before opening the file that is being overridden. An open is caused by the first use of a send or receive command. The system overrides the file on finding any of the following conditions:

- A procedure or program that contains the OVRDSPF command opens the file.
- The file opens in another procedure which transfers control by using the CALLPRC command.
- The file opens in another program which transfers control by using the CALL command.

When you override to a different file, only those record format names referred to on the SNDF, RCVF, or SNDRCVF command need to be in the overriding file. In the following illustration, display file FILEY does not need record format TWO or THREE.



You should make sure that the record format referred to names of the original file and the overriding files have the same field definitions and indicator names in the same order. You may get unexpected results if you specify LVLCHK(*NO).

Another consideration has to do with the DEV parameter on the SNDF, RCVF, and SNDRCVF commands when an OVRDSPF command is applied. If *FILE is specified on the DEV parameter of the RCVF, SNDF, or SNDRCVF command, the system automatically directs the operation to the correct device for the overridden file. If a specific device is specified on the DEV keyword of the RCVF, SNDF, or SNDRCVF command, one of the following may occur:

- If a single device display file is being used, an error will occur if the display file is overridden to a device other than the one specified on the RCVF, SNDF, or SNDRCVF command.
- If a multiple device display file is being used, an error will occur if the device specified on the RCVF, SNDF, or SNDRCVF command is not among those specified on the OVRDSPF command.

Working with Multiple Device Display Files

The normal mode of operation on a system is for the workstation user to sign on and become the requester for an interactive job. Many users can do this at the same time, because each will use a logical copy of the procedure, including the display file in the procedure. Each requester calls a separate job in this kind of use. This is not considered to be multiple device display use.

A multiple device display configuration occurs when a single job called by one requester communicates with multiple display stations through one display file. While only one display file can be handled by a CL procedure, the display file, or different record formats within it, can be sent to several device displays. Commands used primarily with multiple device display files are:

- End Receive (ENDRCV). This command cancels requests for input that have not been satisfied.
- Wait (WAIT). Accepts input from any device display from which user data was requested by one or more previous RCVF or SNDRCVF commands when WAIT(*NO) was specified on the command, or by one or more previous SNDF commands to a record format containing the INVITE DDS keyword.

If you use a multiple device display file, the device names must be specified on the DEV parameter on the CRTDSPF command when the display file is created, on the CHGDSPF command when the display file is changed, or on an override command, and the number of devices must be less than or equal to the number specified on the MAXDEV parameter on the CRTDSPF command.

Multiple device display configurations affect the SNDRCVF and the RCVF commands and you may need to use the WAIT or ENDRCV commands. When an RCVF or SNDRCVF command is used with multiple display devices, the default value WAIT(*YES) prevents further processing until an input-capable field is returned to the program from the device named on the DEV parameter. Because the response may be delayed, it is sometimes useful to specify WAIT(*NO), thus letting your procedure or program continue running other commands before the receive operation is satisfied.

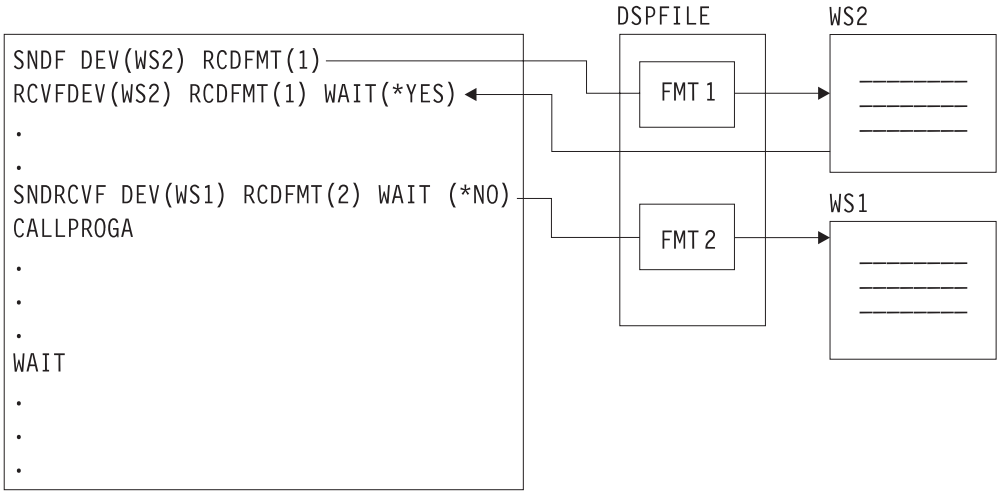
If you use an RCVF or SNDRCVF command and specify WAIT(*NO), the CL procedure or program continues running until a WAIT command is processed.

Using a SNDF command with a record format which has the DDS INVITE keyword is equivalent to using a SNDRCVF command with WAIT(*NO) specified. The DDS INVITE keyword is ignored for SNDRCVF and RCVF commands.

The WAIT command must be issued to access a data record. If no data is available, processing is suspended until data is received from a device display or until the time limit specified in the WAITRCD parameter for the display file on the CRTDSPF, CHGDSPF, or OVRDSPF commands has passed. If the time limit passes, message CPF0889 is issued.

The WAIT will also be satisfied by the job being canceled with the controlled option on the ENDJOB, ENDSYS, PWRDWNSYS, and ENDSBS commands. In this case, message CPF0888 is issued and no data is returned. If a WAIT command is issued without a preceding receive request (such as RCVF . . . WAIT(*NO)), a processing error occurs.

A typical multiple device display configuration (with code) might look like this:



RBAFN506-0

In the above example, the first two commands show a typical sequence in which the default is taken; processing waits for the receive operation from WS2 to complete. Because WS2 is specified on the DEV parameter, the RCVF command does not run until WS2 responds, even if prior outstanding requests (not shown) from other stations are satisfied.

The SNDRCVF command, however, has WAIT(*NO) specified and so does not wait for a response from WS1. Instead, processing continues and PROGA is called. Processing then stops at the WAIT command until an outstanding request is satisfied by a workstation, or until the function reaches time-out.

The WAIT command has the following format:

```
WAIT DEV(CL-variable-name)
```

If the DEV parameter is specified, the CL variable name is the name of the device that responded. (The default is *NONE.) If there are several receive requests (such as RCVF. . . WAIT(*NO)), this variable takes the name of the first device to respond after the WAIT command is encountered and processing continues. The data received is placed in the variable associated with the field in the device display.

A RCVF command with WAIT(*YES) specified can be used to wait for data from a specific device. The same record format name must be specified for both the operation that started the receive request and the RCVF command.

In some cases, several receive requests are outstanding, but processing cannot proceed further without a reply from a specific device display. In the following example, three commands specify WAIT(*NO), but processing cannot continue at label LOOP until WS3 replies:

```
PGM
.
.
.
SNDF DEV(WS1) RCDFMT(ONE)
SNDF DEV(WS2) RCDFMT(TWO)
SNDRCVF DEV(WS3) RCDFMT(THREE) WAIT(*NO)
RCVF DEV(WS2) RCDFMT(TWO) WAIT(*NO)
RCVF DEV(WS1) RCDFMT(ONE) WAIT(*NO)
CALL...
```

```
         CALL...
         .
         .
         RCVF DEV(WS3) RCDFMT(THREE) WAIT(*YES)
LOOP:    WAIT DEV(&WSNAME)
         MONMSG CPF0882 EXEC(GOTO REPLY)
         .
         .
         .
         GOTO LOOP
REPLY:   CALL...
         .
         .
         .
         ENDPGM
```

CL procedures and programs also support the ENDRCV command, which lets you cancel a request for input that has not yet been satisfied. A SNDF or SNDRCVF command will also cancel a request for input that has not yet been satisfied. However, if the data was available at the time the SNDF or SNDRCVF command was processed, message CPF0887 is sent. In this case the data must be received with the WAIT command or RCVF command, or the request must be explicitly canceled with a ENDRCV command before the SNDF or SNDRCVF command can be processed.

Receiving Data from a Database File

The only command you can use to receive data from a database file is the RCVF command.

When you run a RCVF command, the next record on the file's access path is read, and the values of the fields defined in the database record format are placed in the corresponding CL variables. Note that CL does not support zoned decimal or binary numbers. Consequently, fields in the database file defined as zoned decimal or binary cause *DEC fields to be defined in the CL procedure or program. *DEC fields are internally supported as packed decimal, and the RCVF command performs the conversion from zoned decimal and binary to packed decimal as required. Database files which contain floating point data cannot be used in a CL procedure or program.

When the end of file is reached, message CPF0864 is sent to the procedure or OPM program. The CL variables declared for the record format are not changed by the processing of the RCVF command when this message is sent. You should monitor for this message and perform the appropriate action for end of file. If you attempt to run additional RCVF commands after end of file has been reached, message CPF0864 is sent again.

Overriding Database Files in a CL Procedure or Program

You can use the Override with Database File (OVRDBF) command to replace the database file named in a CL procedure or program or to change certain parameters of the existing database file. This may be especially useful for files that have been renamed or moved since the procedure or program was created. It can also be used to access a file member other than the first member.

The initial parameters of the OVRDBF command are:

```
OVRDBF FILE(overridden-file-name) TOFILE(new-file-name)
       MBR(member-name)
```

The OVRDBF command is valid for a file referred to by a CL procedure or program only if the file specified in the DCLF command was a database file when the module or program was created. The file used when the program was processed must be of the same type as the file referred to when the module or program was created.

The OVRDBF command must be processed before the file to be overridden is opened for use (an open occurs by the first use of the RCVF command). The file is overridden if it is opened in the procedure or OPM program containing the OVRDBF command, or if it is opened in another program to which control is transferred by the CALL command, or if it is opened in another procedure to which control is transferred using the CALLPRC command.

When you override to a different file, the overriding file must have only one record format. A logical file which has multiple record formats defined in DDS may be used if it is defined over only one physical file member. A logical file which has only one record format defined in the DDS may be defined over more than one physical file member. The name of the format does not have to be the same as the format name referred to when the program was created. You should ensure that the format of the data in the overriding file is the same as in the original file. You may get unexpected results if you specify LVLCHK(*NO).

Referring to Output Files from Display Commands

A number of the IBM display commands allow you to place the output from the command into a database file (OUTFILE parameter). The data in this file can be received directly into a CL procedure or program and processed.

The following CL procedure accepts two parameters, a user name and a library name. The procedure determines the names of all programs, files, and data areas in the library and grants normal authority to the specified users.

```
     PGM PARM(&USER &LIB)
     DCL &USER *CHAR 10
     DCL &LIB *CHAR 10
(1)  DCLF QSYS/QADSPOBJ
(2)  DSPOBJD OBJ(&LIB/*ALL) OBJTYPE(*FILE *PGM *DTAARA) +
     OUTPUT(*OUTFILE) OUTFILE(QTEMP/DSPOBJD)
(3)  OVRDBF QADSPOBJ TOFILE(QTEMP/DSPOBJD)
(4)  READ:  RCVF
(5)  MONMSG CPF0864 EXEC(RETURN)  /* EXIT WHEN END OF FILE REACHED  */
(6)  GRTOBJAUT OBJ(&ODLBNM/&ODOBNM) OBJTYPE(&ODOBTP) +
     USER(&USER) AUT(*CHANGE)
     GOTO READ                    /*GO BACK FOR NEXT RECORD*/
     ENDPGM
```

(1) The declared file, QADSPOBJ in QSYS, is the IBM-supplied file that is used by the DSPOBJD command. This file is the primary file which is referred to by the command when creating the output file. It is referred to by the CL compiler to determine the format of the records and to declare variables for the fields in the record format.

(2) The DSPOBJD command creates a file named DSPOBJD in library QTEMP. This file has the same format as file QADSPOBJ.

(3) The OVRDBF command overrides the declared file (QADSPOBJ) to the file created by the DSPOBJD command.

(4) The RCVF command reads a record from the DSPOBJD file. The values of the fields in the record are copied into the corresponding CL variables, which were implicitly declared by the DCLF command. Because the

OVRDBF command was used, the file QTEMP/DSPOBJD is read instead of QSYS/QADSPOBJ (the file QSYS/QADSPOBJ is not read).

(5) Message CPF0864 is monitored. This indicates that the end of file has been reached, so the procedure returns control to the calling procedure.

(6) The GRTOBJAUT command is processed, using the variables for object name, library name and object type, which were read by the RCVF command.

Chapter 6. Advanced Programming Topics

This chapter introduces more advanced programming topics, including:

- Special functions that can be called from high-level language programs (including CL programs)
- Using prompting and the Programmer Menu to enter program source

See the *CL* section of the **Programming** category of the iSeries Information Center for information on advanced function command processing.

Several sample programs are included at the end of the chapter.

Using the QCAPCMD Program

The Process Commands (QCAPCMD) API performs command analyzer processing on command strings. You can use this API to do the following:

- Check the syntax of a command string prior to running it.
- Prompt the command and receive the changed command string.
- Use a command from a high-level language.
- Display the help for a command.

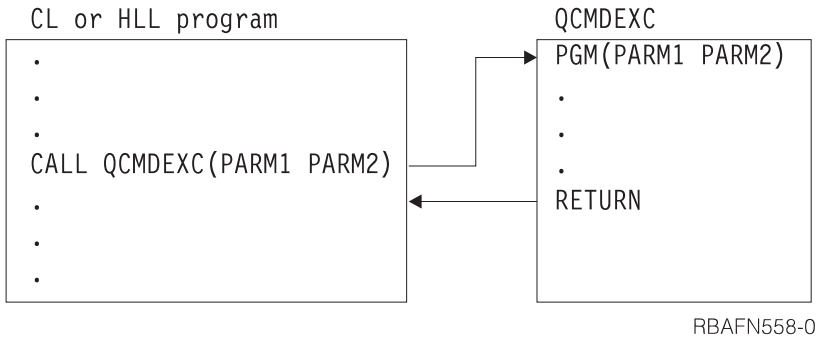
See the *APIs* section of the **Programming** category of the iSeries Information Center for information on the QCAPCMD API.

Using the QCMDEXC Program

Execute Command (QCMDEXC) is an IBM-supplied program that runs a single command. This command is used to activate another command:

- From within a high-level language (HLL) program.
- From within a CL procedure.
- From a program where it is not known at compile time what command is to be run or what parameters are to be used.

The QCMDEXC program is called from within the HLL or CL procedure or program. The command that is to be run is passed as a parameter on the CALL command.



After the command runs, control returns to your HLL or CL procedure or program.

© Copyright IBM Corp. 1997, 2004

The command runs as if it was not in a program. Therefore, variables cannot be used on the command because values cannot be returned by the command to CL variables. Additionally, commands that can only be used in CL procedures or programs cannot be run by the QCMDEXC program. The format of the call to the QCMDEXC program is the following:

```
CALL PGM(QCMDEXC) PARM(command command-length)
```

Enter the command you wish to run as a character string on the first parameter. You must specify the command library.

```
CALL PGM(QCMDEXC )  PARM('QSYS/CRTLIB LIB(TEST)' 22)
```

Remember that you must enclose the command in apostrophes if it contains blanks. The maximum length of the character string is 6000 characters; never count the delimiters (the apostrophes) as part of the string. The length that is specified as the second value on the PARM parameter is the length of the character string that is passed as the command. Length must be a packed decimal value of length 15 with 5 decimal positions.

Thus, to replace a library list, the call to the QCMDEXC program would look like this:

```
CALL PGM(QCMDEXC) PARM('CHGLIBL LIBL(QGPL NEWLIB QTEMP)' 31)
```

It is possible to code this statement into the HLL or CL program to replace the library list when the program runs. The QCMDEXC program does not provide run-time flexibility when used this way.

Providing run-time flexibility is accomplished by:

1. Substituting variables for the constants in the parameter list, and
2. Specifying the values for the variables in the call to the HLL or CL program.

For instance:

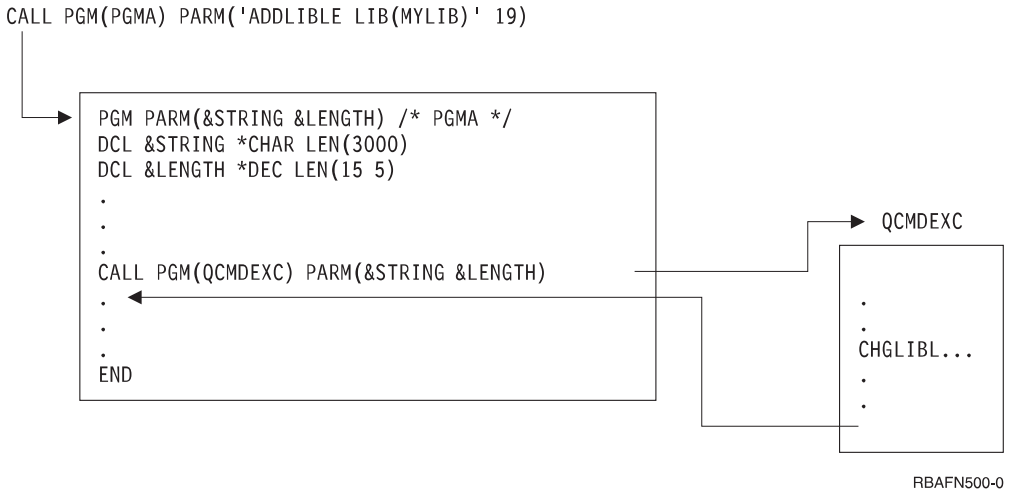


Figure 2. Example of Call PGM

The command length, passed to the QCMDEXC program on the second parameter, is the maximum length of the passed command string. Should the command string be passed as a quoted string, the command length is exactly the length of the quoted string. Should the command string be passed in a variable, the command

length is the length of the CL variable. It is not necessary to reduce the command length to the actual length of the command string in the variable, although it is permissible to do so.

Not all commands can be run using the QCMDEXC program. The command passed on a call to the QCMDEXC program must be valid within the current environment (interactive or batch) in which the call is being made. The command cannot be one of the following:

- An input stream control command (BCHJOB, ENDBCHJOB, and DATA)
- A command that can be used only in a CL program

Refer to the command documentation in the *CL* section of the **Programming** category in the iSeries Information Center for information to help you determine if you can pass a CL command on a call to the QCMDEXC program. To find the environments in which you can run the command, look at the 'Where allowed to run' value located at the beginning of the Information Center documentation file for the CL command. You can also use the Display Command (DSPCMD) command to see where the command can be used.

You can precede the CL command with a question mark (?) to request prompting or use selective prompting when you call QCMDEXC in an interactive job.

If an error is detected while a command is being processed through the QCMDEXC program, an escape message is sent. You can monitor for this escape message in your CL procedure or program using the Monitor Message (MONMSG) command. For more information about monitoring for messages, see Chapter 7 and Chapter 8.

If a syntax error is detected, message CPF0006 is sent. If an error is detected during the processing of a command, any escape message sent by the command is returned by the QCMDEXC program. You monitor for messages from commands run through the QCMDEXC program in the same way you monitor for messages from commands contained in CL procedures and programs.

See the appropriate high level languagereference book for information on how high level language programs handle errors on calls.

Using the QCMDEXC Program with DBCS Data

You can use QCMDEXC to request double-byte character set (DBCS) input data to be entered with a command. The command format used for QCMDEXC to prompt double-byte data is:

```
CALL QCMDEXC ('command' command-length IGC)
```

The third parameter of the QCMDEXC program, IGC, instructs the system to accept double-byte data. For example, the following CL program asks a user to provide double-byte text for a message. Then the system sends the following message:

```
PGM
   CALL QCMDEXC ('?SNDMSG' 7 IGC)
ENDPGM
```

An explanation of the system message follows:

- The ? character instructs the system to present the command prompt for the Send Message (SNDMSG) command.

- The value 7 is the length of the SNDMSG command plus the question mark.
- The value IGC allows you to request double-byte data.

The following display is shown after running the QCMDEXC program. You can use double-byte conversion on this display:

```
                         SEND MESSAGE (SNDMSG)

TYPE CHOICES, PRESS ENTER.

MESSAGE TEXT . . . . . . . . . . . .  ____________________________________
____________________________________________________________________________
____________________________________________________________________________
____________________________________________________________________________
____________________________________________________________________________
____________________________________________________________________________
____________________________________________________________________
TO USER PROFILE . . . . . . . . . .  __________  NAME, *SYSOPR, *ALLACT...






                                                                      BOTTOM
 F3=EXIT   F4=PROMPT   F5=REFRESH   F10=ADDITIONAL PARAMETERS  F12=CANCEL
 F13=HOW TO USE THIS DISPLAY   F24=MORE KEYS
```

Using the QCMDCHK Program

QCMDCHK is an IBM-supplied program that performs syntax checking for a single command, and optionally prompts for the command. The command is not run. If prompting is requested, the command string is returned to the calling procedure or program with the updated values as entered through prompting. The QCMDCHK program can be called from a CL procedure or program or an HLL procedure or program.

Typical uses of QCMDCHK are:

- Prompt the user for a command and then store the command for later processing.
- Determine the options the user specified.
- Log the processed command. First, prompt with QCMDCHK, run with QCMDEXC, and then log the processed command.

The format of the call to QCMDCHK is:

```
CALL PGM(QCMDCHK) PARM(command  command-length)
```

The first parameter passed to QCMDCHK is a character string containing the command to be checked or prompted. If the first parameter is a variable and prompting is requested, the command entered by the workstation user is placed in the variable.

The second parameter is the maximum length of the command string being passed. If the command string is passed as a quoted string, the command length is exactly the length of the quoted string. If the command string is passed in a variable, the command length is the length of the CL variable. The second parameter must be a packed decimal value of length 15 with 5 decimal positions.

The QCMDCHK program performs syntax checking on the command string which is passed to it. It verifies that all required parameters are coded, and that all parameters have allowable values. It does not check for the processing environment. That is, a command can be checked whether it is allowed in batch only, interactive only, or only in a batch or interactive CL program. QCMDCHK does not allow checking of command definition statements.

If a syntax error is detected on the command, message CPF0006 is sent. You can monitor for this message to determine if an error occurred on the command. Message CPF0006 is preceded by one or more diagnostic messages that identify the error. In the following example, control is passed to the label ERROR within the program, because the value 123 is not valid for the PGM parameter of the CRTCLPGM command.

```
CALL QCMDCHK ('CRTCLPGM PGM(QGPL/123)' 22)
MONMSG CPF0006 EXEC(GOTO ERROR)
```

You can request prompting for the command by either placing a question mark before the command name or by placing selective prompt characters before one or more keyword names in the command string.

If no errors are detected during checking and prompting for the command, the updated command string is placed in the variable specified for the first parameter. The prompt request characters are removed from the command string. This is shown in the following example:

```
DCL &CMD *CHAR 2000
 .
 .
CHGVAR &CMD '?CRTCLPGM'
CALL QCMDCHK (&CMD 2000)
```

After the call to the QCMDCHK program is run, variable &CMD contains the command string with all values entered through the prompter. This might be something like:

```
CRTCLPGM PGM(PGMA) SRCFILE(TESTLIB/SOURCE) USRPRF(*OWNER)
```

Note that the question mark preceding the command name is removed.

When prompting is requested through the QCMDCHK program, the command string should be passed in a CL variable. Otherwise, the updated command string is not returned to your procedure or program. You must also be sure that the variable for the command string is long enough to contain the updated command string which is returned from the prompter. If it is not long enough, message CPF0005 is sent, and the variable containing the command string is not changed. Without selective prompting, the prompter only returns entries that were typed by the user.

The length of the variable is determined by the value of the second parameter, and not the actual length of the variable. In the following example, escape message CPF0005 is sent because the specified length is too short to contain the updated command, even though the variable was declared with an adequate length.

```
DCL &CMD *CHAR 2000
 .
 .
CHGVAR &CMD '?CRTCLPGM'
CALL QCMDCHK (&CMD 9)
```

If you press F3 or F12 to exit from the prompter while running QCMDCHK, message CPF6801 is sent to the procedure or program that called QCMDCHK, and the variable containing the command string is not changed.

If PASSATR(*YES) is specified on the PARM, ELEM, or QUAL command definition statement, and the default value is changed using the CHGCMDDFT command, the default value is highlighted as though this was a user-specified value and not a default value. If a default value of a changed PARM, ELEM, or QUAL command definition statement is changed back to its original default value, the default value will no longer be highlighted.

Using Message Subfiles in a CL Program or Procedure

In CL procedures and programs, message subfiles are the only type of subfiles supported. To use subfile message support, run a SNDF or SNDRCVF command using the subfile message control record. In the DDS, supply SFLPGMQ data and always have SFLINZ active.

When you use message subfiles in CL procedures and programs, you must name a procedure or program. You cannot specify an * for the SFLPGMQ keyword in DDS. When you specify a procedure or OPM program name, all messages sent to that procedure's or program's message queue are taken from the invocation message queue and placed in the message subfile. All messages associated with the current request are taken from the CALL message queue and placed in the message subfile.

Message subfiles let a controlling procedure or program display one or more error messages.

Allowing User Changes to CL Commands at Run Time

With most CL procedures and programs, the workstation user provides input to the procedure or program by specifying parameter values passed to the procedure or program or by typing into input-capable fields on a display prompt.

You can also prompt the workstation user for input to a CL procedure or program in the following ways:

- If you enter a ? before the CL command in the CL procedure or program source, the system displays a prompt for the CL command. Parameter values you have already specified in your procedure or program are filled in and cannot be changed by the workstation user. See "Using the OS/400 Prompter within a CL Procedure or Program" later in this section.
  - If you call the QCMDEXC program and request selective prompting, the system displays a prompt for a CL command, but you need not specify in the CL program source which CL command is to be used at processing time. For more information on the QCMDEXC program, see "Using the QCMDEXC Program" on page 163.

Using the OS/400 Prompter within a CL Procedure or Program

You can request prompting within the interactive processing of a CL procedure or program. For example, the following procedure can be compiled and run:

```
PGM
.
.
```

```
.
?DSPLIB
.
.
.
ENDPGM
```

In this case, the prompt for the Display Library (DSPLIB) command appears on the display during processing of the program. Processing of the DSPLIB command waits until you have entered values for required parameters and pressed the Enter key.

Any values specified in the source procedure cannot be changed directly by the operator (or user). For example:

```
PGM
.
.
.
?SNDMSG TOMSGQ(WS01 WS02)
.
.
.
ENDPGM
```

When the procedure is called and the prompt for the Send Message (SNDMSG) command appears, the operator (or user) can enter values on the MSG, MSGTYPE, and RPYMSGQ parameters, but cannot alter the values on the TOMSGQ parameter. For example, the operator (or user) cannot add WS03 or delete WS02. See "QCMDEXC with Prompting in CL Procedures and Programs" on page 173 for an exception to this restriction. The following restrictions apply to the use of the prompter within a CL procedure at processing time:

- When the prompter is called from a CL procedure or program, you cannot enter a variable name or an expression for a parameter value on the prompt.
- Prompting cannot be requested on a command embedded on an IF, ELSE, or MONMSG command:

  ```
  Correct                              Incorrect

  IF (&A=5) THEN(DO)       IF (&A=5) THEN(?SNDMSG)
                ?SNDMSG
                ENDDO
  ```

- Prompting *cannot* be used for the following commands:

CALL	**CALLPRC**	**CHGVAR**	**COPYRIGHT**
DCL	**DCLF**	**DO**	**DOFOR**
DOUNTIL	**DOWHILE**	**ELSE**	**ENDDO**
ENDPGM	**ENDRCV**	**ENDSELECT**	**GOTO**
IF	**ITERATE**	**LEAVE**	**MONMSG**
OTHERWISE	**PGM**	**RCVF**	**RETURN**
SELECT	**SNDF**	**SNDRCVF**	**WAIT**
WHEN			

- Prompting cannot be used in batch jobs.

When you enter a prompting request (?) on a command in a CL source file member, you may receive a diagnostic message on the command and still have a successful compilation. In this case, you must examine the messages carefully to see that the errors can be corrected by values entered through the prompt display when the procedure or program runs.

You can prompt for all commands you are authorized to in any mode while in an interactive environment except for the previously listed commands, which cannot be prompted for during processing of a CL procedure or program. This allows you to prompt for any command while at a workstation and reduces the need to refer to the manuals that describe the various commands and their parameters.

If you press F3 or F12 to cancel the prompted command while running that command, an escape message (CPF6801) is sent to the CL procedure or program. You can monitor for this message using the MONMSG command in the CL procedure or program.

When you prompt for a command, your procedure or program does not receive the command string you entered. To achieve this, prompt using QCMDCHK, then run the command using QCMDEXC. You can also use QCAPCMD to prompt and run the command.

Selective Prompting for CL Commands

You can request to prompt for selected parameters within a command. This is especially helpful when you are using some of the longer commands and do not want to be prompted for certain parameters.

Selective prompting can be used during interactive prompting or entered as source (in SEU) for use within a CL procedure or program. You can enter the source for selective prompting with SEU but you cannot use selective prompting while entering commands in SEU.

You can use selective prompting to:

- Select the parameters for which prompting is needed.
- Determine which parameters are protected.
- Omit parameters from the prompt.

The following restrictions apply to selective prompting:

- The command name or label must be preceded by a ? (question mark):
  - When one or more of the selective prompt options is ?- (question mark, minus).
  - To avoid getting a CPF6805 message (a message that indicates a diagnostic problem on the command although compilation is successful)
- Parameters can be specified by position but they cannot be preceded by selective prompt characters.
- A parameter must be in keyword form to be selectively prompted for.
- Blanks cannot be entered between the selective prompt characters and the keyword.
- Selective prompting is only applicable at a parameter level; that is, you cannot specify particular keyword values within a list of values.
- ?- is not allowed in prompt override programs.
- If a parameter is required, the ?? selective prompt must be used.

  You can tell that a parameter is required because the input slot is highlighted when the command is prompted.

User-specified values are marked with a special symbol (>) in front of the values in both selective and regular prompting. If a user-specified value on the parameter prompt is not preceded by this symbol, the command default is passed to the command processing program.

If PASSATR(*YES) is specified on the PARM, ELEM, or QUAL command definition statement, and the default value is changed using the CHGCMDDFT command, the default value is shown as a user-specified value (using the > symbol) and not a default value. If a default value of a changed PARM, ELEM, or QUAL command definition statement is changed back to its original default value, the > symbol is removed.

You can press F5 while you are using selective prompting to again display those values initially shown on the display.

If a CL variable is used to specify a value for a parameter which is to be displayed through selective prompting, you can change the value on the prompt, and the changed value is used when the command is run. The value of the variable in the procedure or program is not changed. If a CL procedure contains the following:

```
OVRDBF ?*FILE(FILEA) ??TOFILE(&FILENAME) ??MBR(MBR1)
```

the three parameters, FILE, TOFILE, and MBR is shown on the prompt display. The value specified for the FILE parameter cannot be changed by you, but the values for the TOFILE and MBR parameters can be changed. Assume that the CL variable &FILENAME has a value of FILE1, and you change it to FILE2. When the command is run, the value of FILE2 is used, but the value of &FILENAME is not changed in the procedure. The following tables list the various selective prompting characters and the resulting action.

You Enter	Value Displayed	Protected	Value Passed to CPP if Nothing Specified	Marked with > Symbol
??KEYWORD()	Default	No	Default	No
??KEYWORD(VALUE)	Value	No	Value	Yes
?*KEYWORD()	Default	Yes	Default	No
?*KEYWORD(VALUE)	Value	Yes	Value	Yes
?<KEYWORD()	Default	No	Default	No
?<KEYWORD(VALUE)	Value	No	Default	No
?/KEYWORD()	Default	Yes	Default	No
?/KEYWORD(VALUE)	Value	Yes	Default	No
?-KEYWORD()	None	N/A	Default	N/A
?-KEYWORD(VALUE)	None	N/A	Value	N/A
?&KEYWORD()	Default	No	Default	No
?&KEYWORD(VALUE)	Value	No	Default	No
?%KEYWORD()	Default	Yes	Default	No
?%KEYWORD(VALUE)	Value	Yes	Default	No

You Enter	Display Value When F5 Pressed or Blanked Out	Description
??KEYWORD()	Default	Normal keyword prompt with command default.
??KEYWORD(VALUE)	Value	Normal keyword prompt with program specified default.
?*KEYWORD()	Default	Show protected prompt (as information) where command default is the only value used.
?*KEYWORD(VALUE)	Value	Show protected prompt (as information) where program specified value is the only value used. For example, when a value should be shown as information but not changed.
?<KEYWORD()	Default	Normal keyword prompt with command default.
?<KEYWORD(VALUE)	Value	Normal keyword prompt with program specified default.
?/KEYWORD()	Default	Reserved for IBM use.
?/KEYWORD(VALUE)	Value	Reserved for IBM use.
?&KEYWORD()	Default	Normal keyword prompt with command default.
?&KEYWORD(VALUE)	Value	Normal keyword prompt with program specified default.
?%KEYWORD()	Default	Show protected prompt (as information) where command default is the only value used.
?%KEYWORD(VALUE)	Value	Show protected prompt (as information) where program specified value is the only value used. For example, when a value should be shown as information but not changed.

Selective prompting can be used with the QCMDEXC or QCMDCHK program. The format of the call is:

```
CALL PGM(QCMDEXC or QCMDCHK) PARM(command command-length)
```

Following is a brief description of the selective prompting characters:

Selective Prompting Character	Description
??	The parameter is displayed and input-capable.
?*	The parameter is displayed but is not input-capable. Any user-specified value is passed to the command processing program.
?<	The parameter is displayed and is input-capable, but the command default is sent to the CPP unless the value displayed on the parameter is changed.

Selective Prompting Character	Description
?/	Reserved for IBM use.
?–	The parameter is not displayed. The specified value (or default) is passed to the CPP. Not allowed in prompt override programs.
?&	The parameter is not displayed until F9=All parameters is pressed. Once displayed, it is input-capable. The command default is sent to the CPP unless the value displayed on the parameter is changed.
?%	The parameter is not displayed until F9=All parameters is pressed. Once displayed, it is not input-capable. The command default is sent to the CPP.

For a further discussion of QCMDEXC or QCMDCHK refer to “Using the QCMDEXC Program” on page 163 and “Using the QCMDCHK Program” on page 166..

QCMDEXC with Prompting in CL Procedures and Programs

The QCMDEXC program may be used to call the prompter. This use of QCMDEXC with prompting in CL procedures and programs allows you to alter all values on the command except the command name itself. This is more flexible than direct use of the prompter, where you can only enter values not specified in the source (see previous section). If the prompter is called directly with a command such as:

```
?OVRDBF FILE(FILEX)
```

you can specify a value for any parameter except FILE. However, if the command is called during processing of a program using the QCMDEXC program, such as:

```
CALL QCMDEXC PARM('?OVRDBF FILE(FILEX)' 19)
```

you can specify a value for any parameter, including FILE. In this example, FILEX is the default.

Prompting with modifiable specified values may also be accomplished using selective prompting as described earlier in this chapter. However, each keyword must be explicitly selected. The prompter is called directly with a command such as:

```
OVRDBF ??FILE(FILEX) ??TOFILE(*N) ??MBR(*N)
```

Using the Programmer Menu

The programmer menu can be called directly by calling the QPGMMENU program, or by using the Start Programmer Menu (STRPGMMNU) command. You can use the command to specify in advance the defaults that you use with the programmer menu. In addition, the STRPGMMNU command also supports other options that can be used to tailor the use of the programmer menu.

For a description of the STRPGMMNU command and its parameters, see the *CL* section of the **Programming** category in the iSeries Information Center.

Uses of the Start Programmer Menu (STRPGMMNU) Command

The Start Programmer Menu command can be used for the following:

- Performing the same function as a call to QPGMMENU
- Filling in the standard input fields

  Four of the command parameters allow you to fill in the standard input fields at the bottom of the menu. These parameters are the following:

  – Source file
  – Source library
  – Object library
  – Job description

The command may be used with one or more of the parameters that control the initial values of the menu. You could design this as part of an initial program for sign-on or for situations in which a user calls a specific user-written function. The following example shows such a program, with a separate function for each application area requiring different initial values.

```
      PGM
      CHGLIBL   LIBL(PGMR1 QGPL QTEMP)
LOOP:
      STRPGMMNU SRCLIB(PGMR1) OBJLIB(PGMR1) JOBD(PGMR1)
      MONMSG    MSGID(CPF2320) EXEC(GOTO END) /* F3 or F12 to leave menu */
      GOTO LOOP
END:  ENDPGM
```

- Controlling programmer menu options

  The other parameters assist you in controlling the menu and its functions. For example, you can specify ALWUSRCHG(*NO) to prevent a user from changing the values that appear on the menu. This parameter should not be considered to be a security feature because a user who is using the menu can call the STRPGMMNU command and change the values in a separate call. (The user can also start functions by using F10 to call the command entry display.) If the menu is displayed by the STRPGMMNU command, you can prevent the user (by authorization) from calling the QPGMMENU program directly, but you cannot prevent the user from requesting another call of the STRPGMMNU command.
- Adapting the menu create option

  The EXITPGM and DLTOPT parameters allow you to provide your own support for the menu create option (option 3). The system may call a user program when you request option 3. IBM provides online information that discusses the parameters and the parameter list that are passed to the user program. Refer to the *CL* section of the **Programming** category in the iSeries Information Center. The following describes some typical uses of the EXITPGM parameter.

The EXITPGM Parameter

The EXITPGM parameter can be used for the following purposes:

- To change the defaults used on the create commands submitted by option 3.

  For example, if F4 (Prompt) is not used, the EXITPGM parameter could change one or more of the create commands to specify your own default requirements. If F4 is used, the EXITPGM parameter could submit the command as entered by the programmer (with no parameters changed).
- To change parameters regardless of the programmer's use of F4.

  This requires scanning the value of the &RQSDTA512 parameter (which is passed to the exit program) to see if it had already been used and substituting the required value.
- To change other parameters on the SBMJOB command.

For example, the user parameter of the SBMJOB command could be changed to specify the value of the job description instead of the value of *CURRENT. It is also possible to retrieve the values of one or more job attributes by using the RTVJOBA command, entering the attributes as specific values.

- To enforce local programming conventions.

  For example, if you have a naming standard that requires all physical files to be named with 7 characters and end with a P, the exit program could reject any attempt to use the CRTPF command with a name that did not follow this standard.

Command Analyzer Exit Points

The exit program registration facility provides two exit points for the system.

- The QIBM_QCA_CHG_COMMAND exit point can register one and only one exit point for a specific command. The program specified for this exit point is called by the command analyzer before it passes control to the prompter.
- You can register up to ten exit programs for each command for the QIBM_QCA_RTV_COMMAND exit point. The command analyzer calls these exit programs after running the validity checking program (VCP) and before running the command processing program (CPP) for the command.

See the *APIs* section of the **Programming** category in the iSeries Information Center for a complete description of these exit points.

Application Programming for DBCS Data

Special considerations must be made when designing application programs to process double-byte data or converting alphanumeric application programs to double-byte programs.

Designing DBCS Application Programs

Design your application programs for processing double-byte data in the same way you design application programs for processing alphanumeric data, with the following additional considerations:

- Identify double-byte data used in the database files, if any.
- Design display and printer formats that can be used with double-byte data.
- If needed, provide double-byte conversion as a means of entering data for interactive applications. Use the DDS keyword for double-byte conversion (IGCCNV) to specify DBCS conversion in display files.
- Write double-byte error messages to be displayed by the program.
- Specify extension character processing so that the system prints and displays all double-byte data.
- Determine which double-byte characters, if any, must be defined. ADTS/400:

  Character Generator Utility describes how to define double-byte characters for DBCS-supported countries.

Converting Alphanumeric Programs to Process DBCS Data

If an alphanumeric application program uses externally-described display files, you can change that application program to a double-byte application program by changing only the files. To convert an application program, do the following:

1. Create a duplicate copy of the source statements for the alphanumeric file you want to change.

2. Change alphanumeric constants and literals to double-byte constants and literals.
3. Change fields in the file to one of the following data types to enter DBCS data:
   - DBCS-open (O) data type
   - DBCS-only (J) data type
   - DBCS-either (E) data

   You do not have to change the length of the fields.
4. Store the converted display file in a separate library. Give the file the same name as its alphanumeric version.
5. To use the converted file in a job, change the library list, using the Change Library List (CHGLIBL) command, for the job in which the file is used. The library in which the double-byte display file is stored is then checked before the library in which the alphanumeric version of the file is stored.

Using DBCS Data in a CL Program

The following program shows the use of different keyboard shifts within a CL program. Note how the double-byte data is used only as text values in this program; the commands themselves are in alphanumeric characters.

When run, this program shows you how the different keyboard shifts for DDS display files are used.

```
PGM

      DCLF    IGCTEST

START: CHGVAR &OUTPUTA  'ABCDEFGHIJ'

       CHGVAR &OUTPUTJ  'ABCD'
       CHGVAR &BOTHJ    'ABCD'
       CHGVAR &OUTPUTE  'EFGH'
       CHGVAR &OUTPUTO  'A B C D F'


LOOP:  SNDRCVF

      IF &IN01 RETURN
       CHGVAR &OUTPUTA  &INPUTA
       CHGVAR &OUTPUTJ  &INPUTJ
       CHGVAR &OUTPUTE  &INPUTE
       CHGVAR &BOTHE    &INPUTE
       CHGVAR &OUTPUTO  INPUTO


    GOTO LOOP

ENDPGM
```

RV3W197-0

Sample CL Programs

The following sample programs demonstrate the flexibility, simplicity, and versatility of CL programs. The following programs are described by their function and probable user.

Note: Code generated by the ILE CL compiler in V4R3 and later releases is threadsafe. However, many commands are not threadsafe. Therefore, do not consider a CL procedure as threadsafe unless all the commands the CL procedure uses are threadsafe. You can use the Display Command (DSPCMD) command to determine if a command is threadsafe. For additional information on threads, access the iSeries Information Center and open the topics under the **Programming** category of information.

Initial Program for Setup (Programmer)

```
PGM
CHGLIBL LIBL(TESTLIB QGPL QTEMP)
CHGJOB OUTQ(WSPRTR)
TFRCTL QPGMMENU
ENDPGM
```

The test library is placed first on the library list, an output queue is selected for a convenient printer, and the programmer menu is displayed.

Moving an Object from a Test Library to a Production Library (Programmer)

```
PGM PARM(&OBJ &OBJTYPE &OPER)
DCL &OBJ *CHAR LEN(10)
DCL &OBJTYPE *CHAR LEN(7)
DCL &OPER *CHAR LEN(1)  /* R=Replace M=Move */
IF ((&OPER *NE 'M') *AND (&OPER *NE 'R')) THEN(DO)
        SNDPGMMSG MSG('Operation code must be "R" or "M" ')
        RETURN
        ENDDO
IF ((&OBJTYPE *NE *PGM) *AND (&OBJTYPE *NE *FILE) *AND (&OBJTYPE +
    *NE *DTAARA)) THEN(DO)
        SNDPGMMSG MSG('Object' *BCAT &OBJ *BCAT ' must be *PGM, +
                  *FILE, or *DTAARA')
        RETURN
        ENDDO
CHKOBJ BLDLIB/&OBJ OBJTYPE(&OBJTYPE)
MONMSG MSGID(CPF9801) EXEC(DO)
        SNDPGMMSG MSG('Object or object type does not exist +
                  in BLDLIB')
        RETURN
        ENDDO
IF (&OPER *EQ 'M') THEN(DO)
        MOVOBJ BLDLIB/&OBJ OBJTYPE(&OBJTYPE) TOLIB(PRODLIB)
        MONMSG MSGID(CPF3208) EXEC(DO)
                        SNDPGMMSG MSG('Object' *BCAT &OBJ *BCAT ' +
                                  already exists in PRODLIB')
                        RETURN
                        ENDDO
        CHKOBJ PRODLIB/&OBJ OBJTYPE(&OBJTYPE)
        MONMSG MSGID(CPF9801) EXEC(DO)
        SNDPGMMSG MSG('Object or object type does not +
                  exist in PRODLIB')
                  RETURN
```

```
                ENDDO
        ENDDO
RETURN
ENDPGM
```

The object name, object type, and operation code are passed from another program or procedure. Checks are performed to see that the operation code and object type are correct, and that the object exists in the test library. The object is moved unless it already exists in the production library. The move is then confirmed. More commands can be added to grant additional authority to the object or to handle additional exceptions and additional object types.

Saving Specific Objects in an Application (System Operator)

Example

```
PGM
SAVOBJ OBJ(FILE1 FILE2) LIB(LIBA) OBJTYPE(*FILE) DEV(TAP01) +
  CLEAR(*YES)
SAVOBJ OBJ(DTAARA1) LIB(LIBA) OBJTYPE(*DTAARA) DEV(TAP01)
SNDPGMMSG MSG('Save of daily backup of LIBA completed') +
  MSGTYPE(*COMP)
ENDPGM
```

This program ensures consistent command entry for regularly repeated procedures.

Additional Save Object (SAVOBJ) commands can, of course, be added. However, this program relies on the operator selecting the correct diskette or tape for each periodic backup of each application. This can be controlled by assigning unique names to each diskette or tape set for each save operation. If you want to save your payroll files separately each week for four weeks, for instance, you might name each diskette or tape differently and write the program to compare the name of the diskette or tape against the correct name for that week.

Recovery from Abnormal End (System Operator)

```
PGM
DCL &SWITCH *CHAR LEN(1)
RTVSYSVAL SYSVAL(QABNORMSW) RTNVAR(&SWITCH)
IF (&SWITCH *EQ '1') THEN(DO)  /*CALL RECOVERY PROGRAMS*/
        SNDPGMMSG MSG('Recovery programs in process. +
          Do not start subsystems until notified') +
          MSGTYPE(*INFO) TOMSGQ(QSYSOPR)
        CALL PGMA
        CALL PGMB
        SNDPGMMSG MSG('Recovery programs complete. +
          Startup subsystems') +
          MSGTYPE(*INFO) TOMSGQ(QSYSOPR)
        RETURN
        ENDDO
ENDPGM
```

Submitting a Job (System Operator)

```
PGM  /*DAILYAC*/
SBMJOB JOB(DAILYACCRC) JOBD(ACCRC2) +
             CMD(CALL ACCRC305 PARM(DAILY))
SNDPGMMSG    MSG('Daily Accounts Receivable job DAILYACCRC +
             submitted to batch') MSGTYPE(*COMP)
ENDPGM
```

Instead of typing in all the parameters for submitting a job, the system operator calls DAILYAC.

Timing Out While Waiting for Input from a Device Display

```
DCLF FILE(QGPL/MENU)
DOWHILE '1' /* DO FOREVER */
  SNDRCVF DEV(*FILE) RCDFMT(MENUFMT) WAIT(*NO)
  WAIT  MONMSG MSGID(CPF0889) EXEC(SIGNOFF)
  CHGVAR VAR(&IN99) VALUE('0')
  IF COND(&IN01) THEN(ITERATE)
  SELECT
    WHEN (&OPTION *EQ '1') (CALL ORDENT) /* OPTION 1-ORDER ENTRY      */
    WHEN (&OPTION *EQ '2') (CALL ORDDSP) /* OPTION 2-ORDER DISPLAY    */
    WHEN (&OPTION *EQ '3') (CALL ORDCHG) /* OPTION 3-ORDER CHANGE     */
    WHEN (&OPTION *EQ '4') (CALL ORDPRT) /* OPTION 4-ORDER PRINT      */
    WHEN (&OPTION *EQ '9') (SIGNOFF)     /* OPTION 9-SIGNOFF          */
    OTHERWISE DO                         /* OPTION SELECTED NOT VALID */
      CHGVAR VAR(&IN99) VALUE('1')
      ENDDO
  ENDSELECT
ENDDO
ENDPGM
```

This program illustrates how to write a CL program using a display file that will wait for a specified amount of time for the user to enter an option. If he does not, the user is signed off.

The display file was created with the following command:

```
CRTDSPF FILE(MENU) SRCFILE(QGPL/QDDSSRC) SRCMBR(MENU)  +
    DEV(*REQUESTER) WAITRCD(60)
```

The display file will use the *REQUESTER device. When a WAIT command is issued, it waits for the number of seconds (60) specified on the WAITRCD keyword. The following is the DDS for the display file:

```
SEQNBR *... ... 1 ... ... 2 ... ... 3 ... ... 4 ... ... 5 ... ... 6 ... ... 7 ... ... 8

 0100       A                                      PRINT CA01(01)
 0200       A          R MENUFMT                   BLINK
 0300       A                                      TEXT('Order Entry Menu')
 0400       A                                    1 31'Order Entry Menu'
 0500       A                                    2  2'Select one of the following:    '
 0600       A                                    3  4'1.  Enter Order'
 0700       A                                    4  4'2.  Display Order'
 0800       A                                    5  4'3.  Change Order'
 0900       A                                    6  4'4.  Print Order'
 1000       A                                    7  4'9.  Sign Off'
 1100       A                                   23  2'Option:'
 1200       A            OPTION        1   I 23 10
 1300       A  99                                  ERRMSG('Invalid option selected.')

                          * * * * * *  E N D  O F  S O U R C E  * * * * *
```

The program performs a SNDRCVF WAIT(*NO) to display the menu and request an option from the user. Then it issues a WAIT command to accept an option from the user. If the user enters a 1 through 4, the appropriate program is called. If the user enters a 9, the SIGNOFF command is issued. If the user enters an option that is not valid, the menu is displayed with an 'OPTION SELECTED NOT VALID' message. The user can then enter another valid option. If the user does not respond within 60 seconds, the CPF0889 message is issued to the program and the MONMSG command issues the SIGNOFF command.

A SNDF command using a record format containing the INVITE DDS keyword could be used instead of the SNDRCVF WAIT(*NO). The function would be the same.

Performing Date Arithmetic

```
/* Calculate new date from current system date.  Pass negative     */
/* number to subtract, positive number to add                      */
/*                                                                 */
/* The first parameter is a character 8 date in YYYYMMDD format    */
/* or the special value *CURRENT                                   */
/*                                                                 */
/* The second parameter is a decimal value for the number of days  */
/* to adjust the first parameter by                                */
/*                                                                 */
/* Test cases:  CALL CALCDATE (*CURRENT -5)                        */
/*              CALL CALCDATE (*CURRENT  5)                        */
/*              CALL CALCDATE ('20030225' -90)                     */
/*              CALL CALCDATE ('30020228' 90)                      */
/*                                                                 */
/* There is no error handling in this sample, so make sure the     */
/* input dates are valid (that is, no 20031325).  The valid date   */
/* date range is Oct 14 1582 to Dec 31 9999                        */
/*                                                                 */
            PGM        PARM(&curdate &DAYSTOCHG)
            DCL        VAR(&CURDATE) TYPE(*CHAR) LEN(8)
            DCL        VAR(&DAYSTOCHG) TYPE(*DEC) LEN(15 5)
            DCL        VAR(&DATETIME) TYPE(*CHAR) LEN(17)
            DCL        VAR(&DATE) TYPE(*CHAR) LEN(8)
            DCL        VAR(&LILDATEINT) TYPE(*CHAR) LEN(4)
            DCL        VAR(&LILDATEDEC) TYPE(*DEC) LEN(10 0)
            DCL        VAR(&ERRCOD) TYPE(*CHAR) LEN(4) +
                         VALUE(X'00000000')
            DCL        VAR(&MSG) TYPE(*CHAR) LEN(50)
            IF         COND(&CURDATE = '*CURRENT') THEN(DO)
            CALL       PGM(QWCCVTDT) PARM('*CURRENT' ' ' '*YYMD' +
                         &DATETIME &ERRCOD) /* Get current system +
                         date and time in YYYYMMDD */
            CHGVAR     VAR(&DATE) VALUE(%SST(&DATETIME 1 8)) /* Get +
                          just the date portion */
            ENDDO
            ELSE       CMD(CHGVAR VAR(&DATE) VALUE(&CURDATE)) /* +
                          Use the date provided */
            CALLPRC    PRC(CEEDAYS) PARM(&DATE 'YYYYMMDD' +
                          &LILDATEINT *OMIT) /* Get Lilian date for +
                          current date */
            CHGVAR     VAR(&LILDATEDEC) VALUE(%BIN(&LILDATEINT)) /* +
                          Get Lilian date in decimal format */
            CHGVAR     VAR(&LILDATEDEC) VALUE(&LILDATEDEC + +
                          &DAYSTOCHG) /* Adjust specified number +
                          of days */
            CHGVAR     VAR(%BIN(&LILDATEINT)) VALUE(&LILDATEDEC) /* +
                          Get Lilian date in integer format */
            CALLPRC    PRC(CEEDATE) PARM(&LILDATEINT 'YYYYMMDD' +
                         &DATE *OMIT) /* Return calculated date in +
                         YYYYMMDD format */
            CHGVAR     VAR(&MSG) VALUE('The new date is ' *CAT &DATE)
            SNDPGMMSG  MSG(&MSG) TOPGMQ(*EXT)
            ENDPGM
```

This program illustrates how to write a CL program that adds or subtracts a given number of days for the current system date.

Retrieving Program Attributes

You can use the Display Program (DSPPGM) command to display the attributes of a program. To retrieve some of the attributes (such as program type, source member, text, creation date) into CL variables, you can use the Display Object Description (DSPOBJD) command to build an output file. The system can then read a CL procedure or program that uses the Declare File (DCLF) and Receive File (RCVF) commands. To access other attributes of the DSPPGM command (such as USRPRF), you can use the Retrieve program information API (QCLRPGMI).

Loading and Running an Application from Tape or Optical Media

The Load and Run Media Program (LODRUN) command allows the user to load and run an application written by another user or a software vendor from tapes or optical media supplied by the other user.

When the LODRUN command is run:

- The media is searched for the user-written program, which must be named QINSTAPP. If tape is used, the tape is rewound first.
- If a QINSTAPP program already exists in the QTEMP library on the user's system, it is deleted.
- The QINSTAPP program is restored to the QTEMP library using the RSTOBJ command.
- Control of the system is passed to the QINSTAPP program. The QINSTAPP program may be used, for example, to restore other applications to the user's system and run those applications.

Responsibilities of the Application Writer

The user supplying the QINSTAPP program is responsible for writing and supporting it. The QINSTAPP program is not supplied by IBM*. The program can be designed to accomplish many different tasks. For example, the program could:

- Restore and run other programs or applications
- Restore a library
- Delete another program or application
- Create specific environments
- Correct problems in existing applications

Figure 3 on page 182 shows an example of a QINSTAPP program. The program is saved to a tape or optical media by the program writer and loaded on the system using the LODRUN command. The LODRUN command passes control of the system to the program, which then performs the tasks written into the program.

```
PGM         PARM(&DEV)  /* "Device" is only Parm allowed          */
DCL         VAR(&DEV)   TYPE(*CHAR) LEN(10)
DCL         VAR(&MODEL) TYPE(*CHAR) LEN(4)

/* Can check for appropriate model number, release level, and so on */
RTVSYSVAL   SYSVAL(QMODEL) RTNVAR(&MODEL)
IF          (&MODEL *EQ 'xxxxx') THEN...

/* Install a library for new application (programs, data):        */
RSTLIB      SAVLIB(NEWAPP) DEV(&DEV) ENDOPT(*LEAVE) +
              MBROPT(*ALL)
/* Install a command to start new application:                    */
            RSTOBJ OBJ(NEWAPP) SAVLIB(QGPL) DEV(&DEV) +
              MBROPT(*ALL)

END:        ENDPGM
```

Figure 3. Example of an Application Using the LODRUN Command

Chapter 7. Defining Messages

On the iSeries server, communication between procedures or programs, between jobs, between users, and between users and procedures or programs occurs through messages. A message can be predefined or immediate:

- A predefined message is created and exists outside the program that uses it. Predefined messages are stored in message files and have a message number. An example of a system predefined message is:

  ```
  CPF0006  Errors occurred in command.
  ```

- An immediate message is created by the sender at the time it is sent. An immediate message is not stored in a message file. An example of an immediate message received at a display station is:

  ```
  From  . . . :   QSYSOPR      06/12/94   10:50:54
  System going down at 11:00; please sign off
  ```

Your system comes with an extensive set of predefined messages that allow communication between programs within the system and between the system and its users. Each licensed program you order has a message file that is stored in the same library as the licensed program it applies to. For example, system messages are stored in the file QCPFMSG in the library QSYS.

The system uniquely identifies each predefined message in a message file by a 7-character code and defines it by a message description. The message description contains information, such as message text and message help text, severity level, valid and default reply values, and various other attributes. See the Add Message Description (ADDMSGD) command description in online help, or in the *CL* section of the **Programming** category in the iSeries Information Center.

All messages that are sent or received in the system are transmitted through a message queue. Messages that are issued in response to a direct request, such as a command, are automatically displayed on the display from which the request was made. For all other messages, the user, program or procedure must receive the message from the queue or display it. There are several IBM-supplied message queues in the system; these message queues are described later in this chapter (see "Types of Message Queues" on page 201).

The system also writes some of the messages that are issued to logs. A job log contains information related to requests entered for a job, the history log contains job, subsystem, and device status information. See "Message Logging" on page 272 for more information on logging.

You can create your own message files and message descriptions. By creating predefined messages, you can use the same message in several procedures or programs but define it only once. You can also change and translate predefined messages into languages other than English (based on the user viewing the messages) without affecting the procedures and programs that use them. If the messages were defined in the procedure or program, the module or program would have to be recompiled when you change the messages.

In addition to creating your own messages and message files, the system message handling function allows you to:

© Copyright IBM Corp. 1997, 2004

- Create and change message queues (Create Message Queue [CRTMSGQ], Change Message Queue [CHGMSGQ], and Work with Message Queues [WRKMSGQ] commands)
- Create and change message files (Create Message File [CRTMSGF], Change Message File [CHGMSGF] commands)
- Add message descriptions (Add Message Description [ADDMSGD] command)
- Change message descriptions (Change Message Description [CHGMSGD] command)
- Remove message descriptions (Remove Message Description [RMVMSGD] command)
- Send immediate messages (Send Message [SNDMSG], Send Break Message [SNDBRKMSG], Send Program Message [SNDPGMMSG], and Send User Message [SNDUSRMSG] commands)
- Display messages and message descriptions (Display Messages [DSPMSG], Display Message Description [DSPMSGD], Work with Messages [WRKMSG], and Work with Message Descriptions [WRKMSGD] commands)
- Use a CL procedure or program to:
  - Send a message to a workstation user or the system operator (Send User Message [SNDUSRMSG] command)
  - Send a message to a message queue (Send Program Message [SNDPGMMSG] command)
  - Receive a message from a message queue (Receive Message [RCVMSG] command)
  - Send a reply for a message to a message queue (Send Reply [SNDRPY] command)
  - Retrieve a message from a message file (Retrieve Message [RTVMSG] command)
  - Remove a message from a message queue (Remove Message [RMVMSG] command)
  - Monitor for escape, notify, and status messages that are sent to a call message queue (Monitor Message [MONMSG] command)
- Use the system reply list to specify the replies for predefined inquiry messages sent by a job (Add Reply List Entry [ADDRPYLE], Change Reply List Entry [CHGRPYLE], Remove Reply List Entry [RMVRPYLE], and Work with Reply List Entry [WRKRPYLE] commands)

When a message is sent, it is defined as one of the following types:

- Informational (*INFO). A message that conveys information about the condition of a function.
- Inquiry (*INQ). A message that conveys information but also asks for a reply.
- Notify (*NOTIFY). A message that describes a condition for which a procedure or program requires corrective action or a reply from its calling procedure or program. A procedure or program can monitor for the arrival of notify messages from the programs or procedures it calls.
- Reply (*RPY). A message that is a response to a received inquiry or notify message.
- Sender's copy (*COPY). A copy of an inquiry or notify message that is kept by the sender.
- Request (*RQS). A message that requests a function from the receiving procedure or program. (For example, a CL command is a request message.)

- Completion (*COMP). A message that conveys completion status of work.
- Diagnostic (*DIAG). A message about errors in the processing of a system function, in an application program, or in input data.
- Status (*STATUS). A message that describes the status of the work done by a procedure or program. A procedure or program can monitor for the arrival of status messages from the program or procedure it calls. Status messages sent to the external message queue (*EXT) are shown at the display station and can be used to inform the display station user of an operation in progress.
- Escape (*ESCAPE). A message that describes a condition for which a procedure or program must end abnormally. A procedure or program can monitor for the arrival of escape messages from the program or procedure it calls or from the machine. Control does not return to the sending program after an escape message is sent.

This chapter describes:

- How to create your own message files
- How to add message descriptions to a message file
- Types of message queues
- How to create message queues

Creating a Message File

To create your own predefined messages, you must first create the message file into which the messages are to be placed. Use the Create Message File (CRTMSGF) command to create the message file. You then use the Add Message Description (ADDMSGD) command to describe your messages and place them in the message file.

On the CRTMSGF command, you can specify the maximum size in K bytes on the SIZE parameter. The following formula can be used to determine the maximum:

```
S + (I x N)
```

where:

S Is the initial amount of storage

I Is the amount of storage to add each time

N Is the number of times to add storage

The defaults for S, I, and N are 10, 2, and *NOMAX, respectively.

For example, you specify S as 5, I as 1, and N as 2. When the file reaches the initial storage amount of 5K, the system automatically adds another 1K to the initial storage. The amount added (1K) can be added to the storage two times to make the total maximum of 7K. If you specify *NOMAX as N, the maximum size of the message file is 16M.

When you specify a maximum size for a message file and the message file becomes full, you cannot change the size of the message file. You then need to create another message file and re-create the messages in the new file. The Merge Message File (MRGMSGF) command can be used to copy message descriptions from one message file to another. Since you will want to avoid this step, it is important to calculate the size needed for your message file when you create it, or specify *NOMAX.

Message Files in Independent ASPs

Message files can be created in an independent auxiliary storage pool (ASP), but it is not recommended because the independent ASP can be taken offline. This would prevent messages in job logs and message queues from being displayed correctly if the independent ASP is offline.

Determining the Size of a Message File

You can determine the size of a message by using the following formula. (The ADDMSGD command parameters are given in parentheses.)

- Message index equals 42 bytes base plus the length of the message.
- Message text (MSG) equals 16 bytes base plus the length of the message.
- Message online help information (SECLVL), if any, equals 16 bytes base plus the length of the message help.
- Formats (FMT), if any, equal 14 bytes plus (3 x number of FMTS).
- Type and length (TYPE and LEN) equal 48 bytes.
- Special value (SPCVAL) equals 2 plus (64 x number of SPCVALs).
- Values (VALUES) equal 32 x (number of VALUES).
- Range (RANGE) equals 64 bytes.
- Relation (REL) equals the length of the relation.
- Default (DFT) equals the length of the default reply.
- Default program, log problem, and dump list (DFTPBM, LOGPRB, DMPLST) equal 35 plus (2 x number in DMPLST).
- ALROPT equals 12 bytes.

The smallest possible entry in a message file is 59 bytes and the largest possible entry is 5764 bytes. The following table describes the largest possible entry:

Message index	42 bytes
Message text	148 bytes
Message help text	3016 bytes
99 formats	311 bytes
Type and length	48 bytes
20 special values	1282 bytes
20 values	640 bytes
Default reply value	32 bytes
Default program and dump list	233 bytes
Alert option	12 bytes

In the following example, the CRTMSGF command creates the message file USRMSG:

```
CRTMSGF   MSGF(QGPL/USRMSG) +
          TEXT('Message file for user-created messages')
```

If you are creating a message file to be used with the DSPLY operation code in RPG for OS/400, the message file must be named QUSERMSG.

Adding Messages to a File

You use the Add Message Description (ADDMSGD) command to describe your predefined messages and to add them to the message file you created. On the ADDMSGD command, you specify the message identifier, the name of the message file into which the message is to be placed, and the message description. In the message description, you can specify:

- Message text (required) with optional substitution variables
- Message help text with optional substitution variables
- Severity code
- Description of the format of the message data to be used for the substitution variables
- Validity checking values for a reply
- Default value for a reply
- Default message handling action for escape messages
- Creation level
- Alert options
- Entry in the error log
- Coded Character Set ID (CCSID)

Each of the items that can be contained in the message description is described in more detail on the following pages.

The following commands are also available for use with message descriptions:

Change Message Description (CHGMSGD)
Changes a message description.

Display Message Description (DSPMSGD)
Displays a message description. (A range of message identifiers can be specified in this command.)

Remove Message Description (RMVMSGD)
Removes a message description from a message file.

Retrieve Message (RTVMSG)
Retrieves a message from a message file.

Merge Message File (MRGMSGF)
Merges messages from one message file into another message file.

Work with Message Descriptions (WRKMSGD)
Displays a list of messages in a file and allows you to add, change, or delete message descriptions.

Assigning a Message Identifier

The message identifier you specify on the ADDMSGD command is used to refer to the message and is the name of the message description. The message identifier must be 7 characters:

```
pppmmnn
```

where `ppp` is the product or application code, `mm` is the numeric group code, and `nn` is the numeric subtype code. The number specified as `mmnn` can be used to further divide a set of product or application messages. Numeric group and subtype codes consist of decimal numbers 0 through 9 and the characters A through F.

For example:

```
CPF1234
```

is message 1234 of CPF.

When you create your own messages, using the letter U as the first character in the product code is a good way to distinguish your messages from system messages. For example:

```
USR3567
```

The first character of the code must be alphabetic, the second and third characters can be alphanumeric; the group code and the subtype code must consist of decimal numbers 0 through 9 and the characters A through F. Note that although this range can be called a set of hexadecimal numbers, any sorting of the message numerics treats A through F as characters.

For example, when displaying a range of message descriptions, CPFA000 precedes CPF1000.

You should use care in using a numeric subtype code of 00 in the message identifier. If you use a numeric subtype code of 00 for a message that can be sent as an escape, notify, or status message and that can, therefore, be monitored, a subtype code of 00 in the Monitor Message (MONMSG) command causes all messages in the numeric group to be monitored. See "Monitoring for Messages in a CL Program or Procedure" on page 239 for more information.

Defining Messages and Message Help

You can define two levels of messages on the ADDMSGD command. The text of the message is required and should identify the condition that caused the message to be issued. Message help is optional and should explain the condition further or explain the corrective action to be taken. To get message help, the display station user must move the cursor to the message line and press the Help key when the message is displayed. Message help can be formatted for the display station using three format control characters. These characters may be used to make the message help (usually online help information) more readable for the user.

Each of the three format control characters must be followed by a blank to separate them from the message text.

&Nb (where b is a blank)
Forces the text to a new line (column 2). If the text is longer than one line, the next lines are indented to column 4 until the end of the text or until another format control character is found.

&Pb (where b is a blank)
Forces the text to a new line, indented to column 6. If the text is longer than one line, the next lines start in column 4 until the end of the text or until another format control character is found.

&Bb (where b is a blank)
Forces the text to a new line, starting in column 4. If the text is longer than one line, the next lines are indented to column 6 until the end of the text or until another format control character is found.

Assigning a Severity Code

The severity code you assign to a message on the ADDMSGD command indicates how important the message is. The higher the severity code the more serious the condition is. The following lists the severity codes you can use and their meanings. (These severity codes and their meanings are consistent with the severity codes assigned to IBM-predefined messages.)

00: Information. For information purposes only; no error was detected and no reply is needed. The message could indicate that a function is in progress or that a function has completed successfully.

10: Warning. A potential error condition exists. The procedure or program may have taken a default, such as supplying missing input. The results of the operation are assumed to be successful.

20: Error. An error has been detected, but it is one for which automatic recovery procedures probably were applied; processing has continued. A default may have been taken to replace erroneous input. The results of the operation may not be valid. The function may have been only partially completed; for example, some items in a list processed correctly while others failed.

30: Severe error. The error detected is too severe for automatic recovery, and no defaults are possible. If the error was in source data, the entire input record was skipped. If the error occurred during procedure or program processing, it leads to an abnormal end of the procedure or program (severity 40). The results of the operation are not valid.

40: Abnormal end of procedure or function. The operation has ended, possibly because the procedure or program was unable to handle data that was not valid, or possibly because the user has canceled it.

50: Abnormal end of job. The job was ended or was not started. A routing step may have ended abnormally or failed to start, a job-level function may not have been performed as required, or the job may have been canceled.

60: System status. Issued only to the system operator. It gives either the status of or a warning about a device, a subsystem, or the system.

70: Device integrity. Issued only to the system operator. It indicates that a device is malfunctioning or in some way is no longer operational. The user may be able to recover from the failure, or the assistance of a service representative may be required.

80: System alert. A message with a severity code of 80 is issued for immediate messages. It also warns of a condition that, although not severe enough to stop the system now, could become more severe unless preventive measures are taken.

90: System integrity. Issued only to the system operator. It describes a condition that renders either a subsystem or the system inoperative.

99: Action. Some manual action is required, such as entering a reply, changing printer forms, or replacing diskettes.

For a detailed discussion of the SEV parameter, see the *CL* section of the **Programming** category in the iSeries Information Center.

Defining Substitution Variables

On the FMT parameter on the ADDMSGD command, you can specify substitution variables for either first- or second-level messages. For example:

```
File &1 not found
```

contains the substitution variable &1. When the message is displayed or retrieved, the variable &1 is replaced with the name of the file that could not be found. This name is supplied by the sender of the message. For example:

```
File ORDHDRP not found
```

Compare this to:

```
File not found
```

Substitution variables can make your message more specific and more meaningful.

The substitution variable must begin with & (ampersand) and be followed by n, where n is any number from 1 through 99. For example, for the message:

```
File &1 not found
```

the substitution variable is defined as:

```
FMT((*CHAR 10))
```

When you assign numbers to substitution variables, you must begin with the number 1 and use the numbers consecutively. For example, &1, &2, &3, and so on. However, you do not have to use all the substitution variables defined for a message description in the message that is sent.

For example, the message:

```
File &3 not available
```

is valid even though &1 and &2 are not used in the messages. However, to do this, you must define &1, &2, and &3 on the FMT parameter of the ADDMSGD command. For the preceding message, the FMT parameter could be:

```
FMT((*CHAR 10) (*CHAR 2) (*CHAR 10))
```

where the first value describes &1, the second &2, and the third &3. The description for &1 and &2 must be present if &3 is used. In addition, when this message is sent, the MSGDTA parameter on the Send Program Message (SNDPGMMSG) command should include all the data described on the FMT parameter. To send the preceding message, the MSGDTA parameter should be at least 22 characters long.

For the preceding message, you could also specify the FMT parameter as:

```
FMT((*CHAR 0) (*CHAR 0) (*CHAR 10))
```

Because &1 and &2 are not used in the message, they can be described with a length of 0. Then no message data needs to be sent. (The MSGDTA parameter on the SNDPGMMSG command needs to be only 10 characters long in this case.)

An example of using &3 in the message and including &1 and &2 in the FMT parameter is when &1 and &2 are specified on the DMPLST parameter. (The DMPLST parameter specifies that the data is to be dumped when this message is sent as an escape message to a program that is not monitoring for it.)

The substitution variables do not have to be specified in the message in the same order in which they are defined in the FMT parameter. For example, three values can be defined in the FMT parameter as:

```
FMT((*CHAR 10) (*CHAR 10) (*CHAR 7))
```

The substitution variables can be used in the message as follows:

```
Object &1 of type &3 in library &2 is not available
```

If this message is sent in a CL procedure or program, you can concatenate the values used for the message data such as:

```
SNDPGMMSG .....MSGDTA(&OBJ *CAT &LIB *CAT &OBJTYPE)
```

You must specify the format of the message data field for the substitution variable by specifying data type and, optionally, length on the ADDMSGD command. The valid data types for message data fields are:

- Quoted character string (*QTDCHAR). A string of character data to be enclosed in apostrophes. Preceding and trailing blanks are not deleted. If length is not specified in the message description, the sender determines the length of the field.
- Character string (*CHAR). A string of character data not to be enclosed in apostrophes. Trailing blanks are deleted. If length is not specified in the message description, the sender determines the length of the field.
- Convertible character string (*CCHAR). A string of character data not to be enclosed in apostrophes. Trailing blanks are deleted. The length is always determined by the sender. If data of this type is sent to a message queue that has a CCSID tag other then 65535 or 65534, the data is converted from the CCSID of the message data to the CCSID of the message queue. Conversions can also occur on data of this type when the data is obtained from the message queue using a receive or display function. See the *Globalization* topic in the **Programming** category of the iSeries Information Center for more information on the use of message handlers with CCSIDs.
- Hexadecimal (*HEX). A string to be preceded by the character X and enclosed in apostrophes; each byte of the string is to be converted into two hexadecimal characters (0 through 9 and A through F). If length is not specified in the message description, the sender determines the length of the field.
- Binary (*BIN). A binary integer (either 2, 4, or 8 bytes long) formatted as a signed decimal integer. Unless provided a specified length, the system will assume that the binary integer is 2.
- Unsigned binary (*UBIN). An unsigned binary integer (either 2, 4 or 8 bytes long) formatted as an unsigned decimal integer. Unless provided a specified length, the system will assume that the binary integer is 2.
- Decimal (*DEC). A packed decimal number to be formatted as a signed decimal number with a decimal point. Length must be specified; decimal positions default to 0.
- System pointer (*SYP). A 16-byte pointer to a system object. In a message or message help, the 10-character name of the object is formatted the same as the *CHAR type data.
- Space pointer (*SPP). A 16-byte pointer to a program object. In a dump, the data in the object is formatted the same as the *HEX type data. *SPP cannot be used as substitution text in a message; it can only be used as part of the DMPLST parameter on the ADDMSGD command.

The following data types are valid only in IBM-supplied message descriptions and should not be used for other messages:

- Time interval (*ITV). An 8-byte time interval that contains the time to the nearest whole second for various wait time out conditions.
- Date and time stamp (*DTS). An 8-byte system date and time stamp for which the date is to be formatted as specified in the QDATFMT and QDATSEP system values and the time is to be formatted as hh:mm:ss.

Specifying Validity Checking for Replies

On the ADDMSGD command, you can specify the type of reply that is valid for an inquiry or notify message. You can specify (parameters are given in parentheses):

- Type of reply (TYPE)
  - Decimal (*DEC)
  - Character (*CHAR)
  - Alphabetic (*ALPHA)
  - Name (*NAME)
- Maximum length of reply (LEN)
  - For decimal, 15 digits (9 decimal positions)
  - For character and alphabetic, 32 characters
  - For name, 10 characters

  Note: If you do not specify any validity checking (VALUES, RANGE, REL, SPCVAL, DFT), the maximum length of a reply is 132 characters for types *CHAR and *ALPHA.
- Values that can be used for the reply
  - A list of values (VALUES)
  - A list of special values (SPCVAL)
  - A range of values (RANGE)
  - A simple relationship that the reply value must meet (REL)

  Note: The special values are values that can be accepted but that do not satisfy any other validity checking values.

When a display station user enters a reply to a message, the keyboard is in lower shift which causes lowercase characters to be entered. If your program needs the reply to be in uppercase characters, you can do one of the following:

- Use the SNDUSRMSG command which supports a translation table option which defaults to converting lowercase to uppercase.
- Require the display station user to enter uppercase characters by specifying only uppercase characters for the VALUES parameter.
- Specify the VALUES parameter as uppercase and use the SPCVAL parameter to convert the corresponding lowercase characters to uppercase.
- Use TYPE(*NAME) if the characters to be entered are all letters (A-Z). The characters are converted to uppercase before being checked.

Sending an Immediate Message and Handling a Reply

In this example, the procedure does the following:

- Sends an immediate inquiry message to QSYSOPR
- Requests a reply of yes or no (Y or N)

- Ensures that a valid reply has been entered
- Does a time-out if the operator does not reply within 120 seconds

```
           PGM
           DCL        &MSGKEY *CHAR LEN(4)
           DCL        &MSGRPY *CHAR LEN(1)
SNDMSG:    SNDPGMMSG  MSG('.... Reply Y or N') TOMSGQ(QSYSOPR) +
                        MSGTYPE(*INQ) KEYVAR(&MSGKEY)
           RCVMSG     MSGTYPE(*RPY) MSGKEY(&MSGKEY) WAIT(120) +
                        MSG(&MSGRPY)
           IF         ((&MSGRPY *EQ 'Y') *OR (&MSGRPY *EQ 'y')) DO
           .
           .
           GOTO       END
           ENDDO      /* Reply of Y */
           IF         ((&MSGRPY *EQ 'N') *OR (&MSGRPY *EQ 'n')) DO
           .
           .
           GOTO       END
           ENDDO      /* Reply of N */
           IF         (&MSGRPY *NE ' ') DO
           SNDPGMMSG  MSG('Reply was not Y or N, try again') +
                        TOMSGQ(QSYSOPR)
           GOTO       SNDMSG
           ENDDO      /* Reply not valid */
    /* Timeout occurred */
           SNDPGMMSG  MSG('No reply from the previous message +
                        was received in 120 seconds and a 'Y'' +
                        value was assumed') TOMSGQ(QSYSOPR)
           .
           .
END:       ENDPGM
```

The SNDUSRMSG command cannot be used instead in this procedure because it does not support a time-out option (SNDUSRMSG waits until it receives a reply or until the job is canceled).

The SNDPGMMSG command sends the message and specifies the KEYVAR parameter. This returns a message reference key, which uniquely identifies this message so that the reply can be properly matched with the RCVMSG command. The KEYVAR value must be defined as a character field length of 4.

The RCVMSG command specifies the message reference key value from the SNDPGMMSG command for the MSGKEY parameter to receive the specific message. The reply is passed back into the MSG parameter. The WAIT parameter specifies how long to wait for a reply before timing out.

When the reply is received, the procedure logic checks for an upper or lower case value of the Y or N. Normally the value is entered by the operator as a lower case value. If the operator enters a non-blank value other than Y or N, the procedure sends a different message and then repeats the inquiry message.

If the operator had entered a blank, no reply is sent to the procedure. If a blank is returned to the procedure, the time out occurred (the operator did not reply). The procedure sends a message to the system operator stating that a reply was not received and the default was assumed (the 'Y'' value is shown as 'Y' in the message queue). Because the assumed value of 'Y' is not displayed as the reply, you cannot determine when looking at a message queue whether the message should be answered or has already timed out. The procedure does not remove a message from the message queue once it has been sent. The second message should minimize this concern and provides an audit trail for what has occurred.

If the time out has already occurred and the operator replies to the message, the reply is ignored. The operator receives no indication that the reply has been ignored.

Sending Immediate Messages with Double-Byte Characters

To send an immediate message with double-byte text, limit the text to 37 double-byte characters plus the shift control characters. The limited size of the message ensures it is properly displayed.

Defining Default Values for Replies

The ADDMSGD command allows you to specify a default value for a reply to your message. A default reply must meet the same validity checking values as the other replies for the message or be specified as a special value in the message description. A default value is used when a user has indicated (using the CHGMSGQ command) that default replies should be issued for all inquiry messages sent to the user's message queue. Default replies are also sent when the unanswered inquiry messages are deleted. For example, the workstation user uses the DSPMSG command to display messages, and removes unanswered inquiry messages by pressing either F13 to delete all the messages or F11 to delete a particular message.

Default replies are also used when the job attribute of INQMSGRPY is set to *DFT and may be used if set to *SYSRPYL option. You can use the system reply list to change the default reply.

Default replies are also used on the Display Program Messages screen (which shows messages that are sent to `*EXT`). The sending of the default reply occurs during either of the two following conditions:

- The Display Program Messages screen appears showing an unanswered inquiry message and the user presses Enter (to continue) without typing any reply.
- The user pressed the F3 key to exit the **Display Program Messages** display.

.

Specifying Default Message Handling for Escape Messages

For each message you create that can be sent as an escape message, you can set up a default message handling action to be used if the message, when sent, is not handled any other way.

Default message handling actions can consist of:

- Default program name. A program to be called that takes default action to handle a message. The following parameters are passed to the default program:
  - Call message queue name. This parameter is a structure that consists of many fields that identify where the system sent the message. For information about default message handling exit programs and details on the fields in the parameter, refer to Message Handling APIs in the *APIs* section of the**Programming** category in the **iSeries Information Center**
  - Message reference key (4 characters). The message reference key of the escape message on the call message queue.
- Dump list. A list of message data field numbers (the same numbers as the substitution variables) that indicate which objects are to be dumped.

  In addition, you can dump any of the following:
  - The data areas for the job

- An internal machine data structure of a job
- A job

  Specifying a dump list for a job is equivalent to specifying the Display Job (DSPJOB) command with the parameters JOB(*) OUTPUT(*PRINT).

If you do not specify default actions in message descriptions, you will get a dump of the job (as if DSPJOB JOB(*) OUTPUT(*PRINT) was specified).

The default action specified in a message is taken only after the message percolation action is completed without the escape message being handled. See "Default Handling" on page 244 for more information on handling defaults.

Example of a Default Program

The following program is a sample default program that could be used when a diagnostic message is sent followed by an escape message. This program could be an OPM CL program or an ILE program that has this single CL procedure.

```
            PGM        PARM(&MSGQ &MRK)
            DCL        VAR(&MRK) TYPE(*CHAR) LEN(4)
            DCL        VAR(&MSGQ) TYPE(*CHAR) LEN(6381)
            DCL        VAR(&QNAME) TYPE(*CHAR) LEN(4096)
            DCL        VAR(&MODNAME) TYPE(*CHAR) LEN(10)
            DCL        VAR(&BPGMNAME) TYPE(*CHAR) LEN(10)
            DCL        VAR(&BLANKMRK) TYPE(*CHAR) LEN(4) VALUE(' ')
            DCL        VAR(&DIAGMRK) TYPE(*CHAR) LEN(4) VALUE(' ')
            DCL        VAR(&SAVEMRK) TYPE(*CHAR) LEN(4)
            DCL        VAR(&MSGID) TYPE(*CHAR) LEN(7)
            DCL        VAR(&MSGDTA) TYPE(*CHAR) LEN(100)
            DCL        VAR(&MSGF) TYPE(*CHAR) LEN(10)
            DCL        VAR(&MSGLIB) TYPE(*CHAR) LEN(10)
            DCL        VAR(&OFFSET) TYPE(*DEC)
            DCL        VAR(&LENGTH) TYPE(*DEC)

            /* Check for OPM program type                          */

            IF         (%SST(&MSGQ 277 1) *EQ '0') THEN(DO)
              CHGVAR     VAR(&QNAME) VALUE(%SST(&MSGQ 1 10))
              CHGVAR     VAR(&MODNAME) VALUE('*NONE')
              CHGVAR     VAR(&BPGMNAME) VALUE('*NONE')
             ENDDO
            ELSE DO
            /* Not an OPM program; always use the long procedure name */
              CHGVAR     VAR(&OFFSET) VALUE(%BIN(&MSGQ 281 4))
              CHGVAR     VAR(&LENGTH) VALUE(%BIN(&MSGQ 285 4))
              CHGVAR     VAR(&QNAME) VALUE(%SST(&MSGQ &OFFSET &LENGTH))
              CHGVAR     VAR(&MODNAME) VALUE(%SST(&MSGQ 11 10))
              CHGVAR     VAR(&BPGMNAME) VALUE(%SST(&MSGQ 1 10))
             ENDDO
GETNEXTMSG: CHGVAR     VAR(&SAVEMRK) VALUE(&DIAGMRK)
            RCVMSG     PGMQ(*SAME (&QNAME &MODNAME &BPGMNAME)) +
                         MSGTYPE(*DIAG) RMV(*NO) KEYVAR(&DIAGMRK)
            IF         (&DIAGMRK *NE &BLANKMRK) THEN(GOTO GETNEXTMSG)
            ELSE IF (&SAVEMRK *NE ' ') THEN(DO)
            /* If no diag message is sent, no message is sent to the previous program */
                RCVMSG     PGMQ(*SAME (&QNAME &MODNAME &BPGMNAME)) +
                           MSGKEY(&SAVEMRK) RMV(*NO) MSGDTA(&MSGDTA) +
                           MSGID(&MSGID) MSGF(&MSGF) MSGFLIB(&MSGLIB)
                SNDPGMMSG  MSGID(&MSGID) MSGF(&MSGLIB/&MSGF) +
                           MSGDTA(&MSGDTA) TOPGMQ(*PRV (&QNAME +
                           &MODNAME &BPGMNAME))
                           MSGTYPE(*ESCAPE)
              ENDDO
            ENDPGM
```

The program receives all the diagnostic messages in FIFO order. Then it sends the last diagnostic message as an escape message to allow the previous program to monitor for it.

Specifying the Alert Option

On the ADDMSGD command, you can specify an alert option to allow an alert to be created for a message. A message, for which an alert can be created, can cause an SNA alert to be created and sent to a problem management focal point. The alert created for a message can be defined using the Add Alert Description (ADDALRD) command. For more information about the OS/400 alerts support, see the DSNX Support book.

Example of Describing a Message

In the following example, the ADDMSGD command creates a message to be used in applications such as order entry. The message is issued when a customer number entered on the display is not found. The message is:

```
Customer number &1 not found
```

The ADDMSGD command for this message is:

```
ADDMSGD   MSGID(USR4310) +
          MSGF(QGPL/USRMSG)  +
          MSG('Customer number &1 not found') +
          SECLVL('Change customer number') +
          SEV(40) +
          FMT((*CHAR 8))
```

The message is added to the USRMSG file in the library QGPL.

You can use the DSPMSGD or WRKMSGD command to print or display message descriptions.

The SECLVL parameter provides very simple text. To make this appear on the Additional Message Information display, you specify `SECLVL('message text')`. The text you specify on this parameter appears on the Additional Message Information display when you press the Help key after placing the cursor on this message.

Defining Double-Byte Messages

To define a message with double-byte text, write a CL procedure or program using the ADDMSGD command. The defined message is put into a message file and then sent normally. When writing the program, do the following:

1. Make sure the source file containing the program is a double-byte file. Specify IGCDTA(*YES) on the Create Source Physical File (CRTSRCPF) command.
2. Use the source entry utility (SEU) to enter the program. CL commands using double-byte characters can only be entered through SEU. For this reason, double-byte messages must be created in a CL program.
3. Limit the length of the message to 37 double-byte characters, so the complete message can be displayed or printed.

   When using the MONMSG command, also limit the Comparison Data (CMPDATA) parameter to 6 double-byte characters.
4. If the double-byte message file replaces an alphanumeric message file (such as files of translated messages to be sent only to double-byte display stations), enter a command similar to the following to override the alphanumeric message file:

   ```
   OVRMSGF MSGF(QCPFMSG) TOMSGF(DBCSLIB/QCPFMSG)
   ```

   Double-byte messages can be displayed only at double-byte display stations.

System Message File Searches

The system uses the following two steps when searches are performed to retrieve a message from a message file:

1. The system processes any overrides that are in effect for the message file name.

   See "Overriding Message Files" for more information.

2. If the message file name has not been overridden, the system searches for the message file based on the message file name and library specified when the message was used.

   See "Searching for a Message File" for more information.

Searching for a Message File

When a message file has not been overridden, the message file name and library specified (at the time the message file was sent) are used to search for the message file from which the message description is retrieved.

When a message file name is overridden but the message identifier is not contained in the overridden file, the message file name and library specified are also used to search for the message file.

The system search depends on whether you specify the message file library as either *CURLIB or *LIBL. The following describes the search path for *CURLIB and *LIBL:

- Specify as *CURLIB or explicitly specify the message file library

  The system searches for the message file named in the specified library or the job's current library (*CURLIB).

- Specify the message file library as *LIBL

  The system searches for the message file named in the job's library list (*LIBL). The search stops after finding the first message file with the specified name.

If the message file is found, but does not contain a description for the message identifier, the message attributes and text of message CPF2457 in QCPFMSG are used in place of the missing message description.

If the message file was not found, the system attempts to retrieve the message from the message file that was used at the time the message was sent.

Note: A message file may be found but cannot be accessed due to damage or an authorization problem.

Overriding Message Files

You can override message files used in a procedure or program. The creation (Override Message File command), deletion (Delete Override command), and display (Display Override command) of message file overrides is similar to other types of overrides. Here, however, only the name of the message file, not the attributes, is overridden, and the rules for applying the overrides are slightly different.

To override a message file, use the Override Message File (OVRMSGF) command. The file overridden is specified in the MSGF parameter; the file overriding it is specified in the TOMSGF parameter.

For example, to override QCPFMSG with a user message file named USRMSGF, the following command would be used:

```
OVRMSGF MSGF(QCPFMSG) TOMSGF(USRMSGF)
```

When a predefined message is retrieved or displayed, the overriding file is searched for a message description. If the message description is not found in that file, the overridden file is searched.

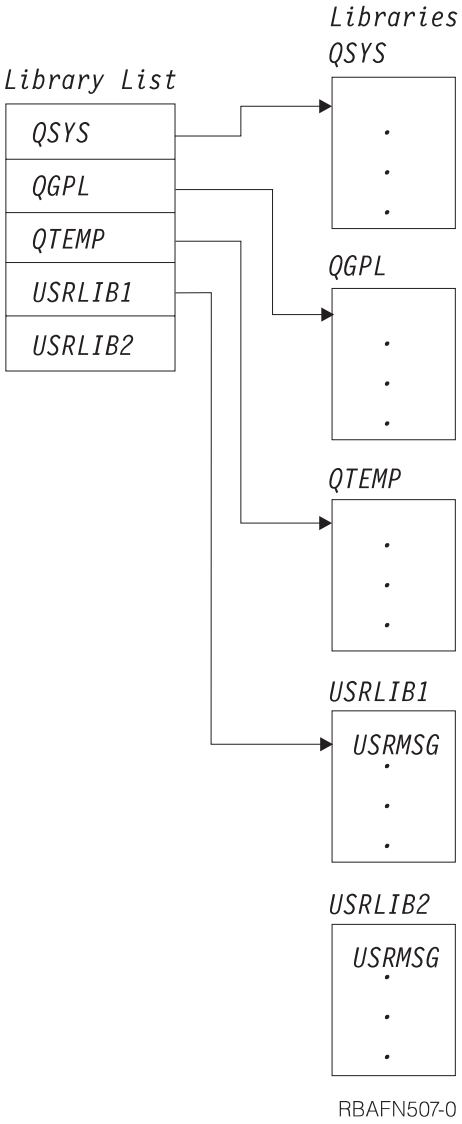
There are several basic reasons to override message files:

- To provide changed default replies or dump lists. A message file can be created with message descriptions for messages with changed default replies or dump lists because those in the original message descriptions are not satisfactory. You can establish several operating environments, each with different default replies.
- To change severity levels of the messages.
- To provide a default program.
- To change the text of a message. If the text is blank, it appears to the user as if no message was sent. For example, you may not want the status message sent by the Copy File (CPYF) command to appear to the user.
- To provide translation of messages into national languages. Message files written in English can be overridden by message files written in other languages. (If all messages are changed, use the library list for the job to change the order of the message files instead of overriding the message files.)

Another way you can select the message file from which messages are to be retrieved is by changing the order of the files in the library list for the job. However, if you use this approach, the search for the message stops on the first message file found that has the specified name. If the message is not in that file, the search stops.

For example, assume that a message file named USRMSG is in library USRLIB1, and another message file named USRMSG is in library USRLIB2. To use the

message file in USRLIB1, USRLIB1 should precede USRLIB2 in the library list:



The system searches the first message file found with the correct name. If that file does not contain the message, the search stops. However, if you use the OVRMSGF command, the system searches the overriding file, and if the message is not there, it searches the overridden file.

Example of Overriding a Message File

Assume that you want to change an IBM-supplied message for use in a job. For example, suppose you want to change message CPC2191, which says:

```
Object XXX in YYY type *ZZZ deleted
```

to say:

```
Object XXX in YYY deleted
```

Specifics on how to describe the FMT parameter are provided by displaying the detailed description of CPC2191.

First, you create a message file:

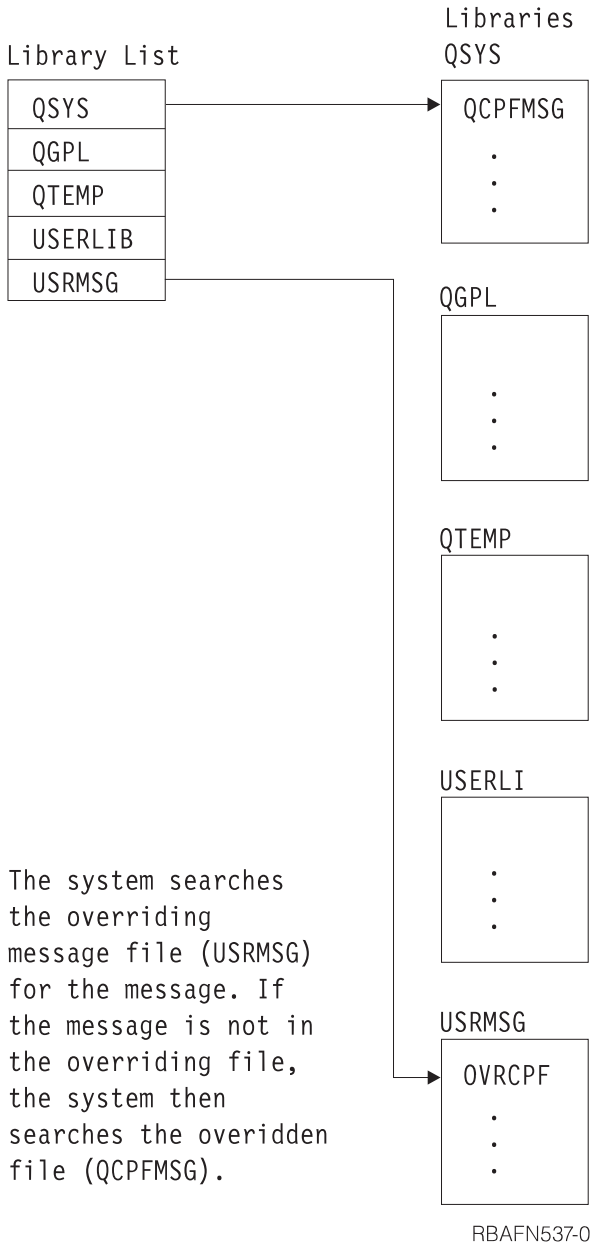
```
CRTMSGF MSGF(USRMSG/OVRCPF)
```

Then you use the message CPC2191 as a basis for your message and add it to the message file:

```
ADDMSGD  MSGID(CPC2191) MSGF(USRMSG/OVRCPF) +
         MSG('Object &1 in &2 deleted') +
         SEV(00) FMT((*CHAR 10) (*CHAR 10))
```

You then use the OVRMSGF command to override the message file when you run the job:

```
OVRMSGF MSGF(QCPFMSG) TOMSGF(USRMSG/OVRCPF)
```



RBAFN537-0

If you want to change this message for use in all your jobs, you can use the Change Message Description (CHGMSGD) command to change the message. Then you do not have to override the system message file.

If you use the CHGMSGD command to change an IBM-supplied message, the message will need to be changed again when a new release of the system is

installed. To change the message again, you can place any changes in an input stream or a program that can be run at any time.

You can also override overriding files. For example, you can specify the following OVRMSGF commands during a job.

```
OVRMSGF  MSGF(MSGFILE1)  TOMSGF(MSGFILE2)
OVRMSGF  MSGF(MSGFILE2)  TOMSGF(MSGFILE3)
```

First, file MSGFILE1 was overridden with MSGFILE2. Second, MSGFILE2 was overridden with MSGFILE3. When a message is sent, the files are searched in this order:

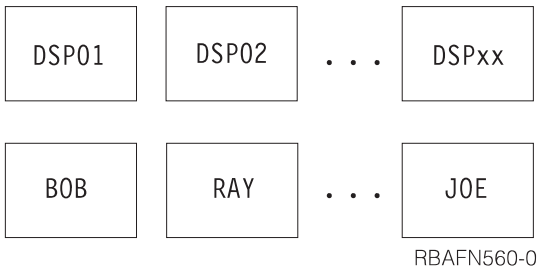
1. MSGFILE3
2. MSGFILE2
3. MSGFILE1

You can prevent message files from being overridden. To do so, you must specify the SECURE parameter on the OVRMSGF command.

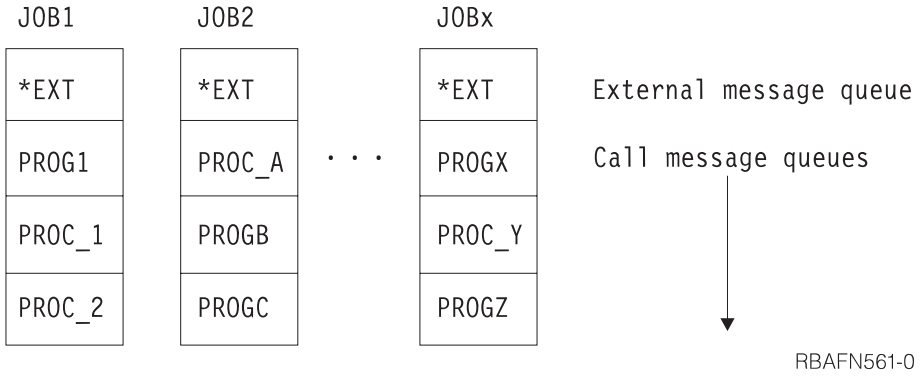
Types of Message Queues

All messages on the system are sent to a message queue. The system user or program associated with the message queue receives the message from the queue. Similarly, a reply to a message is sent back to the message queue of the user or program requesting the reply.

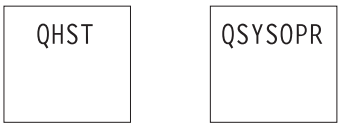
The following diagrams show the message queues supplied by IBM. A message queue is supplied for each display station (where DSP01 and DSP02 are display station names) and each user profile (where BOB and RAY are user profile names):



Job message queues are supplied for each job running on the system. Each job is given an external message queue (*EXT) and each call of an OPM program or ILE procedure within the job has its own call message queue.



Message queues are also supplied for the system history log (QHST) and the system operator (QSYSOPR):



RBAFN562-0

These message queues are used as follows:

- Workstation message queues are used for sending and receiving messages between workstation users and between workstation users and the system operator. The name of the queue is the same as the name of the workstation. The queue is created by the system when the workstation is described to the system.
- User profile message queues can be used for communication between users. User profile message queues are automatically created in library QUSRSYS when the user profile is created.
- Job message queues are used for receiving requests to be processed (such as commands) and for sending messages that result from processing the requests; the messages are sent to the requester of the job. Job message queues exist for each job and only exist for the life of the job. Job message queues consist of an external message queue (*EXT) and a set of call stack entry message queues. See "Job Message Queues" on page 206 for more information.
  - System operator message queue (QSYSOPR) is used for receiving and replying to messages from the system, display station users, and application programs.
  - The history log message queue is used for sending information to the history log (QHST) from any job in the system.

  In addition to these message queues, you can create your own user message queues for sending messages to system users and between application programs.

Creating or Changing a Message Queue

To create your own user message queues, you use the Create Message Queue (CRTMSGQ) command. In addition, you also use the Change Message Queue (CHGMSGQ) command to change the following attributes of your message queue.

The attributes of a message queue are:

- Whether changes to the message queue must be written immediately to the disk. Writing the changes immediately to the disk ensures that no messages are lost in cases like a system failure. Note that this will cause a decrease in system performance.
  - The method of delivery for messages arriving at a message queue. When a message queue is created, the method of delivery is defined as hold delivery. When a display station is signed on, the user's message queue is set to the mode specified in the user profile. The types of delivery you can specify on the CHGMSGQ command are:
    - Break delivery. A job is interrupted and a program is called to deliver the message. If a user program is not specified on the CHGMSGQ command that requests break delivery, or if *SAME is specified, the Display Message (DSPMSG) command automatically displays the message. Break messages for a job can be controlled with the BRKMSG parameter on the CHGJOB command.

- Notify delivery. Adisplay station user is notified by means of the Attention light or audible alarm (or by both) that a message is on the queue. The display station user can view the message by using the DSPMSG command.
  - Hold delivery. The message queue holds the messages until the display station user requests them with the DSPMSG command.
    - Default delivery. All messages are ignored, and any messages requiring a reply are sent the default reply for the message.

- How to handle messages for break delivery.
  - Automatically run the DSPMSG command. For an interactive job, the messages are displayed at the display station if the severity code is high enough. For a batch job, the messages are listed to a spooled printer file if the severity code is high enough.
  - Call a break-handling program to handle the messages. You must use the CHGMSGQ command to specify the called program and to set the method of delivery to break mode. You can specify whether other jobs can reply to inquiry messages on the queue while it is in break mode with a break-handling program.
  - The severity code for filtering messages for break and notify delivery. Messages with severity equal to or greater than the minimum severity code specified are displayed. When the queue is created, the minimum severity code is set to 00. To change the minimum severity code, you must use the CHGMSGQ command.

    When the DSPMSG command is used to display messages on the message queue, the severity code filter (SEV) parameter can be used to filter the messages shown. This filter is used rather than the severity filter specified for the message queue at creation time. To use this filter, specify DSPMSG SEV(*MSGQ). You can use the DSPMSG command to determine the current severity code used for filtering break and notify messages. The code is displayed on the heading line of the message display.
    - Coded character set identifier (CCSID) associated with the message queue. Messages sent to this queue are converted to this CCSID. No conversions occur if the message queue CCSID is 65534 or 65535. If the message queue CCSID is 65534, each message contains its own CCSID which is established by the sender.
      - Allow alerts for standard message queues. Allow alerts specifies if the queue being created allows alerts to be generated from alert messages that are sent to it.
      - Action to take when the message queue becomes full. You cannot change this attribute for message queue QHST; QHST sends CPF2460 when it is full. IBM ships QSYSOPR with this attribute that is originally set to wrap.
        - Send CPF2460 (Message queue cannot be extended) to the program or user that sends a message to the full queue.
        - Wrap the queue. Wrapping will remove messages on the queue to make space for a new message that is sent to the queue.

      Note: When a workstation device description is created, the system establishes a message queue for the device to receive all action messages for the device. For workstation printers, tape drives, and APPC devices, the MSGQ parameter can be used to specify a message queue when creating a device description. If no message queue is specified for these devices, the default,

QSYSOPR, is used as the message queue. All other devices are assigned to the QSYSOPR message queue when they are created.

The message queue defined in your user profile is known as a user message queue. When you sign on the system using your profile, the user message queue is put into the delivery mode specified in your user profile.

If your user message queue is in break or notify delivery mode while you are signed on a display station and then you sign on another display station, the user message queue will not change the delivery mode for the new sign on. User message queues (along with workstation message queues and the QSYSOPR message queue) cannot have their delivery mode changed by a job when the message queue is in break or notify delivery mode for a different job.

When you sign off the display station, or the job ends unexpectedly, the user message queue delivery mode is changed to hold mode, if the delivery mode of the user message queue is break or notify for this job. The user message queue delivery mode is also changed from break or notify mode to hold mode when you transfer to an alternative job. You can do this using the Transfer Secondary Job (TFRSECJOB) command or by pressing the System Request key and specifying option 1 on the System Request menu.

After transferring to an alternative job, you sign on using your user profile. Your user message queue is put into the delivery mode specified in your user profile. This allows the user message queue to transfer to the alternative job. You are then able to transfer back and forth between these two jobs and have your user message queue follow you.

However, if after transferring to an alternative job, you sign on using a user profile other than your own, the user message queue for the job from which you transferred is left in hold delivery mode. The user message queue for the user profile you signed on with is put in the delivery mode specified in that user profile. Because of this, your user message queue could be put into break or notify delivery mode by another user. If another user still has your user message queue in that delivery mode when you transfer back to the first job, your user message queue delivery mode cannot be changed back to the original delivery mode.

The QSYSOPR message queue is the message queue for the system operator, unless it has been changed. The above situation can occur for a system operator as well.

Message Queues in Independent ASPs

See the Independent ASPs article for details about the independent ASPs.

It is recommended that message queues in independent ASPs should not be put into break mode. When a message queue is in break mode, the break program will not be called if the message queue is not in the thread's library name space when a message is sent to the message queue. The libraries in the independent ASPs in the thread's ASP group plus the libraries in the system ASP (ASP number 1) and basic user ASPs (ASP numbers 2-32) form the library name space for the thread.

When sending an inquiry message to a message queue, the to message queue and the reply message queue both should be either in the system ASP or in the same independent ASP, otherwise the reply may not be sent to the reply message queue if either message queue was taken offline.

Messages cannot be received in these situations:

- from a message queue with a wait time, in an independent ASP that is varied off
- as a reply to an inquiry message sent to a message queue, in an independent ASP that is varied off

A break handing program will not be able to change the library name space for the thread.

Break-Handling Program

A break-handling program is called whenever a message of equal or higher severity than the severity code filter arrives on a message queue that is in break delivery mode. To request a break-handling program, you must specify the name of the program and break delivery on the same CHGMSGQ command. The message handling program must receive the message with the Receive Message (RCVMSG) command so the message is marked as handled and the program is not called again. For more information on receiving messages and break handling programs, see Chapter 8, "Working with Messages."

Note: This program cannot open a display file if the interrupted program is waiting for input data from the device display.

You can use the system reply list to specify that the system issue the reply to specified predefined inquiry messages so that the display station user does not need to reply. See "Using the System Reply List" on page 269 for more information.

Example of Changing the Delivery Mode

When the system is started, it puts the QSYSOPR message queue in break delivery when the controlling subsystem is started. However, if the system operator signs off, the message queue is put in hold delivery. When the system operator signs on again, QSYSOPR is placed in the mode specified in the QSYSOPR user profile.

The following procedure in a CL initial program can be used to place the QSYSOPR message queue in break mode. Initial programs can use similar procedures to monitor message queues other than the one specified in a user's own user profile.

```
PGM /* Procedure to place a msg queue in break mode */
CHGMSGQ  QSYSOPR DLVRY(*BREAK) SEV(50)
MONMSG   MSGID(CPF0000) EXEC(SNDPGMMSG MSG('Unable to put QSYSOPR +
         message queue in *BREAK mode') TOPGMQ(*EXT))
ENDPGM
```

The procedure attempts to set the QSYSOPR message queue to break delivery with a severity level of 50. If this is unsuccessful, a message is sent to the external job message queue (*EXT). When the program which contains this procedure ends, the initial menu is displayed. A severity level of 50 is used to decrease the number of break messages that interrupts the workstation user. A common reason for failure is when another user has QSYSOPR in break mode already.

Job Message Queues

Job message queues are created for each job on the system to handle all the message requirements of the job. Job message queues for a single job consist of an external message queue (*EXT) and a set of call message queues. A call message queue is assigned to each ILE procedure and OPM program that is called within the job. In addition, a job log is created for each job. A job log is a logical queue which maintains all messages sent within a job in chronological order. You may send messages to the *EXT queue or to a call message queue. You do not send messages to the job log. Rather a message sent to either *EXT or a call message queue is also logically added to the job log by the system.

The external message queue (*EXT) is used to communicate with the external requester (such as a display station user) of the job. Messages (except status messages) sent to the external message queue of a job are also placed on the job log (see "Job Log" on page 272 for more information).

If an informational, inquiry, or notify message is sent to the external message queue for an interactive job, the message is displayed on the Display Program Messages display. Additionally, the procedure waits for a reply to inquiry or notify messages from the display station user. Should the user not enter a reply and press the Enter key or F3 (Exit), the default message reply is returned to the sender of the message. If there is no default message reply, *N is sent. If you send an inquiry or notify message to the external message queue for a batch job, the system sends the default reply back to you. If there is no default message reply, *N is the reply. The system reply list may override the displaying of inquiries or the sending of default replies to inquiries to *EXT.

If a status message is sent to the external message queue of an interactive job, the message is displayed on the message line of the display station. You can use status messages like this to inform the display station user of the progress of a long-running operation. For example, the system sends status messages when running the CPYF command if you copy a file with several members.

Note: When your application completes the long-running operation, you must send another message to clear the message line at the display. You can use message CPI9801, which is a blank message, for this purpose. For example:

```
PGM
.
.
.
SNDPGMMSG  MSGID(CPF9898)  MSGF(QCPFMSG)  MSGDTA('Status 1')  +
           TOPGMQ(*EXT)  MSGTYPE(*STATUS)
.
.
.
SNDPGMMSG  MSGID(CPF9898)  MSGF(QCPFMSG)  MSGDTA('Status 2')  +
           TOPGMQ(*EXT)  MSGTYPE(*STATUS)
.
.
.
SNDPGMMSG  MSGID(CPI9801)  MSGF(QCPFMSG)  TOPGMQ(*EXT)  +
           MSGTYPE(*STATUS)
.
.
.
ENDPGM
```

A call message queue is used to send messages between one program or procedure and another program or procedure. As long as a program or procedure is on the call stack (has not returned yet) its call message queue is active and messages can be sent to that program or procedure. Once the program or procedure returns, its call message queue no longer exists and messages can no longer be sent to it. Message types which can be sent to a call message queue include informational, request, completion, diagnostic, status, escape, and notify.

The call message queue for an OPM program or ILE procedure is created when that program or procedure is called. The call message queue is exclusively associated only with the call stack entry in which the program or procedure is running. A call message queue is identified indirectly by identifying the call stack entry. A call stack entry is identified by the name of the program or procedure that is running in that call stack entry.

In the case of an OPM program, the associated call stack entry is identified by the (up to) 10 character program name. In the case of an ILE procedure, the associated call stack entry is identified by a three part name which consists of the (up to) 256 character procedure name, the (up to) 10 character module name, and the (up to) 10 character program name. The module name is the name of the module into which the procedure was compiled. The ILE program name is the name of the ILE program into which the module was bound.

When identifying the call stack entry for an ILE procedure, it is sufficient to specify only the procedure name. If the procedure name by itself does not uniquely identify the call stack entry, the module name or the ILE program name can also be specified. If, at the time a message is sent, a program or procedure is on the call stack more than once, the name specified will identify the most recently called occurrence of that program or procedure.

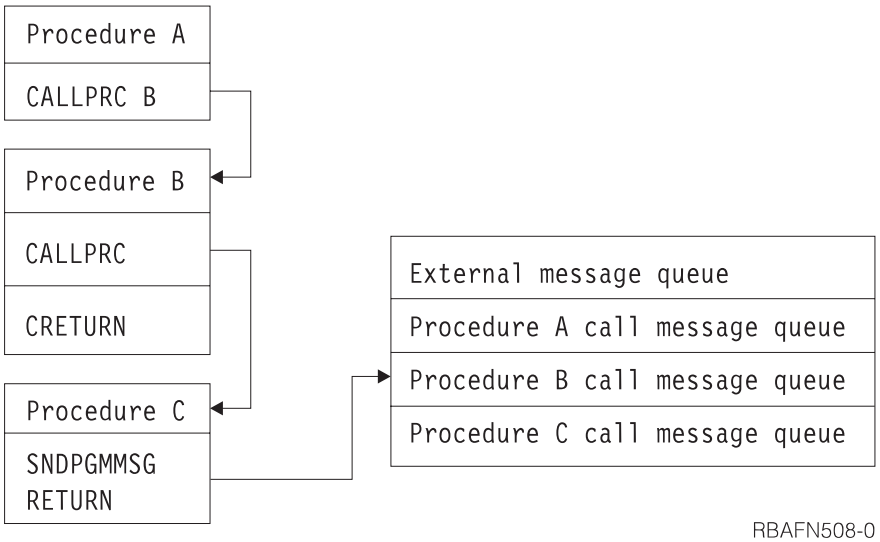
There are other methods to identify a call stack entry. These methods are discussed in detail in “Call Stack Entry Identification on SNDPGMMSG” on page 218.

If an OPM or ILE program is compiled and then replaced while it is on the call stack, care must be taken when the program name is used to reference a call stack entry. For call stack entries that are earlier on the stack than the point at which the replace operation was done, the name reference will resolved to the replaced object which now exists in QRPLOBJ. These name references are valid as long as the replaced object continues to exist in the QRPLOBJ library. For entries on the stack that are more recent then the point at which the replace operation was done, the name reference is for the new version of the program. Because of the manner in which the version to use is determined, you should not place a program directly in the library QRPLOBJ. This library should be used exclusively for the replaced version of a program. A name reference to a program that you place directly into QRPLOBJ will fail.

If a program object is removed or renamed while an occurrence of it is on the call stack, any name reference to the removed program or any name reference using the old name will fail. For ILE procedures, if you are using only the procedure and module name for a reference, renaming the program will not impact the name reference. If you are also using the ILE program name, the name reference will fail.

A message queue for a call stack entry of a program or procedure is no longer available when the program or procedure ends. A message that was on the associated call message queue can only be referenced at that point by using the message reference key of the message.

For example, assume that procedure A calls procedure B which calls procedure C. Procedure C sends a message to procedure B and ends. The message is available to procedure B. However, when procedure B ends, its call message queue is no longer available. As a result, you cannot access procedure B by using procedure A, even though the message appears in the job log. Procedure A cannot access messages that are sent to Procedure B unless Procedure A has the message reference key to that message.

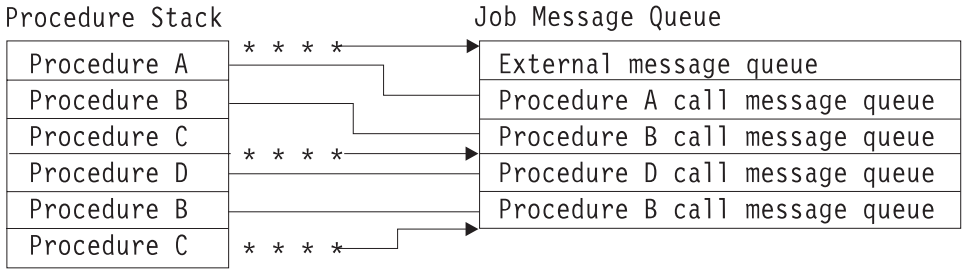


RBAFN508-0

If procedure A needs to delete specific messages, you could do the following:

- Have procedure C send specific messages to procedure A
- Have procedure B resend the messages to procedure A

The following figure shows the relationship of procedure calls, the job message queue, and the call stack entry queues. A connecting line (-----) indicates which message queue is associated with which call of a procedure.



RBAFN532-0

In the preceding figure, procedure B has two call stack entry queues, one for each call of the procedure. There are no message queues for procedure C because no messages were sent to procedure C. When procedure C sends a message to procedure B, the message goes to the call stack entry queue for the last call of procedure B.

Note: When you are using the command entry display, you can display all the messages sent to the job message queue by pressing F10 (Include detailed messages). Once the messages are displayed, you can roll through them using one of the roll keys.

You can also display the messages for a job by using the Display Job Log (DSPJOBLOG) command.

Chapter 8. Working with Messages

This chapter discusses some of the ways that messages can be used to communicate between users and programs. Messages can be sent:

- From one system user to another system user, even if the receiver of the messages is not currently using the system
- From one OPM program or ILE procedure to another OPM program or ILE procedure
- From a program or procedure to a system user, even if the receiver of the messages is not currently using the system

Interactive system users can send only immediate messages and replies.

OPM programs or ILE procedures can send immediate messages or predefined messages with user-defined data. In addition, programs or procedures can:

- Receive messages
- Retrieve a message description from a message file and place it into a program variable
- Remove messages from a message queue
- Monitor for messages

Sending Messages to a System User

Several commands can be used to send messages to system users:

- Send Message (SNDMSG)
- Send Break Message (SNDBRKMSG)
- Send Program Message (SNDPGMMSG)
- Send User Message (SNDUSRMSG)

SNDPGMMSG and SNDUSRMSG can only be used in batch or interactive OPM programs or ILE procedures. These commands cannot be entered on a command line. The SNDMSG command sends an informational or inquiry message to the system operator message queue (QSYSOPR), a display station message queue, or a user message queue. You can send an informational message to more than one message queue at a time. But you can send an inquiry message to only one message queue at a time. The message is delivered by the delivery type specified for the message queue. The message does not interrupt the user unless the message queue is in break mode.

The following SNDMSG command is sent by a display station user to the system operator:

```
SNDMSG   MSG('Mount tape on device TAP1') TOUSR(*SYSOPR)
```

The SNDBRKMSG command sends an immediate message from a workstation, a program, or a job to one or more display stations to be delivered in the break mode regardless of what delivery mode the receiver's message queue is set to. This command can be used to send a message only to display station message queues. You should use the SNDBRKMSG command when sending any message that requires the immediate attention of a display station user. You cannot ensure the

© Copyright IBM Corp. 1997, 2004

message will cause a break, because each job has control by using the BRKMSG parameter on the Change Job (CHGJOB) command.

If you send an inquiry message, you can specify that the reply be sent to a message queue other than that of your display station.

The following SNDBRKMSG command is sent by the system operator to all the display station message queues:

```
SNDBRKMSG MSG('System going down in 15 minutes')
          TOMSGQ(*ALLWS)
```

The disadvantage of sending this message is that it is sent to *all* users, not just those users who are active at the time the message is sent.

Sending Messages from a CL Program

Use the Send Program Message (SNDPGMMSG) command or the Send User Message (SNDUSRMSG) command to send a message from a CL procedure or program.

Using the SNDPGMMSG command, you can send the following types of messages:

- Informational
- Inquiry
- Completion
- Diagnostic
- Request
- Escape
- Status
- Notify

You can send messages from a CL procedure or program to the following types of queues:

- External message queue of the requester of the job (see "Job Message Queues" on page 206)
- Call message queue of a program or procedure called by the job (see "Job Message Queues" on page 206)
- System operator message queue
- Workstation message queue
- User message queue

To send a message from a procedure or program, you can specify the following on the SNDPGMMSG command:

- Message identifier or an immediate message. The message identifier is the name of the message description for a predefined message.
- Message file. The name of the message file containing the message description when a predefined message is sent.
- Message data fields. If a predefined message is sent, these fields contain the values for the substitution variables in the message. The format of each field must be described in the message description. If an immediate message is sent, there are no message data fields.
- Message queue or user to receive the message.

- Message type. The following indicates which types of messages can be sent to which types of queues (V = valid).

Table 8. Valid Message Types for Message Queue Types

Message Type	Message Queue Type				
	External	**Call**	**QSYSOPR**	**WorkStation**	**User**
Informational	V	V	V	V	V
Inquiry	V		V	V	V
Completion	V	V	V	V	V
Diagnostic	V	V	V	V	V
Request	V	V			
Escape		V			
Status	V	V			
Notify	V	V			

- Coded character set identifier (CCSID). Specifies the coded character set identifier (CCSID) that the supplied message or message data is in.
- Reply message queue. The name of the message queue that receives the reply to an inquiry message. By default, the reply is sent to the call message queue of the procedure or program that sent the inquiry message.
- Key variable name. The name of the CL variable to contain the message reference key for a message.

To send the message created in "Example of Describing a Message" on page 196, you would use the following command:

```
SNDPGMMSG   MSGID(USR4310) MSGF(QGPL/USRMSG) +
            MSGDTA(&CUSNO) TOPGMQ(*EXT) +
            MSGTYPE(*INFO)
```

The substitution variable for the message is the customer number. Because the customer number varies, you cannot specify the exact customer number in the message. Instead, declare a CL variable in the CL procedure or program for the customer number (&CUSNO). Then specify this variable as the message data field. When the message is sent, the current value of the variable is passed in the message:

```
Customer number 35500 not found
```

In addition, you do not always know which display station is using the procedure or program, so you cannot specify the exact display station message queue that the message is to be sent to (TOPGMQ parameter); therefore, you specify the external message queue *EXT.

Messages

Inquiry and Informational Messages

Using the SNDUSRMSG command, you can send an inquiry message or an informational message to a display station user, the system operator, or a user-defined message queue. If you use the SNDUSRMSG command to send an inquiry message to the user, the procedure or program waits for a response from the user. The message can be either an immediate message or a predefined message. For an interactive job, the message is sent to the display station operator by default. For a batch job, the message is sent to the system operator by default.

To send a message from a procedure or program using the SNDUSRMSG command, you can specify the following on the SNDUSRMSG command:

- Message identifier or an immediate message. The message identifier is the name of the message description for a predefined message.
- Message file. The name of the message file containing the message description when a predefined message is sent.
- Message data fields. If a predefined message is sent, these fields contain the value for the substitution variables in the message. The format of each field must be described in the message description. If an immediate message is sent, there are no message data fields.
- Valid replies to an inquiry message.
- Default reply value to an inquiry message.
- Message type.
- Message queue to which the message is to be sent.
- Message reply. A CL variable, if any, that is to contain the reply received in response to an inquiry message.
- Translation table. The translation table to be used, if any, to translate the reply value. This is normally used for translating lowercase to uppercase.
- Coded character set identifier (CCSID). Specifies the coded character set identifier (CCSID) that the supplied message or message data is in.

Completion and Diagnostic Messages

Using the SNDPGMMSG command, you can send diagnostic and completion messages. You can send these message types to any message queue from your CL procedure or program. Diagnostic messages tell the calling program or procedure about errors detected by the CL procedure or program. Completion messages tell the results of work done by the CL procedure or program.

Normally, an escape message is sent to the message queue of the calling program or procedure to tell the caller what the problem was or that diagnostic messages were also sent. For a completion message, an escape message is usually not sent because the requested function was performed.

For an example of sending a completion message, assume that the system operator uses the system operator menu to call a CL program SAVPAY to save certain objects. The CL program contains only the following procedure which saves the objects and then issues the following completion message:

```
PGM
SAVOBJ OBJ(PAY1 PAY2) LIB(PAYROLL) CLEAR(*YES)
SNDPGMMSG MSG('Payroll objects have been saved') MSGTYPE(*COMP)
ENDPGM
```

If the SAVOBJ command fails, the CL procedure function checks and the system operator has to display the detailed messages to locate the specific escape message explaining the reason for the failure as described later in this chapter. If the SAVOBJ command completes successfully, the completion message is sent to the call message queue associated with the program that displays the system operator menu.

One of the advantages of completion messages is their consistency with IBM-supplied commands. Many IBM commands send completion messages indicating successful completion. Seeing the type of message sent to the job log can assist in problem analysis.

Status Messages

You can send status messages from your CL procedure or program, using the SNDPGMMSG command, to a call message queue or to the external message queue (*EXT) for the job. When a status message is sent to a call message queue, the receiving program or procedure can monitor for the arrival of the status message and can handle the condition it describes. If the receiving program or procedure does not monitor for the message, control returns to the sender to resume processing. See "Monitoring for Messages in a CL Program or Procedure" on page 239.

Escape and Notify Messages

You can send escape messages from your CL procedure or program to the call message queue of the calling program or procedure with the SNDPGMMSG command. An escape message tells the caller that the procedure or program ended abnormally and why. The caller can monitor for the arrival of the escape message and handle the condition it describes. When the caller handles the condition, control does not return to the sender of an escape message.

If the caller is another procedure within the same program, the program itself does not end. The procedure to which the escape message was sent is allowed to continue. If the escape message was sent to the caller of the program itself, then all active procedures within the program are ended immediately. As a result, the program cannot continue to run. If the caller does not monitor for an escape message, default system action is taken.

You can send notify messages from a CL procedure or program to the message queue of the calling program or procedure or to the external message queue. A notify message tells the caller about a condition under which processing can continue. The calling program or procedure can monitor for the arrival of the notify message and handle the condition it describes. If the caller is an Integrated Language Environment procedure, it can perform the following functions:

- It can handle the condition.
- It can send a reply back to the caller.
- It can allow the sending procedure to continue processing.

If the caller is an OPM program and is not monitoring for the message, the sender receives a default reply. If the caller is an ILE procedure, then the message percolates to the control boundary. When finding no monitor, the system returns a default reply to the sender. The sender then resumes processing. See "Monitoring for Messages in a CL Program or Procedure" on page 239.

Immediate messages are not allowed as escape and notify messages. The system has defined the message CPF9898, which can be used for immediate escape and notify messages in application programs. For example:

```
SNDPGMMSG MSGID(CPF9898) MSGF(QCPFMSG) MSGDTA('Error condition') +
          MSGTYPE(*ESCAPE)
```

Examples of Sending Messages

Example 1: The following CL procedure allows the display station user to submit a job by calling a CL program which contains this procedure instead of entering the Submit Job (SBMJOB) command. The procedure sends a completion message when the job has been submitted.

```
PGM
SBMJOB JOB(WKLYPAY) JOBD(USERA) RQSDTA('CALL WKLY PARM(PAY1)')
SNDPGMMSG MSG('WKLYPAY job submitted') MSGTYPE(*COMP)
ENDPGM
```

Example 2: The following CL procedure changes a message based on a parameter received from a program that is called from within the this procedure. The message is then sent by the CL procedure as a completion message. (The RCDCNT field is defined as characters in PGMA.)

```
PGM
DCL &RCDCNT TYPE(*CHAR) LEN(3)
CALL PGMA PARM(&RCDCNT)
SNDPGMMSG MSG('PGMA completed' *BCAT &RCDCNT *BCAT +
          'records processed') MSGTYPE(*COMP)
ENDPGM
```

Example 3: The following procedure sends a message requesting the system operator to load a special form. The Receive Message (RCVMSG) command waits for the reply. The system operator must enter at least 1 character as a reply to the inquiry message, but the procedure does not use the reply value.

```
PGM
DCL &MSGKEY TYPE(*CHAR) LEN(4)
SNDPGMMSG   MSG('Load special form') TOUSR(*SYSOPR) +
            KEYVAR(&MSGKEY) MSGTYPE(*INQ)
RCVMSG MSGTYPE(*RPY) MSGKEY(&MSGKEY) WAIT(120)
.
.
.
ENDPGM
```

The WAIT parameter must be specified on the RCVMSG command so that the procedure waits for the reply. If the WAIT parameter is not specified, the procedure continues with the instruction following the RCVMSG command, without receiving the reply. The MSGKEY parameter is used in the RCVMSG command to allow the procedure to receive the reply to a specific message. The variable &MSGKEY in the SNDPGMMSG command is returned to the procedure for use in the RCVMSG command.

Example 4: The following procedure sends a message to the system operator when it is run in batch mode or to the display station operator when it is run from a display station. The procedure accepts either an uppercase or lowercase Y or N. (The lowercase values are translated to uppercase by the translation table (TRNTBL parameter) to make program logic easier.) If the value entered is not one of these four, the operator is issued a message indicating the reply is not valid.

```
PGM
DCL &REPLY *CHAR LEN(1)
.
.
SNDUSRMSG MSG('Update YTD Information Y or N') VALUES(Y N) +
     MSGRPY(&REPLY)
IF (&REPLY *EQ Y)
 DO
 .
 .
 .
 ENDDO
ELSE
 DO
 .
 .
 ENDDO
 .
 .
 .
ENDPGM
```

Example 5: The following procedure uses the message CPF9898 to send an escape message. The text of the message is 'Procedure detected failure'. Immediate messages are not allowed as escape messages so message CPF9898 can be used with the message as message data.

```
   PGM
   .
   .
   .
SNDPGMMSG MSGID(CPF9898) MSGF(QCPFMSG) MSGTYPE(*ESCAPE)
     MSGDTA('Procedure detected failure')
   .
   .
   ENDPGM
```

Example 6: The following procedure allows the system operator to send a message to several display stations. When the system operator calls the program, this procedure, contained within the called program, displays a prompt which the system operator can enter the type of message to be sent and the text for the message. The procedure concatenates the date, time, and text of the message.

```
           PGM
           DCLF WSMSGD
           DCL &MSG TYPE(*CHAR) LEN(150)
           DCL &HOUR TYPE(*CHAR) LEN(2)
           DCL &MINUTE TYPE(*CHAR) LEN(2)
           DCL &MONTH TYPE(*CHAR) LEN(2)
           DCL &DAY TYPE(*CHAR) LEN(2)
           DCL &WORKHR TYPE(*DEC) LEN(2 0)
           SNDRCVF RCDFMT(PROMPT)
           IF &IN91 RETURN /* Request was ended  */
           RTVSYSVAL QMONTH RTNVAR(&MONTH)
           RTVSYSVAL QDAY RTNVAR(&DAY)
           RTVSYSVAL QHOUR RTNVAR(&HOUR)
           IF (&HOUR *GT '12') DO
           CHGVAR &WORKHR &HOUR
           CHGVAR &WORKHR (&WORKHR - 12)
           CHGVAR &HOUR &WORKHR /* Change from military time */
           ENDDO
           RTVSYSVAL QMINUTE RTNVAR(&MINUTE)
           CHGVAR    &MSG ('From Sys Opr ' *CAT &MONTH *CAT '/' +
                     *CAT &DAY +
                     *BCAT &HOUR *CAT ':' *CAT &MINUTE +
                     *BCAT &TEXT)
           IF (&TYPE *EQ 'B') GOTO BREAK
NORMAL:    SNDPGMMSG MSG(&MSG) TOMSGQ(WS1 WS2 WS3)
           GOTO ENDMSG
BREAK:     SNDBRKMSG MSG(&MSG) TOMSGQ(WS1 WS2 WS3)
ENDMSG:    SNDPGMMSG MSG('Message sent to display stations') +
                     MSGTYPE(*COMP)
           ENDPGM
```

The DDS for the display file, WSMSGD, used in this program follows:

```
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
     A                                     DSPSIZ(24 80)
     A          R PROMPT                   TEXT('Prompt')
     A                                     BLINK
     A                                     CA03(91 'Return')
     A                                   1  2'Send Messages To Workstations'
     A                                       DSPATR(HI)
     A                                   3  2'TYPE'
     A            TYPE           1   1    +2VALUES('N' 'B')
     A                                     CHECK(ME)
     A                                     DSPATR(MDT)
     A                                   +3'(N = No breaks  B = Break)'
     A                                   5  2'Text'
     A            TEXT         100   1    +2LOWER
     A
     A
```

If the system operator enters the following on the prompt:

```
B

Please sign off by 3:30 today
```

the following break message is sent:

```
From Sys Opr 10/30 02:00  Please sign off by 3:30 today
```

Call Stack Entry Identification on SNDPGMMSG

If the CL procedure is to send a message to an OPM program or another ILE procedure, you must identify the call stack entry to which the message is sent. The message is sent to the call message queue of the identified call stack entry.

The TOPGMQ parameter of the SNDPGMMSG command is used to identify the call stack entry to which a message is sent. Identification of a call stack entry consists of the following two parts:

- Specification of a base entry

  The specification TOPGMQ(*PRV *) identifies the base entry as being the one in which the procedure using the SNDPGMMSG command is running. The offset is specified as being one entry previous to that base. This specification identifies the caller of the procedure which is using the command.
- Offset specification of a base entry

  The offset specification (element 1 of TOPGMQ) identifies if you send the message to the base (*SAME) or if you send the message to the caller of the base (*PRV).

To understand how to identify the base entry, element 2 of TOPGMQ, you also need to understand the call stack when an ILE program is running. Two programs are used to illustrate this. Program CLPGM1 is an OPM CL program and Program CLPGM2 is an ILE program. Since program CLPGM2 is ILE, it can consist of several procedures, such as: CLPROC1, CLPROC2, CLPROC3, and CLPROC4. At runtime the following calls take place:

- CLPGM1 is called first.
- CLPGM1 calls CLPGM2.
- CLPGM2 calls CLPROC1.
- CLPROC1 calls CLPROC2.
- CLPROC2 calls CLPROC3 or CLPROC4.

See Figure 4 on page 220 to understand the structure of the call stack when CLPROC2 calls CLPROC4. This figure illustrates the following considerations:

- There is a one-to-one correspondence between a call stack entry and an OPM program; for each call of an OPM program, one new entry is added to the call stack.
- An ILE program, as a unit, is not represented on the stack; instead, when an ILE program is called, one entry is added to the stack for each procedure that is called in the program. As a result, you send a message to an ILE procedure, not to an ILE program.

Note: The first procedure to run when an ILE program is called is the Program Entry Procedure (PEP) for the program. In CL, this procedure (_CL_PEP) is generated by the system and calls the first procedure you provide. In this example, the entry for the PEP is between the entry for the OPM program CLPGM1 and the entry for the procedure CLPROC1.

Following are different ways of specifying the base call stack entry.

Procedure Using the Command as the Base

If the TOPGMQ parameter specifies either TOPGMQ(*SAME *) or TOPGMQ(*PRV *), the entry for the procedure using the SNDPGMMSG command is used as the base. If TOPGMQ(*SAME *) is specified, the procedure will send a message to itself. If TOPGMQ(*PRV *) is specified, the procedure will send a message to the caller.

Note: You should be aware of the following information when a procedure sends a message to the caller by specifying TOPGMQ(*PRV *).

- When CLPROC4 and CLPROC2 send a message back to the callers, the message does not leave the containing program. The message is sent between procedures that are within the same program. If the objective is to send a message to the caller of the program (CLPGM1 in this example), specifying TOPGMQ(*PRV *) is not the right choice to use.
- When CLPROC1 sends its message back to the caller, the Program Entry Procedure is skipped. The message is sent to CLPGM1 even though the caller is the PEP. When TOPGMQ(*PRV *) is specified, the PEP entry is *not visible* and not included in the send operation. If TOPGMQ is specified in some other way, the PEP is *visible* to the sender.

Figure 5 on page 221 illustrates the results when CLPROC1, CLPROC2, and CLPROC4 each send a message back to the caller of each procedure.

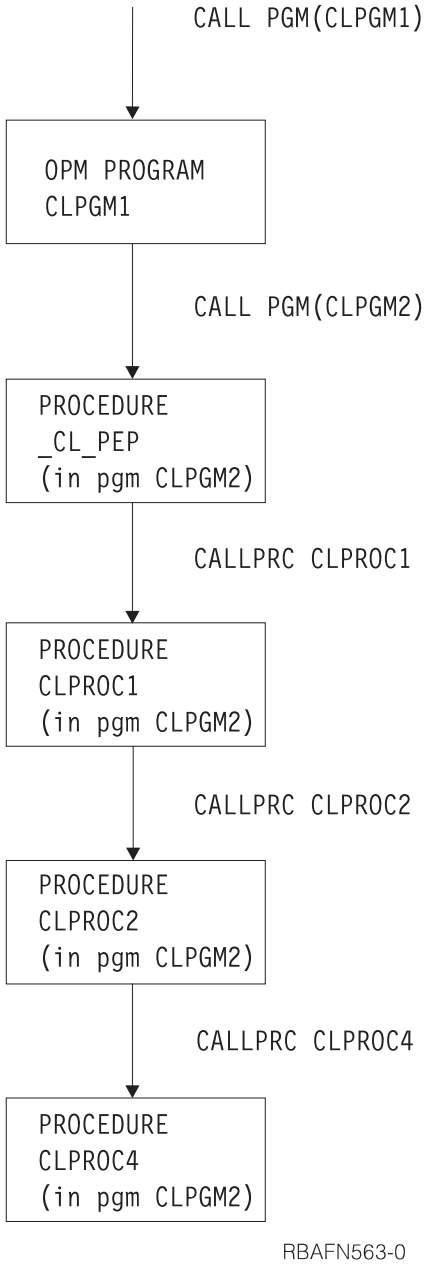


Figure 4. Example of runtime call stack

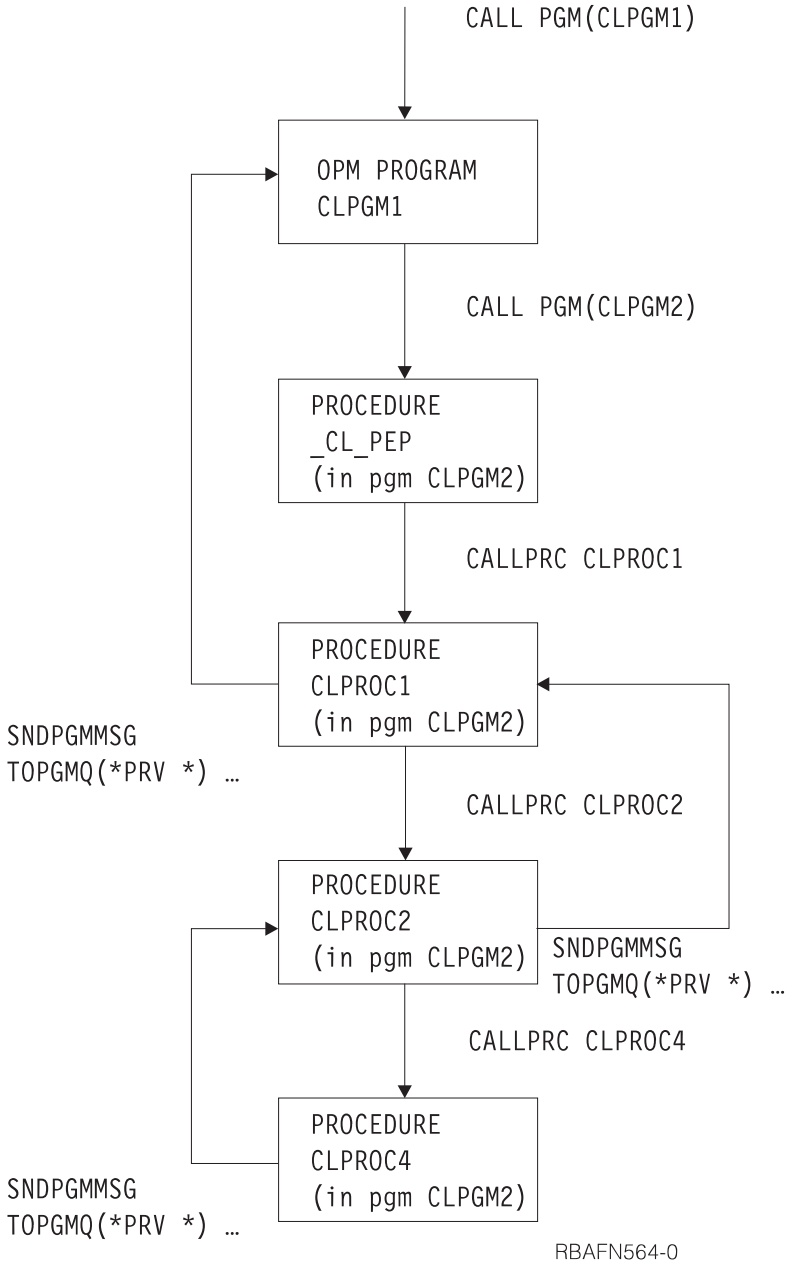


*Figure 5. Example of TOPGMQ(*PRV *)*

Identifying the Base Entry by Name

You can identify the base entry by providing the name of the OPM program or ILE procedure running in that entry. The name provided is either a simple name (one part) or a complex name (two or three parts). Following are descriptions of the simple and complex names:

- Simple name

  A simple name is used to identify an OPM program or an ILE procedure. If the simple name you provide is 10 characters or less in length, it is determined by the system that the name is either an OPM program or an ILE procedure. The base is identified as the most recently called OPM program or ILE procedure by that name.

If the name is longer than 10 characters in length, it is determined by the system that the name is for an ILE procedure (OPM program names cannot be longer than 10 characters). The base is identified as the entry for the most recently called procedure by that name. Entries running OPM programs are not considered.

See Figure 6 on page 223 for an example of sending a message using a simple name. In this example, CLPROC4 is sending a message to CLPROC2 and CLPROC2 is sending a message to CLPGM1.

- Complex name

  A complex name consists of two or three parts. They are:

  – module name

    The module name is the name of the module into which the procedure was compiled.

  – program name

    The program name is the name of the program into which the procedure was bound.

  – procedure name

  When you want to uniquely identify the procedure to which you want to send the message, a complex name can be used in one of the following combinations:

  – procedure name, module name, program name
  – procedure name and module name
  – procedure name and program name

    You must specify the module name as *NONE.

  If you use a complex name, the base being identified cannot be running an OPM program.

See Figure 7 on page 224 for an example of sending a message using a complex name. In this example, CLPROC4 is sending a message to CLPROC1 using a two part name consisting of (procedure name, program name).

Rather than using the full OPM program name or the full ILE procedure name, you may use partial names. IBM provides online information concerning how to specify partial call stack entry names. Refer to the *CL* section of the **Programming** category in the iSeries Information Center for this information.

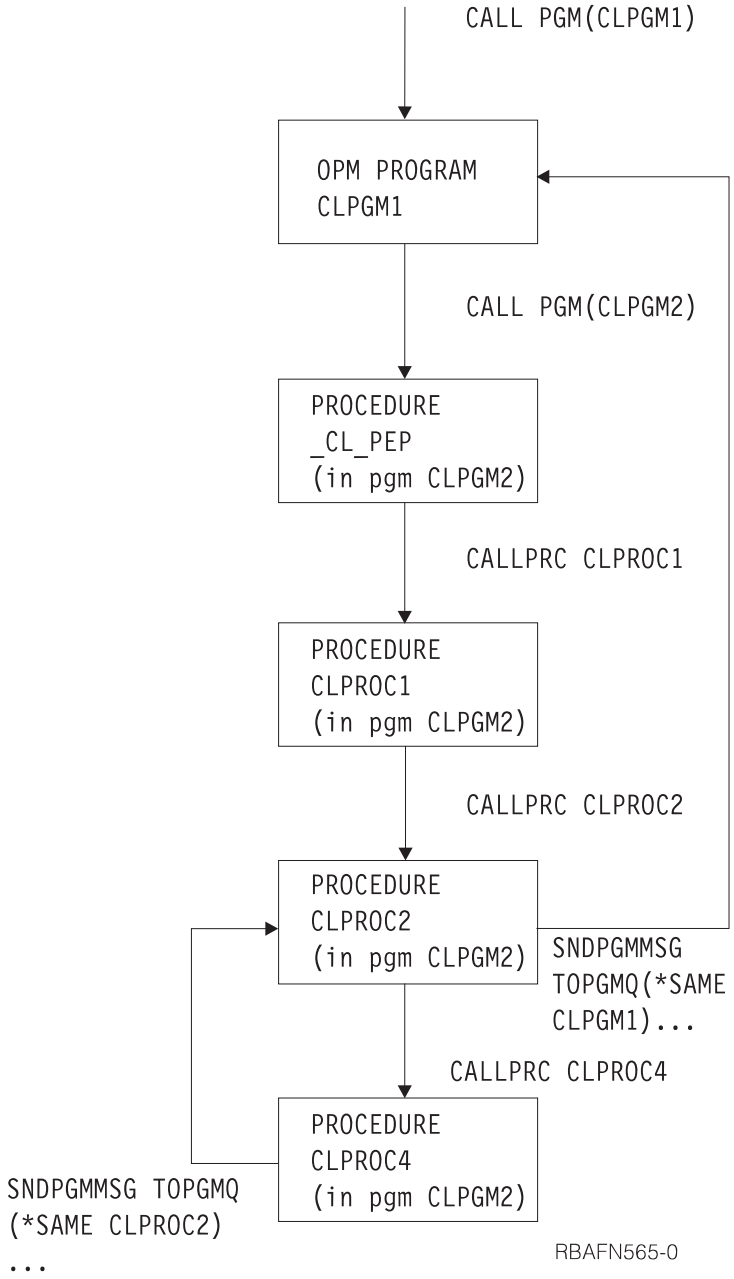


Figure 6. Example of using a simple name

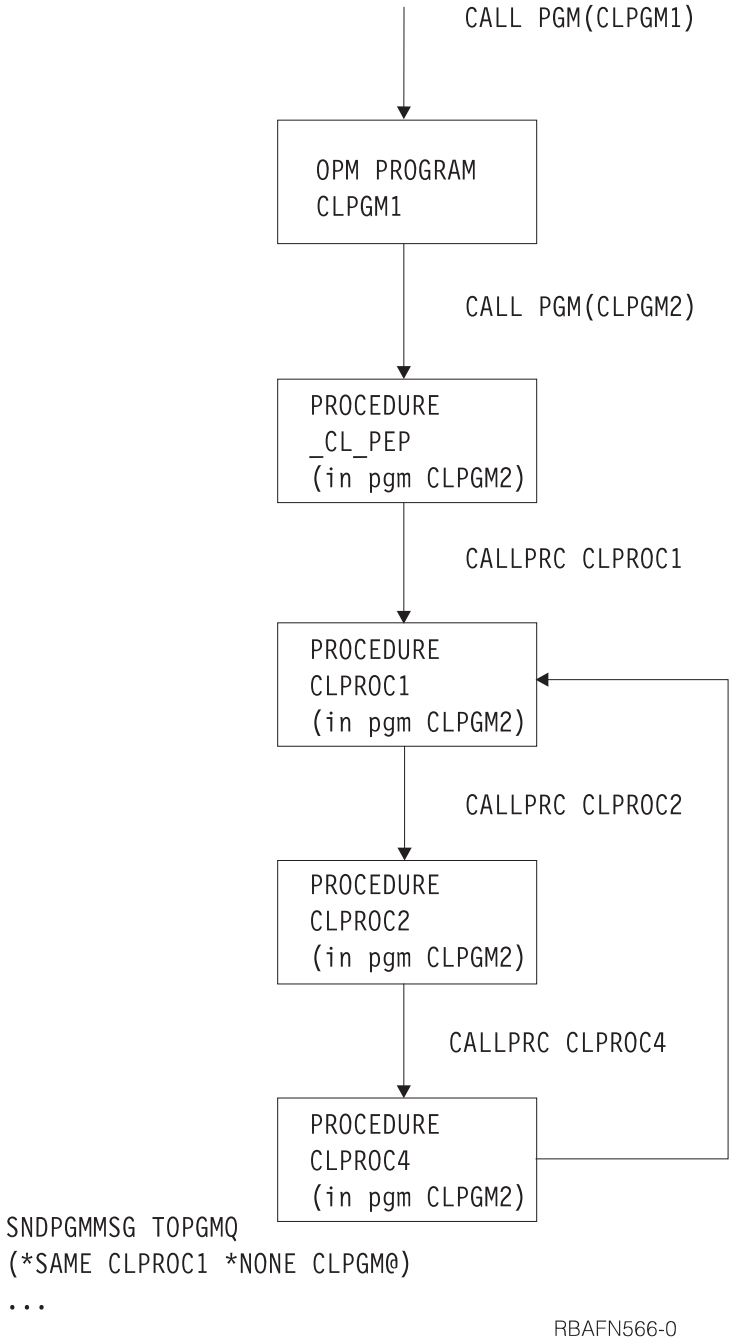


Figure 7. Example of using a complex name

Program Boundary as Base

The special value *PGMBDY is used by itself or with a program name to identify the PEP of a CL program. The entry for the PEP of the identified CL program then is the base entry. This option is useful when you want to send a message from within a CL procedure outside the boundary of the program which contains the procedure.

Refer to Figure 8 on page 226 for an example of sending a message using the special value *PGMBDY. In this example, CLPROC4 is sending a message directly to to CLPGM1 which is the caller of the containing program CLPGM2. CLPROC4

can do this without knowing which program called CLPGM2 or knowing the location of the PEP compared to the procedure sending the message. In this example, *PGMBDY is used without an accompanying program name specified. This means that the program whose boundary is to be identified is the program which contains the procedure that is sending the message.

See Figure 9 on page 227 for an example of sending a message using the special value *PGMBDY and a program name. The following programs and procedures are used in Figure 9 on page 227:

- CLPGM1 and CLPGM2. These are defined as in the previous examples.
- CLPGM3. This is another ILE program
- CLPROCA in CLPGM3. A message is sent from CLPROCA to the caller of CLPGM2.

A message is sent from CLPROCA to the caller of CLPGM2 by using the special value *PGMBDY with program name CLPGM2.

In this example, if the TOPGMQ parameter is specified as TOPGMQ(*PRV _CL_PEP), the message is sent to the caller of CLPGM3 rather than the caller of CLPGM2. This occurs because the most recently called procedure by that name is the PEP for CLPGM3.

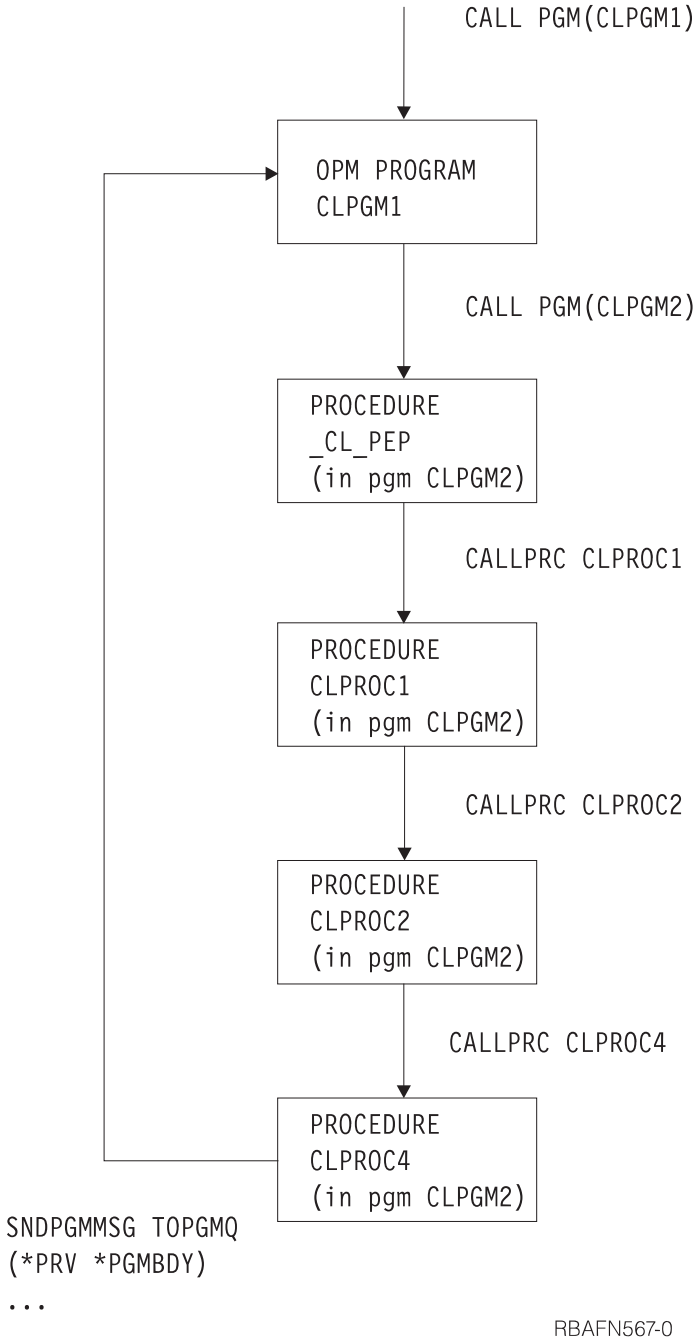


*Figure 8. Example 1 of using *PGMBDY*

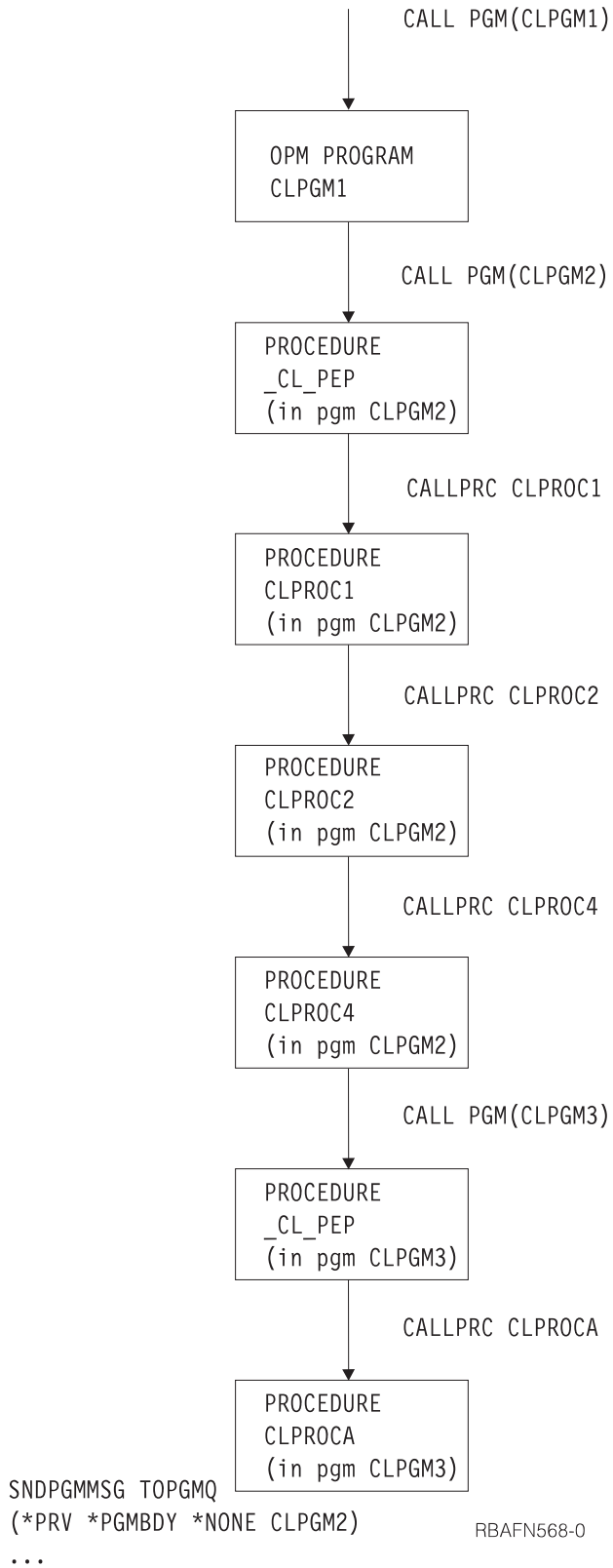


*Figure 9. Example 2 of using *PGMBDY*

The special value *PGMBDY can also be used with an OPM program. If you specify an OPM program name with *PGMBDY, you have the same results as

when only the OPM program name is used. For example, TOPGMQ(*SAME *PGMBDY *NONE opmname) sends the message to the same place as TOPGMQ(*SAME opmname).

The exception to this is when a message is sent to an OPM program that called itself recursively. TOPGMQ(*SAME pgmname) sends the message to the latest recursion level. However, TOPGMQ(*SAME *PGMBDY *NONE pgmname) sends the message to the first recursion level. Figure 10 shows how PGM1 is called and proceeds to call itself recursively two more times. At the third recursion level PGM1 calls PGM2. PGM2 then sends a message back to PGM1. If the program is sent using only the name PGM1, the message goes to the third recursion level of PGM1. If the program is sent using the name PGM1 in conjunction with the special value *PGMBDY, the message goes to the first recursion level of PGM1.

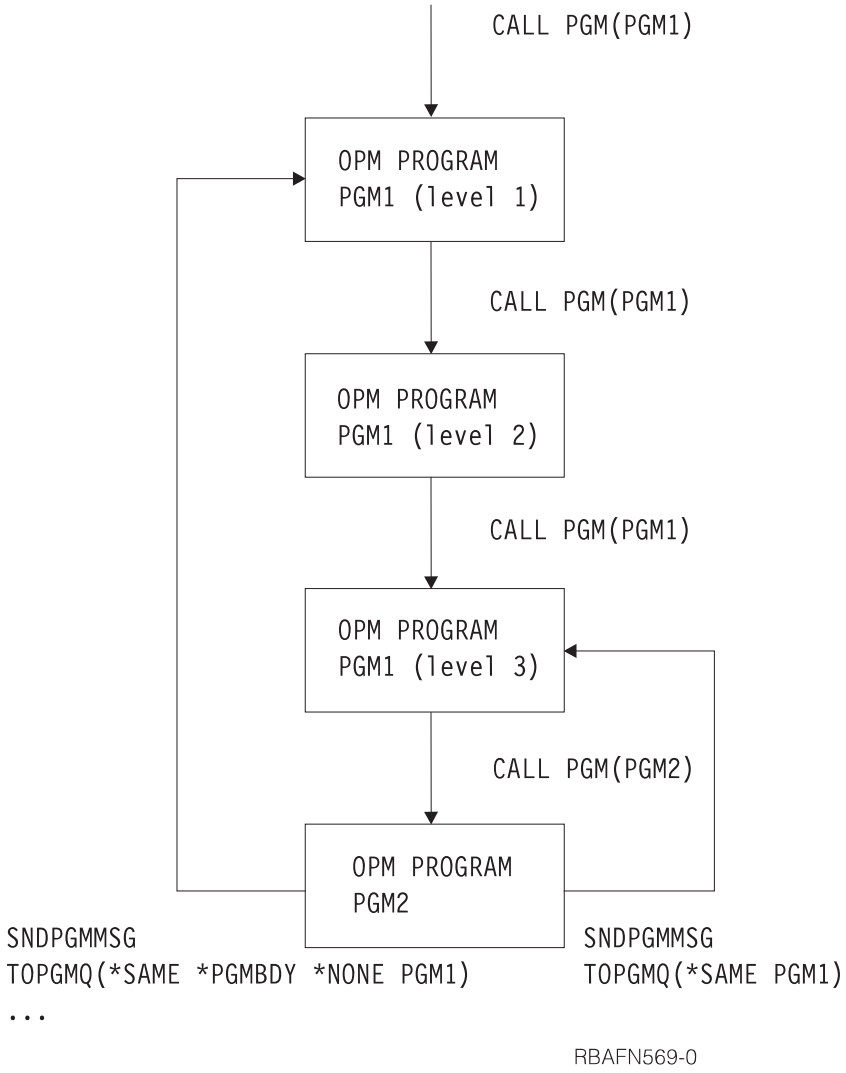


*Figure 10. Example 3 of using *PGMBDY*

Most Recently called Procedure as Base

Although you may not know the name of a procedure, you may want to send a message back to the most recently called procedure of an ILE program. The special

value *PGMNAME is used with a ILE program name to use the base entry name as the name for the most recently called procedure of the identified program. The programs in this example are:

- CLPGM1 is an ILE program with procedures PROCA and PROCB.
- CLPGM2 and CLPGM3 are both OPM programs.
- CLPGM3 is to send a message to CLPGM1 and does not know which procedure is the most recently called.

The send is accomplished using the special value *PGMNAME and the program name CLPGM1.

See Figure 11 on page 230. for an example of how to send a message using the special value *PGMNAME.

The special value *PGMNAME is useful if you convert some CL programs, but not all CL programs, to ILE programs. For example, CLPGM1 is an OPM CL program; CLPGM3 sent messages to CLPGM1 and specifies TOPGMQ(*SAME CLPGM1). If CLPGM1 is converted to ILE, only the SNDPGMMSG command in CLPGM3 (OPM) works. CLPGM1 does not work because there was no entry in the call stack for CLPGM1. If you change the command to TOPGMQ(*SAME *PGMNAME *NONE CLPGM1), CLPGM3 sends messages successfully to CLPGM1 regardless of the names you may have used for procedure names.

The special value *PGMNAME can also be used in with an OPM program name. In this case the effect is the same as if you just used the name. For example, TOPGMQ(*SAME *PGMNAME *NONE opmpgm) sends the message to the same place as TOPGMQ(*SAME opmpgm). The use of *PGMNAME should be considered when you cannot determine whether the message is being sent to an OPM program name or and ILE program name.

Using a Control Boundary as a Base

You can identify the base entry as the one at the nearest control boundary by using the special value *CTLBDY. A control boundary exists between two call stack entries if the two entries are running in two different activation groups. The one identified by using this special value is running in the same activation group as the entry that is sending the message.

See Figure 12 on page 231. for an example of sending a message using the special value *CTLBDY. The three programs in this example (CLPGM1, CLPGM2, and CLPGM3) are all ILE programs. CLPGM1 runs in activation group AG1 while both CLPGM2 and CLPGM3 run in activation group AG2. In this example, PROC3A sends a message back to the entry that immediately precedes the boundary for AG2.

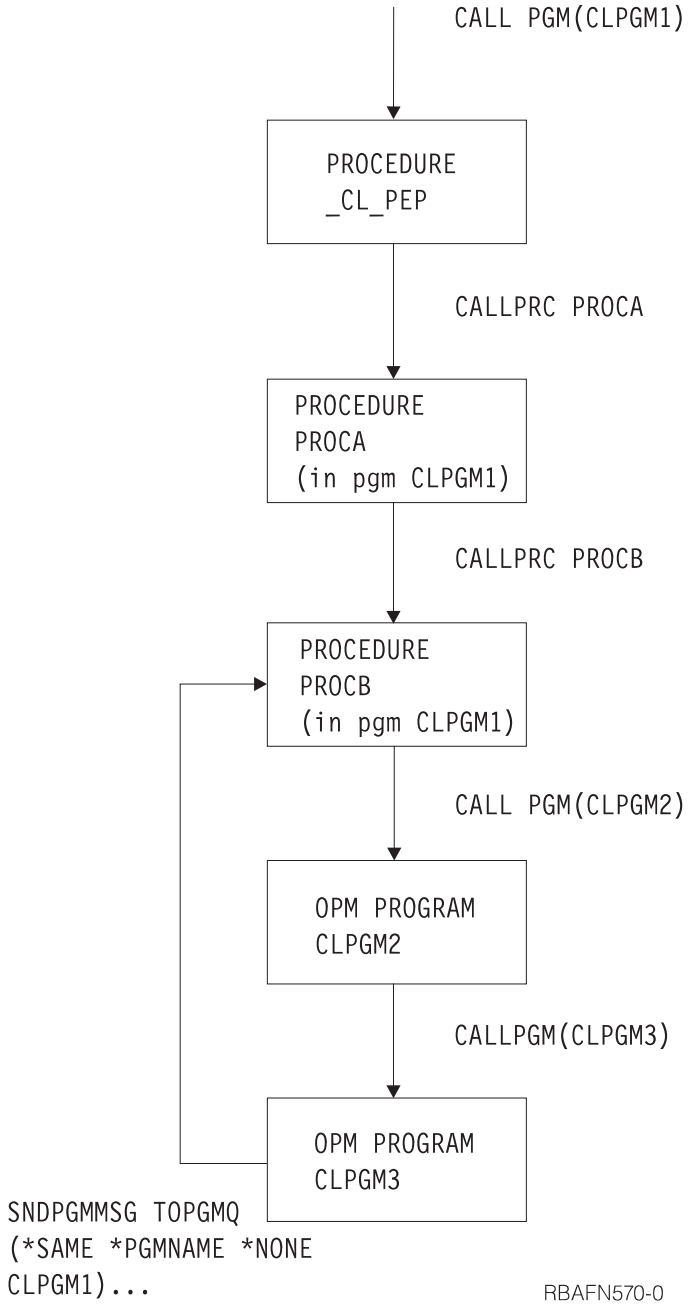


Figure 11. Example of runtime call stack

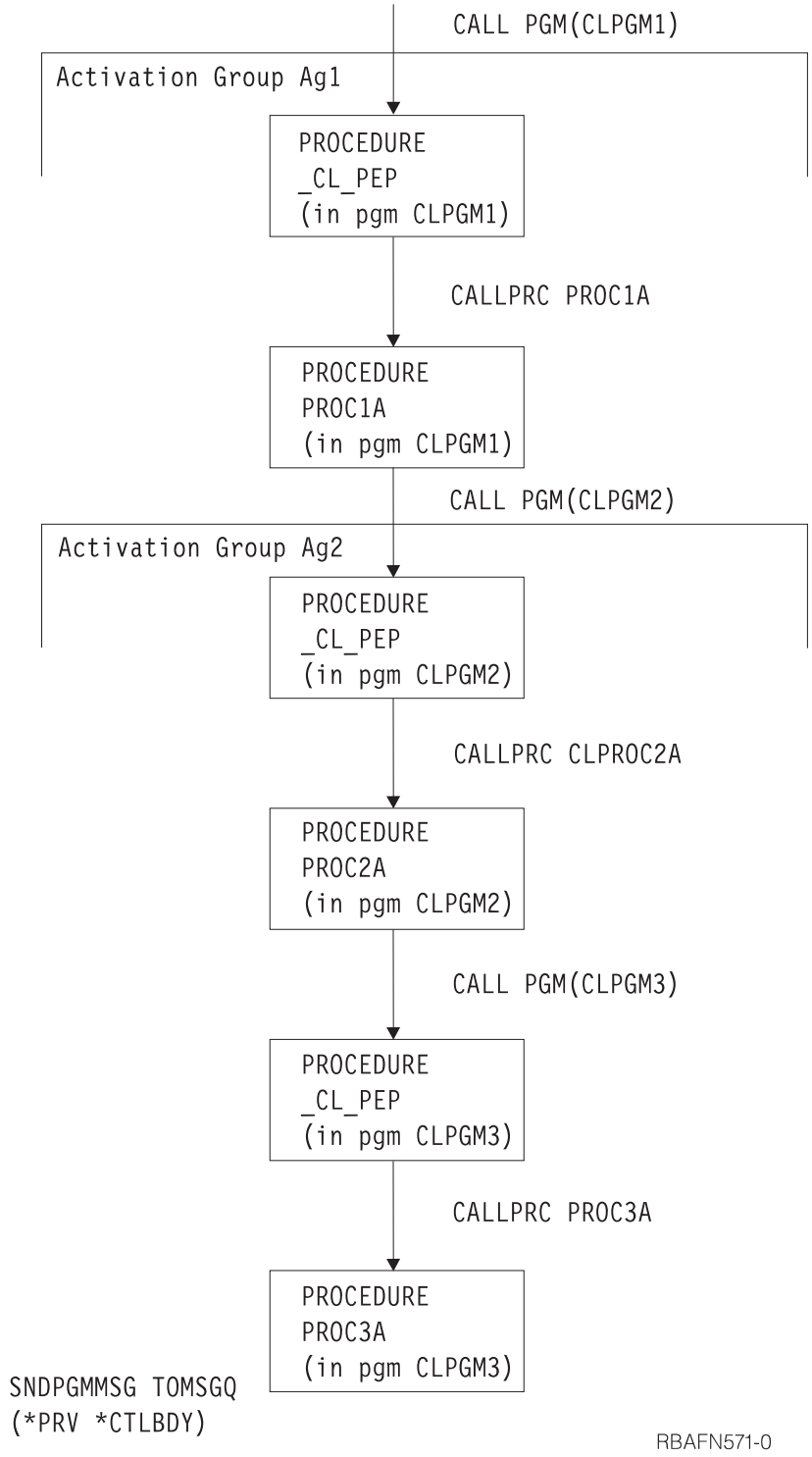


*Figure 12. Example of using *CTLBDY*

Considerations for Service Programs

Previous discussions apply to both ILE programs and ILE service programs. The most important difference between an ILE program and an ILE service program is related to message handling. The service program does not have a PEP.

The PEP is not necessary for any of the options used to identify a base entry. An exception to this is when the name _CL_PEP is used explicitly. For example, TOPGMQ(*PRV *PGMBDY) always sends a message to the caller of the ILE program or service program. If it is an ILE program, the PEP is identified as the base by the *PGMBDY value. If it is an ILE service program, the entry for the first procedure called in the service program is identified by the *PGMBDY value.

Receiving Messages in a CL Procedure or Program

Use the Receive Message (RCVMSG) command to receive messages from a message queue for your procedure or program. Messages can be received in the following ways:

- By message type. You can specify that all types or that a specific type can be received (MSGTYPE parameter). New messages (those that have not been received in the procedure or program) are received in a first-in-first-out (FIFO) order. However, ESCAPE type messages are received in last-in-first-out (LIFO) order.
  - By message reference key. You can do one of the following:
    - Receive a message using its message reference key. The system assigns a message reference key to each message on a message queue and passes the key as variable data because it is unprintable. You must declare this variable in your CL procedure or program (DCL command). You must specify on the RCVMSG command the CL variable through which the key is to be passed (MSGKEY parameter).
    - Receive the next message on a message queue following the message with a specified message reference key. In addition to specifying the MSGKEY parameter, you must specify MSGTYPE(*NEXT).
    - Receive the message on a message queue that is before a message with a specified message reference key. In addition to specifying the MSGKEY parameter, you must specify MSGTYPE(*PRV).
  - By its location on the message queue. You must specify MSGTYPE(*FIRST) for the first message on the message queue; specify MSGTYPE(*LAST) for the last.
  - By both message type and message reference key (MSGTYPE and MSGKEY parameters).

  To receive a message, you can specify:
  - Message queue. Where the message is to be received from.
  - Message type. Either a specific message type can be specified or all types can be specified.
  - Whether to wait for the arrival of a message. After the wait is over and no message is received, the CL variables requested to be returned are filled with blanks (or zeros if numeric) and control returns to the procedure or program running the RCVMSG command.
  - Whether to remove the message from the message queue after it is received. If it is not removed, it becomes an old message on the message queue and can only be received again (by a procedure) through its message reference key. However, if messages on the message queue are reset to new messages through the CHGMSGQ command, you do not have to use the message reference key to receive the message. Note that inquiry messages that have already been replied to are not reset to a new status. (See “Removing Messages from a Message Queue” on page 238 for more information.)

- CCSID to convert to. Specifies the CCSID that you want your message text returned in.
- A group of CL variables into which the following information is placed (each corresponds to one variable):
  - Message reference key of the message in the message queue (character variable, 4 characters)
  - Message (character variable, length varies)
  - Length of message, including length of substitution variable data (decimal variable, 5 decimal positions)
  - Message online help information (character variable, length varies)
  - Length of message help, including length of substitution variable data (decimal variable, 5 decimal positions)
  - Message data for the substitution variables provided by the sender of the message (character variable, length varies)
  - Length of the message data (decimal variable, 5 decimal positions)
  - Message identifier (character variable, 7 characters)
  - Severity code (decimal variable, length of 2)
  - Sender of the message (character variable, minimum of 80 characters)
  - Type of message received (character variable, 2 characters long)
  - Alert option of the message received (character variable, 9 characters)
  - Message file that contains the predefined message (character variable, 10 characters)
  - Message file library name that contains the message file used to receive the message (character variable, 10 characters)
  - Message file library name that contains the message file used to send the message (character variable, 10 characters)
  - Message data CCSID is the coded character set identifier associated with the replacement data returned (decimal variable, 5 decimal positions)
  - Text data CCSID is the coded character set identifier associated with the text returned by the Message and the Message help parameters (decimal variable, 5 decimal positions)

```
RCVMSG MSGQ(QGPL/INVN) MSGTYPE(*ANY) MSG(&MSG)
```

The message received is placed in the variable &MSG. *ANY is the default value on the MSGTYPE parameter.

When working with the call stack entry message queue of an ILE procedure written in a language other than CL, it is possible to receive an exception message (Escape or Notify) when the exception is not yet handled. The RCVMSG command can be used to both receive a message and indicate to the system that the exception has been handled.

This can be controlled by using the RMV keyword. If *NO is specified for this keyword, the exception is handled and the message is left on the message queue as an old message. If *KEEPEXCP is specified, the exception is not handled and the message is left on the message queue as a new message. If *YES is specified, the exception message is handled and the message is removed from the message queue.

The RTNTYPE keyword can be used to determine if the message received is an exception message, and if so, whether the exception has been handled.

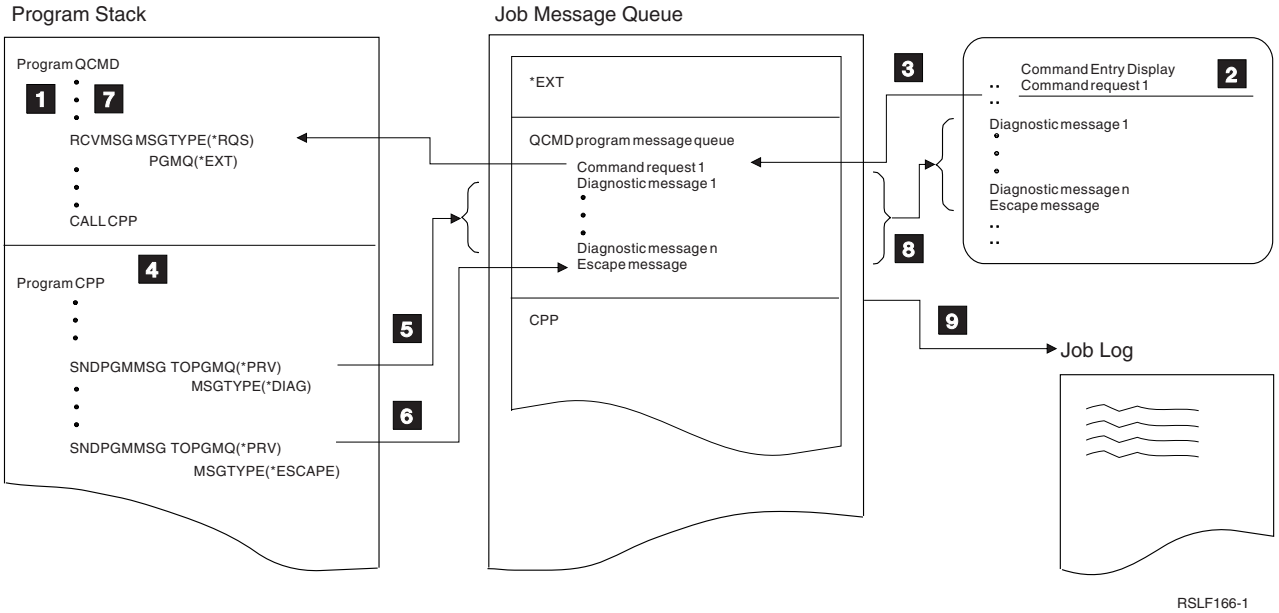
Request Messages

Receiving request messages is a method for your CL procedure or program to process CL commands. For example, your procedure or program can obtain input from a display station and handle the messages that result from the analysis and processing of the program. Usually, request messages are received from the external message queue (*EXT) of the job. For batch jobs, the requests received are those read from the input stream. For interactive jobs, the requests received are those the display station user enters one at a time on the Command Entry display. For example, CL commands are requests that are received by the IBM-supplied CL processor.

Your procedure or program must define the syntax of the data in the request message, interpret the request, and diagnose any errors. While the request is being analyzed or the request function is being run, any number of errors can be detected. As a result of these errors, messages are sent to the call message queue for the procedure or program. The procedure or program handles these messages and then receives the next request message. Thus, a request processing cycle is defined; a request message is received, the request is analyzed and run by your procedure or program with resulting messages displayed, and the next request received. If there are no more request messages to be received in a batch job, an escape message is sent to your procedure or program to indicate this.

More than one OPM program or ILE procedure of a job can receive request messages for processing. The requests received by more recent program calls are considered to be nested within those received by higher level program calls. The request processing cycles at each nesting level are independent of each other. Within an ILE program, one or more procedures within that program can be receiving request messages. If more than one procedure is processing requests than the nesting occurs within the same ILE program and the nesting levels remain independent.

The following diagram shows how request messages are processed by QCMD:



1 The CL processor QCMD receives a request message from *EXT.

2 If there is no request message on *EXT, the Command Entry display is

displayed. The display station user enters a command on the display. When the command is entered, it is placed on *EXT as a request message.

3 The command is then moved to the end of the QCMD call message queue and is passed from there to QCMD.

4 The command is analyzed and its command processing program (CPP) is called.

5 The command processing program sends diagnostic messages to the call message queue for QCMD.

6 Then the command processing program sends an escape message to the call message queue for QCMD. The escape message notifies QCMD that diagnostic messages are on the queue and that QCMD should end processing of the CPP.

7 QCMD is monitoring for the arrival of a request-check (CPF9901) or function-check (CPF9999) escape message. QCMD then tries to receive the next request message. If a request processor receives message CPF9901 or CPF9999, it should run a Reclaim Resources (RCLRSC) command. The request processor should also monitor for messages CPF1907 (end request) and CPF2415 (which indicates that the user pressed F3 or F12 on the Command Entry display).

8 Because a request message was being processed, all the messages on the call message queue for QCMD are written to the Command Entry display, which then prompts the display station user for another command.

9 The previous request message (command) and its associated messages are contained in the job log according to the message logging level specified for the job. For more information, see "Message Logging" on page 272.

Writing Request-Processing Procedures and Programs

Specifying a CL procedure as a request processor within a program has many advantages. The following list specifies three advantages:

- Processes request messages as described in "Request Messages" on page 234.
- Allows the use of the End Request (ENDRQS) command, which can be used from the System Request menu or as part of the disconnect job function.
- Allows filtering of messages to occur.

To become a request-processor procedure or program, your procedure or program must include the Send Program Message (SNDPGMMSG) and Receive Message (RCVMSG) commands. For example, the following commands would allow a procedure or program to become a request processor:

```
SNDPGMMSG  MSG('Request Message') TOPGMQ(*EXT) MSGTYPE(*RQS)
RCVMSG     PGMQ(*EXT) MSGTYPE(*RQS) RMV(*NO)
```

The request message is received from PGMQ *EXT. When any request message is received, it is moved (actually, it is removed and resent) to the call message queue of the procedure or program that specified the RCVMSG command. Therefore, the correct call message queue must be used when the message is removed.

If the request message is removed using the message reference key (MRK), you should obtain the MRK from the KEYVAR keyword of the RCVMSG command and not the SNDPGMMSG command. (The MRK changes when receiving a request message.) You must specify RMV(*NO) on the RCVMSG command because the procedure or program is not a request processor if the request message is removed from the call message queue.

The procedure or program is identified as a request processor when the request message is received. While the procedure or program is a request processor, other called procedures or programs can be ended using option 2 (End request) on the System Request menu. The request-processor procedure or program should include a monitor for message CPF1907 (MONMSG command). This is necessary because the end request function (from either option 2 on the System Request menu or the End Request command) sends this message to the request processor.

The procedure or program remains a request processor until the procedure ends (either normally or abnormally) or until a RMVMSG command is run to remove all the request messages from the request-processor's call message queue. For example, the following command removes all request messages from the message queue and, therefore, ends request processing:

```
RMVMSG  CLEAR(*ALL)
```

Call the QCAPCMD API and specify the message retrieve key to have the OS/400 command analyzer to process a request message for an OS/400 command. You can get the message retrieve key when you receive the request message. Process Commands (QCAPCMD) will update the request message in the job log and add any new value supplied. QCAPCMD also hides any parameter values, such as passwords, that are to hidden in the job log. The system will not update the request message in the job log when one of two conditions exists.

- Using the Execute Command (QCMDEXC or QCAEXEC) APIs.
- Failing to supply a message retrieve key to QCAPCMD.

Determining if a Request-Processor Exists

To determine if a job has a request processor, display the job's call stack. Use either option 11 on the Display Job (DSPJOB) or Work with Job (WRKJOB) command, or select option 10 for the job listed on the WRKACTJOB display. If a number is shown in the request level column on the display of the job's call stack, the program or ILE procedure associated with the number is a request-processor. In the following example, both QCMD and QTEVIREF are request processors:

```
                          Display Call Stack
                                                          System:   S0000000
Job:   WS31           User:   QSECOFR        Number:   000173

Type options, press Enter.
  5=Display details

     Request   Program or
Opt   Level    Procedure      Library        Statement      Instruction
               QCMD           QSYS                             01DC
       1       QCMD           QSYS                             016B
               QTECADTR       QSYS                             0001
       2       QTEVIREF       QSYS                             02BA

                                                                      Bottom

F3=Exit   F10=Update stack   F11=Display activation group   F12=Cancel
F17=Top   F18=Bottom
```

The following is an example of a request-processing procedure:

```
PGM
    SNDPGMMSG  MSG('Request Message') TOPGMQ(*EXT) MSGTYPE(*RQS)
    RCVMSG     PGMQ(*EXT) MSGTYPE(*RQS) RMV(*NO)
      .
      .
      .
    CALL   PGM(PGMONE)
    MONMSG MSGID(CPF1907)
      .
      .
      .
    RMVMSG CLEAR(*ALL)
    CALL   PGM(PGMTWO)
      .
      .
      .
ENDPGM
```

The first two commands in the procedure make it a request processor. The procedure remains a request processor until the RMVMSG command is run. A Monitor Message command is placed after the call to program PGMONE because an end request may be sent from PGMONE to the request-processor. If monitoring is not used, a function check would occur for an end request. No message monitor is specified after the call to PGMTWO because the RMVMSG command ends request processing.

If an end request is attempted when no request-processing procedure or program is called, an error message is issued and the end operation is not performed.

Note: In the sample programs, the RCVMSG command uses the minimal number of parameters needed to become a request processor. You need to say you want to receive a request message but do not want to remove it. You also need to identify the specific call queue from which the message request originated. Other parameters can be added as necessary.

Retrieving Messages in a CL Procedure

You can use the Retrieve Message (RTVMSG) command to retrieve the text of a message from a message file into a variable. RTVMSG operates on predefined message descriptions. You can specify the message identifier and message file name in addition to the following:

- CCSID to convert to. Specifies the coded character set identifier that you want your message text and data returned in.
- Message data fields. The message data for the substitution variables.
- Message data CCSID. Specifies the coded character set identifier that the supplied message data is to be considered in.
- A group of CL variables into which the following information is placed (each corresponds to one variable):
  - Message (character variable, length varies)
  - Length of message, including length of substitution variable data (decimal variable, 5 decimal positions)
  - Message online help information (character variable, length varies)
  - Length of message help, including length of substitution variable data (decimal variable, 5 decimal positions)
  - Severity code (decimal variable, 2 decimal positions)
  - Alert option (character variable, 9 characters)

- Log problem in the service activity log (character variable, 1 character)
- Message data CCSID is the coded character set identifier associated with the replacement data returned (decimal variable, 5 decimal positions)
- Text data CCSID is the coded character set identifier associated with the text returned by the Message and the Message help parameters (decimal variable, 5 decimal positions)

For example, the following command adds the message description for the message USR1001 to the message file USRMSG:

```
ADDMSGD   MSGID(USR1001) MSGF(QGPL/USRMSG) +
          MSG('File &1 not found in library &2') +
          SECLVL('Change file name or library name') +
          SEV(40) FMT((*CHAR 10) (*CHAR 10))
```

The following commands result in the substitution of the file name INVENT in the 10-character variable &FILE and the library name QGPL in the 10-character variable &LIB in the retrieved message USR1001.

```
DCL &FILE TYPE(*CHAR) LEN(10) VALUE(INVENT)
DCL &LIB TYPE(*CHAR) LEN(10) VALUE(QGPL)
DCL &A TYPE(*CHAR) LEN(20)
DCL &MSG TYPE(*CHAR) LEN(50)
CHGVAR VAR(&A) VALUE(&FILE||&LIB)
RTVMSG   MSGID(USR1001) MSGF(QGPL/USRMSG) +
         MSGDTA(&A) MSG(&MSG)
```

The data for &1 and &2; is contained in the procedure variable &A, in which the values of the procedure variables &FILE and &LIB have been concatenated. The following message is placed in the CL variable &MSG:

```
File INVENT not found in library QGPL
```

If the MSGDTA parameter is not used in the RTVMSG command, the following message is placed in the CL variable &MSG:

```
File not found in library
```

After the message is placed in the variable &MSG, you could do the following:

- Send the message using the SNDPGMMSG command
- Use the variable as the text for a message line in DDS (M in position 38)
- Use a message subfile
- Print or display the message

Note: You cannot retrieve the message text with the variable names that are included in the text. The system intends on RTVMSGD to return a sendable message.

Removing Messages from a Message Queue

Messages are held on a message queue until they are removed by a Remove Message (RMVMSG) command, Clear Message Queue (CLRMSGQ) command, the RMV parameter on the Receive Message (RCVMSG) and Send Reply (SNDRPY) commands, the remove function keys of the Display Messages display, or the clear message queue option on the Work with Message Queue display. You can remove:

- A single message
- All messages
- All except unanswered messages
- All old messages

- All new messages
- All messages from all inactive programs

To remove a single message using the RMVMSG command or a single old message using the RCVMSG command, you specify the message reference key of the message to be removed.

Note: The message reference key can also be used to receive a message and to reply to a message.

If you remove an inquiry message that you have not answered, a default reply is sent to the sender of the message and the inquiry message and the default reply are removed. If you remove an inquiry message that you have already answered, both the message and your reply are removed.

To remove all messages for all inactive programs and procedures from a user's job message queue, specify *ALLINACT for the PGMQ parameter and *ALL for the CLEAR parameter on the RMVMSG command. If you want to print your job log before you remove all the inactive messages, use the Display Job Log (DSPJOBLOG) command and specify *PRINT for the OUTPUT parameter.

When working with a call message queue of an ILE procedure, it is possible that an exception message for unhandled exceptions is on the queue at the time the RMVMSG command is run. The RMVEXCP keyword of this command can be used to control actions for messages of this type. If *YES is specified for this keyword, the RMVMSG command causes the exception to be handled and the message to be removed. If *NO is specified, the message is not removed. As a result, the exception is not handled.

The following RMVMSG command removes a message from the user message queue JONES. The message reference key is in the CL variable &MRKEY.

```
DCL &MRKEY TYPE(*CHAR) LEN(4)
RCVMSG MSGQ(JONES) RMV(*NO) KEYVAR(&MRKEY)
RMVMSG MSGQ(JONES) MSGKEY(&MRKEY)
```

The following RMVMSG command removes all messages from a message queue.

```
RMVMSG CLEAR(*ALL)
```

Note: The maximum number of messages on a user message queue or a workstation message queue is 65 535 for each type of message sent. For example, 65 535 diagnostic messages can be on the queue; 65 535 completion messages can be on the queue, and so on. For a call message queue or the *EXT queue, there is no restriction on the maximum number of messages per type.

Monitoring for Messages in a CL Program or Procedure

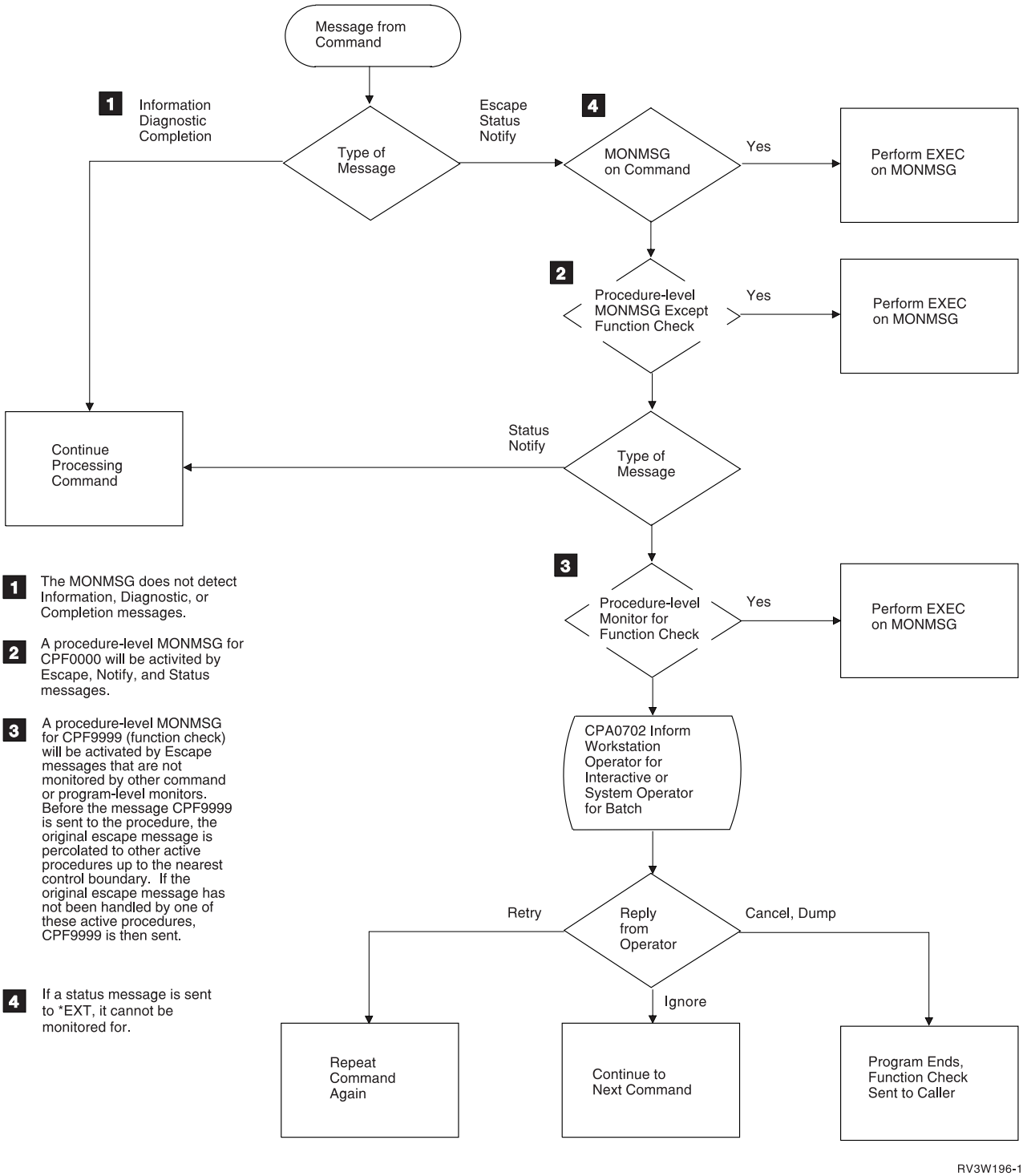
You can monitor for escape, notify, and status messages that are sent to your CL procedure's or program's call message queue by the commands in your procedure or program or by commands in another procedure or program. The Monitor Message (MONMSG) command monitors the messages sent to the call message queue for the conditions specified in the command. If the condition exists, the CL command specified on the MONMSG command is run. The logic involved with the MONMSG command is as follows:

Escape Messages: Escape messages are sent to tell your procedure or program of an error condition that forced the sender to end. By monitoring for escape messages, you can take corrective actions or clean up and end your procedure or program.

Status or Notify Messages Status and notify messages are sent to tell your procedure or program of an abnormal condition that is not serious enough for the sender to end. By monitoring for status or notify messages, your procedure or program can detect this condition and not allow the function to continue.

You can monitor for messages using two levels of MONMSG commands:

- Procedure level: You can monitor for an escape, notify, or status message sent by any command in your procedure by specifying the MONMSG command immediately following the last declare command in your CL procedure or program. This is called a procedure-level MONMSG command. You can use as many as 100 procedure-level MONMSG commands in a procedure or OPM program. (A CL procedure or OPM program can contain a total of 1000 MONMSG commands.) This lets you handle the same escape message in the same way for all commands. The EXEC parameter is optional, and only the GOTO command can be specified on this EXEC parameter.



- Specific command level: You can monitor for an escape, notify, or status message sent by a specific command in your procedure or program by specifying the MONMSG command immediately following the command. This is called a command level MONMSG command. You can use as many as 100 command-level MONMSG commands for a single command. This lets you handle different escape messages in different ways.

To monitor for escape, status, or notify messages, you must specify, on the MONMSG command, generic message identifiers for the messages in one of the following ways:

- `pppmmnn`

  Monitors for a specific message. For example, MCH1211 is the message identifier of the zero divide escape message.

- `pppmm00`

  Monitors for any message with a generic message identifier that begins with a specific licensed program (`ppp`) and the digits specified by mm. For example, CPF5100 indicates that all notify, status, and escape messages beginning with CPF51 are monitored.

- `ppp0000`

  Monitors for every message with a generic message identifier that begins with a specific licensed program (`ppp`). For example, `CPF0000` indicates that all notify, status, and escape messages beginning with `CPF` are monitored.

  Note: Do not use `MONMSG CPF0000` when doing system function, such as install or saving or restoring your entire system, since you may lose important information.

- CPF9999

  Monitors for function check messages for all generic message identifiers. If an error message is not monitored, it becomes a CPF9999 (function check).

Note: Generally, when monitoring, your monitor also gets control when notify and status messages are sent.

In addition to monitoring for escape messages by message identifier, you can compare a character string, which you specify on the MONMSG command, to data sent in the message. For example, the following command monitors for an escape message (CPF5101) for the file MYFILE. The name of the file is sent as message data.

```
MONMSG MSGID(CPF5101) CMPDTA(MYFILE) EXEC(GOTO EOJ)
```

The compare data can be as long as 28 characters, and the comparison starts with the first character of the first field of the message data. If the compare data matches the message data, the action specified on the EXEC parameter is run.

The EXEC parameter on the MONMSG command specifies how an escape message is to be handled. Any command except PGM, ENDPGM, IF, ELSE, DCL, DCLF, ENDDO, and MONMSG can be specified on the EXEC parameter. You can specify a DO command on the EXEC parameter, in which case, the commands in the do group are run. When the command or do group (on the EXEC parameter) has been run, control returns to the command in your procedure or program that is after the command that sent the escape message. However, if you specify a GOTO or RETURN command, control does not return. If you do not specify the EXEC parameter, the escape message is ignored and your procedure continues.

The following shows an example of a Change Variable (CHGVAR) command being monitored for a zero divide escape message, message identifier MCH1211:

```
CHGVAR VAR(&A) VALUE(&A / &B)
MONMSG MSGID(MCH1211) EXEC(CHGVAR VAR(&A) VALUE(1))
```

The value of the variable &A is changed to the value of &A divided by &B. If &B equals 0, the divide operation cannot be done and the zero divide escape message is sent to the procedure. When this happens, the value of &A is changed to 1 (as specified on the EXEC parameter). You may also test &B for zero, and only perform the division if it is not zero. This is more efficient than attempting the operation and monitoring for the escape message.

In the following example, the procedure monitors for the escape message CPF9801 (object not found message) on the Check Object (CHKOBJ) command:

```
          PGM
          CHKOBJ LIB1/PGMA *PGM
          MONMSG MSGID(CPF9801) EXEC(GOTO NOTFOUND)
          CALL LIB1/PGMA
          RETURN
NOTFOUND: CALL FIX001 /* PGMA Not Found Routine */
          ENDPGM
```

The following CL procedure contains two CALL commands and a procedure-level MONMSG command for CPF0001. (This escape message occurs if a CALL command cannot be completed successfully.) If either CALL command fails, the procedure sends the completion message and ends.

```
        PGM
        MONMSG MSGID(CPF0001) EXEC(GOTO ERROR)
        CALL PROGA
        CALL PROGB
        RETURN
ERROR:  SNDPGMMSG MSG('A CALL command failed') MSGTYPE(*COMP)
        ENDPGM
```

If the EXEC parameter is not coded on a procedure-level MONMSG command, any escape message that is handled by the MONMSG command is ignored. If the escape message occurs on any command except the condition of an IF command, the procedure or program continues processing with the command that would have been run next if the escape message had not occurred. If the escape message occurs on the condition of an IF command, the procedure or program continues processing as if the condition on the IF command were false. The following example illustrates what happens if an escape message occurs at different points in the procedure:

```
PGM
DCL &A TYPE(*DEC) LEN(5 0)
DCL &B TYPE(*DEC) LEN(5 0)
MONMSG MSGID(CPF0001 MCH1211)
CALL PGMA PARM(&A &B)
IF (&A/&B *EQ 5) THEN(CALL PGMB)
ELSE CALL PGMC
CALL PGMD
ENDPGM
```

Depending on where an escape message occurs, the following happens:

- If CPF0001 occurs on the call to PGMA, the procedure resumes processing on the IF command.
- If MCH1211 (divide by 0) occurs on the IF command, the IF condition is considered false, and the procedure resumes processing with the call to PGMC.
- If CPF0001 occurs on the call to PGMB or PGMC, the procedure resumes processing with the call to PGMD.
- If CPF0001 occurs on the call to PGMD, the procedure resumes processing with the ENDPGM command, which causes a return to the calling procedure.

You can also monitor for the same escape message to be sent by a specific command in your procedure or program *and* by another command. This requires two MONMSG commands. One MONMSG command follows the command that needs special handling for the escape message; for that command, this MONMSG command is used when the escape message is sent. The other MONMSG command follows the last declare command so that for all other commands, this MONMSG command is used.

MONMSG commands apply only to the CL procedure or OPM program in which they are coded. MONMSG commands from one procedure do not apply to another procedure even though both are part of the same program. IBM provides online help that contains a list of the escape, notify, and status messages that are issued for CL commands. Refer to the *CL* section of the **Programming** category in the iSeries Information Center for this information. You should also keep a list of all messages that you have defined.

Note: The above paragraph is not true for ILE procedures because of the way messages percolate. The system requires MONMSG to handle any escape message that is sent to a procedure. Otherwise, the message percolates up the call stack until it finds a procedure that has a MONMSG to handle it or hits a control boundary.

Default Handling

Many escape messages can be sent to a procedure that calls commands, programs, and procedures. You will not want to monitor and handle all of the messages. However, you may want to monitor and handle the escape messages which pertain to the function of your procedure. The system provides default monitoring and handling of any messages you do not monitor.

Default handling assumes that an error has been detected in a procedure. If you are debugging the procedure, the message is sent to your display station. You can then enter commands to analyze and correct the error. If you are not debugging the procedure, the system performs a message percolation function.

Message percolation is a two-step function that does the following:

- Moves the escape message one step earlier in the call stack.
- Checks to see if the procedure has a MONMSG command for the escape.

If the procedure has a MONMSG command for the escape, the message percolation action stops, and the system takes the action that is specified by the MONMSG command. Message percolation continues until either finding a MONMSG command, or until finding the nearest control boundary. This means that the escape message does not percolate across control boundaries.

The function check processing begins by finding the control boundary before finding a procedure with a MONMSG command which applies to the message. The system considers action on the original escape exception complete. The system then sends the function check message (CPF9999) to the procedure that was the target of the original escape. If that procedure has a MONMSG for the function check message, then it takes the action that is specified by that command. Otherwise, the system sends an inquiry message to the workstation operator if the job is an interactive job. The workstation operation can reply with one of the following replies:

R Retry the failing command in the procedure.

I Ignore the message. Continue processing at the next command in the procedure.

C Cancel the procedure and percolate the function check to the next previous procedure on the call stack.

D Dump the call stack entry for the failing procedure, cancel the procedure, and percolate the function check to the next previous procedure on the call stack. This is the default action if entering no reply, or if the job is a batch job.

The system does not percolate the function check across the control boundary. If any reply causes the function check to move across an activation group boundary, this stops further action on the function check. The system cancels all procedures up to the activation group boundary, and sends the escape message CEE9901 to the prior call stack entry.

You can monitor for function-check escape messages so that you can either:

- Clean up and end the procedure
- Continue with some other aspect of your procedure

Note: If the message description for the unmonitored escape specifies a default action, the default handling program is called before the function check message is sent. When the default handling program returns, function check processing begins.

Notify Messages

Besides monitoring for escape messages, you can monitor for notify messages that are sent to your CL procedure's or program's call message queue by the commands in your procedure or program or by the programs and procedures it calls. Notify messages are sent to tell your procedure or program of a condition that is not typically an error. By monitoring for notify messages, you can specify an action different from what you would specify if the condition had not been detected. Very few IBM-supplied commands send notify messages.

Monitoring for and handling notify messages is similar to monitoring for and handling escape messages. The difference is in what happens if you do not monitor for and handle notify messages. Notify messages are also percolated from procedure to procedure within the boundary of the activation group. If the activation group boundary is reached without a MONMSG command being found for it, the default reply is automatically returned to the sender of the notify message and the sender is allowed to continue processing. Unlike escape messages, unmonitored notify messages are not considered an indication of an error in your procedure or program.

Status Messages

You can monitor for status messages that are sent by the commands in your CL procedure or by the programs or procedures it calls. Status messages tell your procedure the status of the work performed by the sender. By monitoring for status messages, you can prevent the sending program or procedure from proceeding with any more processing.

No message information is stored in a message queue for status messages. Therefore, a status message cannot be received.

If a status message is not monitored for, it is percolated like escape and notify messages are. If the activation group boundary is reached without a MONMSG command being found, action on the message is considered complete and control is returned to the sender of the message to continue processing. Status messages are often sent to communicate normal conditions that have been detected where processing can continue.

Status messages sent to the external message queue are shown on the interactive display, informing the user of a function in progress. For example, the Copy File (CPYF) command sends a message informing the user that a copy operation is in progress.

Only predefined messages can be sent as status messages; immediate messages cannot be sent. You can use the system-supplied message ID, CPF9898, and supply message data to send a status message if you do not have an existing message description.

When the function is completed, your procedure or program should remove the status message from the interactive display. The message cannot be removed using a command, but sending another status message to *EXT with a blank message gives the appearance of removing the message. The system-supplied message ID CPI9801 can be used for this purpose. When control returns to the OS/400 program, the *STATUS message may be cleared from line 24, without sending the CPI9801 message. The following example shows a typical application of message IDs CPF9898 and CPI9801:

```
SNDPGMMSG MSGID(CPF9898) MSGF(QCPFMSG) +
          MSGDTA('Function xxx being performed') +
          TOPGMQ(*EXT) MSGTYPE(*STATUS)
    •
    •   /*  Your processing function  */
    •
SNDPGMMSG MSGID(CPI9801) MSGF(QCPFMSG) +
          TOPGMQ(*EXT) MSGTYPE(*STATUS)
```

Preventing the Display of Status Messages

You cannot prevent commands from sending status messages, but you can prevent the status messages from being displayed at the bottom of the screen.

There are two preferred ways to prevent the status messages from being shown:

- Change User Profile (CHGUSRPRF) command

  You can change your user profile so that whenever you sign on using that profile, status messages are not shown. To do this, use the CHGUSRPRF command and specify *NOSTSMSG on the User Option (USROPT) parameter.
- Change Job (CHGJOB) command

  You can change the job you are currently running so that status messages are not shown. To do this, use the CHGJOB command and specify *NONE on the Status Message (STSMSG) parameter. You can also use the CHGJOB command to see status messages by specifying *NORMAL on the STSMSG parameter.

A third alternative, however less preferred, is to use the Override Message File (OVRMSGF) command and change the status message identifiers to a blank message.

Break-Handling Programs

A break-handling program is one that is automatically called when a message arrives at a message queue that is in *BREAK mode. You must specify the name of both the program and the break delivery name on the same Change Message Queue (CHGMSGQ) command. Although you specify the program on the CHGMSGQ command, it is one or more procedures within the program that processes the message. A procedure within this program must run a Receive Message (RCVMSG) command to receive the message. To receive and handle the message, the user-defined program called to handle messages for break delivery receives parameters. Specifically, the first procedure to run within the program receives these parameters. The parameters identify the message queue and the message reference key (MRK) of the message that is causing the break. The online help information for the Receive Message (RCVMSG) command describes parameters of the Break Handling exit program. Refer to the command documentation in the *CL* section of the **Programming** category in the **iSeries Information Center**. If the system calls a break-handling program, it interrupts the job that has the message queue in break mode. When the break-handling program ends, the original program resumes processing.

The following program (PGMA), which consists of only this one procedure, is an example of a break-handling program.

```
PGM PARM(&MSGQ &MSGLIB &MRK)
DCL VAR(&MSGQ) TYPE(*CHAR) LEN(10)
DCL VAR(&MSGLIB) TYPE(*CHAR) LEN(10)
DCL VAR(&MRK) TYPE(*CHAR) LEN(4)
DCL VAR(&MSG) TYPE(*CHAR) LEN(75)
RCVMSG  MSGQ(&MSGLIB/&MSGQ) MSGKEY(&MRK) +
        MSG(&MSG)
.
.
.
ENDPGM
```

After the break-handling program is created, running the following command connects it to the QSYSMSG message queue.

```
CHGMSGQ   MSGQ(QSYS/QSYSMSG) DLVRY(*BREAK) PGM(PGMA)
```

Notes:

1. When messages are handled, they should be removed from the message queue. When a message queue is put in break mode, any message on the queue will cause the break-handling program to get called.
2. The procedure or program receiving the message should not be coded with a wait-time other than zero to receive a message. You can specify a value other than zero for the wait parameter with the Receive Message (RCVMSG) command. The message arrival event cannot be handled by the system while the job is running a break-handling event.

An example of a break-handling program is to have the program send a message, which is normally sent to the QSYSOPR queue, to another queue in place of or in addition to QSYSOPR.

The following is an example of a user-defined program (again with only one procedure) to handle break messages. The display station user does not need to respond to the messages CPA5243 (Press Ready, Start, or Start-Stop on device &1) and CPA5316 (Verify alignment on device &3) when this program is used.

```
BRKPGM:     PGM (&MSGQ &MSGQLIB &MSGMRK)
            DCL &MSGQ TYPE(*CHAR) LEN(10)
            DCL &MSGQLIB TYPE(*CHAR) LEN(10)
            DCL &MSGMRK TYPE(*CHAR) LEN(4)
            DCL &MSGID TYPE(*CHAR) LEN(7)
            RCVMSG MSGQ(&MSGQLIB/&MSGQ) MSGKEY(&MSGMRK) +
                   MSGID(&MSGID) RMV(*NO)
            /* Ignore message CPA5243 */
            IF (&MSGID *EQ 'CPA5243') GOTO ENDBRKPGM
            /* Reply to forms alignment message */
            IF (&MSGID *EQ 'CPA5316') +
                   DO
                   SNDRPY MSGKEY(&MSGMRK) MSGQ(&MSGQLIB/&MSGQ) RPY(I)
                   ENDDO
            /* Other messages require user intervention */
            ELSE CMD(DSPMSG MSGQ(&MSGQLIB/&MSGQ))
ENDBRKPGM:  ENDPGM
```

Attention:

In the above example of a break-handling program, if a CPA5316 message should arrive at the queue while the DSPMSG command is running, the DSPMSG display shows the original message that caused the break and the CPA5316 message. The DSPMSG display waits for the operator to reply to the CPA5316 message before proceeding.

Note: This program cannot open a display file if the interrupted program is waiting for input data from the display.

You can use the system reply list to indicate the system will issue a reply to predefined inquiry messages. The display station user, therefore, does not need to reply. For more information, see "Using the System Reply List" on page 269.

A procedure within a user break-handling program may need a Suspend and Restore procedure to ensure the display is suspended and restored while the message handling function is being performed. The Suspend and Restore procedure is necessary only if the following conditions exist:

- A procedure in the break-program displays other menus or screens
- The break-program calls other programs which may display other menus or screens.

The following example clarifies the user procedure and display file needed to suspend and restore the display:

Note: RSTDSP(*YES) must be specified to create the display file.

```
      A          R SAVFMT                    OVERLAY  KEEP
      A*
      A          R DUMMY                     OVERLAY
      A                                      KEEP
      A                                      ASSUME
      A            DUMMYR         1A     1  2DSPATR(ND)
```

```
   PGM PARM(&MSGQ &MSGLIB &MRK)
       DCL VAR(&MSGQ) TYPE(*CHAR) LEN(10)
       DCL VAR(&MSGLIB) TYPE(*CHAR) LEN(10)
       DCL VAR(&MRK) TYPE(*DEC) LEN(4)
       DCLF FILE(UDDS/BRKPGMFM)
       SNDF RCDFMT(SAVFMT)
       CALL PGM(User's Break Program)
       SNDF RCDFMT(SAVFMT)
   ENDPGM
```

If you do not want the user specified break-handling program to interrupt the interactive job, the program may be submitted to run in batch. You may do this by specifying a break-handling program that receives the message and then performs a SBMJOB. The SBMJOB performs a call to the current break-handling program with any parameters that you want to use. (An example is information from the receive message.) Control will then be returned to the interactive job and it will continue normally.

QSYSMSG Message Queue

The QSYSMSG message queue is an optional queue that you can create in the QSYS library. If it exists and is not damaged, certain messages are directed to it instead of, or in addition to, the QSYSOPR message queue. This allows a user-written program to gain control when certain messages are sent. You should not create the QSYSMSG queue unless you want it to receive specific messages.

Enter the following command to create the QSYSMSG queue:

```
CRTMSGQ  QSYS/QSYSMSG +
         TEXT('Optional MSGQ to receive specific system messages')
```

Once the QSYSMSG message queue is created, all of the specific messages (shown below in "Messages Sent to QSYSMSG Message Queue") are directed to it. You can write a program to receive messages for which you can perform special action and send other messages to the QSYSOPR message queue or another message queue. This program should be written as a break-handling program.

Messages Sent to QSYSMSG Message Queue

This topic describes the specific messages sent to the QSYSMSG message queue. If the QSYSMSG message queue exists, the system sends the following messages to QSYSMSG instead of QSYSOPR:

- CPF1269
- CPF1393
- CPF1397
- CPI2209
- CPI9014
- CPI96C0 through CPI96C7

The system sends certain messages to both QSYSMSG and QSYSOPR, or only to QSYSMSG, or only to QSYSOPR, depending on the system reference code (SRC) sent with the message and whether the SRC is being logged with critical message handling. These messages include the following.

- CPP0DDD
- CPP1604
- CPPEA02
- CPPEA04
- CPPEA05
- CPPEA12
- CPPEA13
- CPPEA26
- CPPEA32
- CPPEA38

- CPPEA39

The system sends all other messages in this topic to both QSYSMSG and QSYSOPR.

CPD4070

```
Negative response received for remote location &5, device
description &4.
```

CPF0907

```
Serious storage condition may exist.  Press HELP.
```

This message is sent if the amount of available auxiliary storage in the system auxiliary storage pool has reached the threshold value.

The system service tools function can be used to display and change the threshold value. For more information, see the Backup and Recovery book.

CPF1269

```
Program start request received on communications device was rejected
with reason codes.
```

This message is sent when a start request is rejected and contains a reason code identifying why the rejection occurred.

If a password is not valid or an unauthorized condition occurs using APPC, it may mean that a normal job is in error or that someone is attempting to break security. You may choose to prevent further use of the APPC device description until the condition is understood by doing the following:

- Sending the message to the QSYSOPR message queue.
- Recording the attempt for the security officer to review.
- Issuing the End Mode (ENDMOD) command to set the allowed jobs to zero. This allows jobs that are currently using the peer device description to remain active, but prevents other jobs from starting until the condition is understood.
- Counting the number of attempts in a given time period. You could establish a threshold in your program for the number of attempts that were not valid before you take serious action (such as changing the maximum number of sessions to zero). You may want to assign this threshold value by unit of work identifier (which may be blank), by APPC device description, or for your entire APPC environment.

CPF1393

```
User profile has been disabled because maximum number of sign-on
attempts has been reached.
```

This message is sent when a user has attempted to sign-on multiple times, causing the User Profile to be disabled.

CPF1397

```
Subsystem varied off workstation.
```

This message is sent if the threshold value assigned by the system value QMAXSIGN is reached and the device is varied off. The message indicates that a user is not entering a valid password. The message data for CPF1397 contains the name of the device from which the message was sent. You can use this information and design a program to take appropriate action. You could consider performing one or several of the following:

- Send the same message to the QSYSOPR message queue
- Record the attempt for the security officer to review
- Automatically vary on the device after a significant time delay

CPF510E

`Network interface &9 failed while doing a read or write to device &4.`

The system has detected a network interface failure and is attempting error recovery for the network interface.

Try the request again. Recommendations on program recovery are in the Communications Management book. If the problem occurs again, enter the Analyze Problem (ANZPRB) command to run problem analysis.

CPF5167

`SNA session for remote location &5, device description &4 ended abnormally.`

The Systems Network Architecture (SNA) session ended due to a request shutdown (RSHUTD), request recovery (RQR), unbind (UNBIND), or notify power off (NOTIFY) command received from the remote controller.

Contact the remote controller operator to determine why the communications support ended the session. Correct the error, and try the request again.

CPF5244

`Internal system failure for remote location &5, device description &4.`

Vary off the device. Vary the device on and try the request again. If the problem continues, report the problem (ANZPRB command).

CPF5248

`SNA protocol violation for data received for remote location &5, device description &4.`

The Systems Network Architecture (SNA) request received for remote location &5, device description &4 violates SNA protocol. The system received a negative response with sense data &7 to the controller.

Correct the problem in the controller program and try the request again. For more information on sense data and associated errors, see the Finance Communications Programming book.

CPF5250

`Negative response with sense data &7 received for remote location &5.`

The system received a negative response with sense data &7 for remote location &5 device description &4. The first four characters of the data did not begin with 10xx, 08xx, or 0000. The Systems Network Architecture (SNA) session ended if it existed.

Correct the error and try the request again. For more information on sense data and the causes of negative responses, see *Systems Network Architecture Formats, GA27-3136.*

CPF5251

`Password or user ID not valid for request for remote location &5.`

A Systems Network Architecture (SNA) INIT-SELF command was received for finance remote location &5, device description &4 that did not contain valid authorization data. One of the following occurred:

- The system could not find the user ID or password.
- The system could not find the user ID.
- The password was not valid for this user ID.
- No authorization exists for this user ID to use device description &4.
- The user profile was not accessible.
- The user ID contained a character that is not valid.

Have the user try the request again with a valid user ID and password. If the user has no authorization to the device, use the Grant Object Authority (GRTOBJAUT) command to authorize the user to this device.

CPF5257

`Failure for device or member &4 file &2 in library &3.`

An error occurred during a read or write operation. If this is a display file, the display may not be usable.

See the previously listed messages, correct the errors, and try the request again. If the problem continues, report the problem (ANZPRB command).

CPF5260

`Switched connection failed for device &4 in file &2 in &3.`

Close the file and then try the request again.

CPF5274

`Error on device for remote location &5 file &2 in &3.`

The program attempted an input operation or an output operation to program device &4, remote location &5 that had a prior error.

Vary off device associated with remote location &5 and then on again (VRYCFG or WRKCFGSTS command). Then try the request again.

CPF5341

`SNA session not established for remote location &5, device description &4.`

The Systems Network Architecture (SNA) session could not be established. The Synchronous Data Link Control (SDLC) frame size is not compatible with the request/response unit (RU) size. This is either a configuration error, or the SDLC frame size has been negotiated to a smaller value by OS/400. This occured while the remote controller is using the Exchange Identification (XID) command.

The MAXLENRU parameter of the device description contains the specification of the RU size for retail and finance devices.

The MAXFRAME parameter of the line description contains the SDLC frame size specification. The MAXFRAME parameter of the controller description also contains the specification for retail and finance devices.

Do one or more of the following and try the request again:

- Verify that the frame size and the RU size values are compatible.
- Increase the SDLC frame size, or decrease the RU size, if necessary.
- Verify that this configuration is compatible with the remote controller configuration.

- If you are making configuration changes, you must vary the configuration off and on before the changes will take effect.

CPF5342

`Line &9 failed on device description &4, remote location &5.`

The system has detected line failure while processing input or output and is attempting error recovery for the line.

Try the request again. If the problem continues, start problem analysis (ANZPRB command). For more information on program recovery, see the Communications Management book.

CPF5344

`Error on controller &9, device description &4.`

The system has detected a controller failure and is attempting error recovery for the controller.

Try the request again. If the problem continues, start problem analysis (ANZPRB command).

CPF5346

`Error for remote location &5, device description &4.`

Close the file. Vary off the device (VRYCFG command). Look at any system operator messages in the job log to determine if any action is necessary before you vary on the device. Correct the error, and vary on the device (VRYCFG command). Then try the request again. If the problem continues, start problem analysis (ANZPRB command).

CPF5355

`Not able to locate object &7 in &8 of type *&9.`

Object &7 type *&9 is either being used in another process, is not varied on, or has no sessions available for advanced program-to-program communications.

Close file &2. Try the request again when object &7 is available or varied on. If the object that could not be allocated is for APPC, there were no sessions available to use. You can change the WAITFILE parameter to allow the system to wait longer for a session to become available. The mode can also be changed (MAXSSN parameter) to make more sessions available. The remote system may also have to configure again to accept a greater number of sessions. If configuration exists for sufficient sessions, use the CHGSSNMAX command to attempt to increase the current session limit.

CPF8AC4

`Reserved library name &7 in use.`

A user has created a library name that is reserved for the QDLS file system. For example, for user ASP 5, library name QDOC0005 is reserved for the system. The system will attempt to create this library when the user is trying to create the first document library object (DLO) on the user ASP. However, if a user has created their own library using the reserved name QDOC0005, then message CPF8AC4 will be sent. The user-created library must be deleted or renamed before a DLO can be created in the user ASP.

CPF9E7C

`Operating System/400 grace period expired.`

The software license grace period for Operating System/400 has expired. Successful completion of the next initial program load (IPL) requires a software license key.

Contact your IBM marketing representative or IBM business partner for a new Operating system/400 software license key. Use the Add License Key Information (ADDLICKEY) command to add the software license key.

CPI091F

`PWRDWNSYS &1 command in progress.`

This message is sent in a secondary partition, when the primary partition ends abnormally.

CPI0948

`Mirrored protection is suspended on disk unit &1;`

The system is not able to locate a storage unit. Data has not been lost. The following information indicates where the system located the storage unit before the storage unit was missing from the system configuration:

- Disk serial number: &5
- Disk type: &3
- Disk model: &4
- Device resource name: &26

Do the following:

1. Use the system Resource Configuration List display to see the storage unit that is identified as missing.
2. Ensure the proper installation of power cable connections on the storage unit.

CPI0949

`Mirrored protection is suspended on disk unit &1;`

The mirrored protection for the disk is suspended.

CPI0950

`Storage unit now available.`

A storage unit, which was missing from the configuration, is now available. Data has not been lost.

CPI0953

`ASP storage threshold reached.`

This message is sent if the amount of available storage in the specified auxiliary storage pool (ASP) has reached the threshold value. The message data for CPI0953 contains the auxiliary storage capacity, the auxiliary storage used, the percentage of threshold, and the percentage of auxiliary storage available. You can use this information to take appropriate action.

CPI0954

`ASP storage limit exceeded.`

This message is sent if all available storage in the specified ASP has been used.

CPI0955

`System ASP unprotected storage limit exceeded.`

This message is sent if all available storage in the system ASP has been used.

CPI095A

IASP &1 mirror copy storage threshold reached.

This message is sent if the amount of available storage for the mirror copy of the specified independent auxiliary storage pool (IASP) has reached the threshold value. The message data for CPI095A contains the independent auxiliary storage capacity, and the percentage of threshold. You can use this information to take appropriate action.

CPI0964

Weak battery condition exists.

This message is sent if the external uninterruptible power supply or internal battery indicates a weak battery condition.

CPI0965

Failure of battery power unit feature in system unit.

This message is sent if there is a failure of the battery or the battery charger for the battery power unit feature in the system unit.

CPI0966

Failure of battery power unit feature in expansion unit.

This message is sent if there is a failure of the battery or the battery charger for the battery power unit feature in the expansion unit.

CPI096B

Geographic mirroring is suspended for IASP &1.

This message is sent to indicate that geographic mirroring for the specified independent auxiliary storage pool (IASP) has been suspended due to an inability to communicate with the system containing the mirror copy. Check the specified system to determine if this is an expected situation or if it indicates a communications problem.

CPI096C

Geographic mirroring is still suspended for IASP &1.

This message is sent to indicate that geographic mirroring for the specified independent auxiliary storage pool (IASP) is still suspended. This message indicates that no action has been taken to resume geographic mirroring on an independent auxiliary storage pool that was previously suspended.

CPI096D

Copy of IASP has been rejected.

This message is sent to indicate that an attempt was made to configure a mirror copy of an independent auxiliary storage pool (IASP) on the same system that contains the production copy of that independent auxiliary storage pool (IASP). The mirror copy must be on a different system

CPI096E

Disk unit connection is missing.

This message is sent to indicate that all the expected connections to an Enterprise Storage Subsystem (ESS) have not reported in. The following information identifies the disk unit:

- Disk serial number: &5
- Disk type: &3
- Disk model: &4

This message may indicate that a cable has become disconnected, a configuration has been changed, or there is a problem. Use the disk unit information above to locate the unit in the Hardware service manager service tool to determine if it is a configuration change or a problem.

CPI0970

`Disk unit &1 not operating`

Disk unit &1 has stopped operating. No data has been lost. The following information identifies the disk unit that is not operating:

- Disk serial number: &3
- Disk type: &5
- Disk model: &6
- Disk address: &4
- IOP resource name: &26
- Device controller resource name: &27
- Device resource name: &28

Press F14 to run problem analysis.

CPI0988

`Mirrored protection is resuming on disk unit &1;`

This message is sent if the mirroring synchronization of a disk unit has started and disk mirroring protection is being resumed. One of the steps the system performs before disk mirroring protection is resumed is to copy data from one disk unit to another so that the data on both disk units is the same. You may observe slow system performance during the time that the data is being copied. After the copy of the disk data is complete, message CPI0989 is sent to this message queue, and disk mirroring protection resumes.

CPI0989

`Mirrored protection resumed on disk unit &1;`

This message is sent if the mirroring synchronization of a disk unit completed successfully. The system completed the copy of data from one disk unit to the other. Disk mirroring protection is resumed.

CPI0998

`Error occurred on disk unit &1;`

This message is sent if errors were found on disk unit &1; The message does not include information about the failure to run problem analysis.

CPI0999

`Storage directory threshold reached.`

The storage directory is nearing capacity. This is a potentially serious system condition. The system repeats this message until it receives an IPL.

You must reduce the amount of storage that is used on the system. To reduce the amount of storage that is used, do the following:

- Delete objects from the system that are not needed.
- Save objects that are not needed online by specifying STG(*FREE) on the Save Object (SAVOBJ) command.

CPI099C

`Critical storage lower limit reached.`

The amount of storage that is used in the system auxiliary storage pool has reached the critical lower limit value. The system will now take the action that is specified in the QSTGLOWACN system value: &5. The possible actions are:

- *MSG — The system takes no further action.
- *CRITMSG — The system sends message CPI099B to the user that is specified by the CRITMSGUSR service attribute.
- *REGFAC — The system submits a job to run the exit programs that are registered for the QIBM_QWC_QSTGLOWACN exit point.
- *ENDSYS — The system ends to the restricted state.
- *PWRDWNSYS — The system powers down immediately and restarts.

You can reduce use of storage through the following actions:

- Delete any unused objects.
- Save objects by specifying STG(*FREE)
- Save the old unused log versions of QHST and then delete them.
- Print or delete spooled files on the system.

Failure to reduce the storage usage may lead to a situation that requires initialization of auxiliary storage and loss of user data. Use the WRKSYSSTS command to monitor the amount of storage that is used. Use the PRTDSKINF command to print information about storage usage. The WRKSYSVAL command can be used to display and change the auxiliary storage lower limit value (QSTGLOWLMT) and action (QSTGLOWACN).

CPI099D

`System starting in storage restricted state.`

The system is being started to the restricted state because the amount of storage available is below the auxiliary storage lower limit. Failure to reduce storage usage may lead to a situation that requires initialization of auxiliary storage and the loss of user data. The console is the only active device.

You can reduce the use of storage through the following actions:

- Delete any unused objects.
- Save objects by specifying STG(*FREE).
- Save the old unused log versions of QHST and then delete them.
- Print or delete spooled files on the system.

Failure to reduce the storage usage may lead to a situation that requires initialization of auxiliary storage and loss of user data. Use the WRKSYSSTS command to monitor the amount of storage that is used. Use the PRTDSKINF command to print information about storage usage. The WRKSYSVAL command can be used to display and change the auxiliary storage lower limit value (QSTGLOWLMT) and action (QSTGLOWACN).

CPI099E

`Storage lower limit exit program error occurred.`

An error occurred while calling a user exit program for exit point QIBM_QWC_QSTGLOWACN. The reason code is &1. The reason codes and their meanings follow:

1. An error occurred while running the user exit program.
2. The system did not find a user exit program.

3. The system did not find a registered user exit program.
4. A user exit program did not complete in 30 minutes.
5. The job running the user exit program ended.
6. The system did not submit the user exit program job because the system is ending.
7. The system did not submit the user exit program job because errors occurred.
8. The system submitted the user exit program job, but issued warnings as well.
9. The system could not retrieve registration information for the exit point.
10. The system did not submit the user exit program job because the maximum number of failed exit program jobs was exceeded.
11. An unexpected error occurred in the user exit program job.

CPI099F

`PWRDWNSYS &1 command in progress.`

This message is sent in a secondary partition, when the primary partition powers down.

CPI116A

`Mirrored protection suspended on the load source disk unit.`

Suspension of mirrored protection occurs on disk unit 1. Data has not been lost. Disk unit 1 is attached to the Multi-Function I/O Processor (MFIOP). Repair the disk unit as soon as possible. Do not power down the system, IPL the system, or perform any operation which would IPL the system until after completing disk repairs to unit 1.

The following information identifies the unit that is suspended:

- Disk serial number: &5
- Disk type: &3
- Disk model: &4
- Device resource name: &26

The system will automatically resume mirroring after corrections are made to the error.

CPI116B

`Mirrored protection still suspended on the load source disk unit.`

Mirrored protection remains suspended on disk unit 1. Data has not been lost. Disk unit 1 is attached to the Multi-Function I/O Processor (MFIOP). Repair the disk unit as soon as possible. Do not power down the system, IPL the system, or perform any operation which would IPL the system until after repairing disk unit 1.

The following information identifies the unit that is suspended:

- Disk serial number: &5
- Disk type: &3
- Disk model: &4
- Device resource name: &26

See the previously listed messages in this message queue to determine the failure that caused suspension of mirrored protection. Perform the recommended recovery procedures.

CPI116C

`Compressed disk unit &1 is full.`

Compressed disk unit &1 is temporarily full. The storage subsystem controller has detected the condition and is repositioning the data on the compressed disk unit. The system does this to maximize the amount of storable data on the disk unit. This operation could take a number of minutes to complete. When the storage subsystem controller has completed the repositioning of the data, the system will resume normal operations.

The following information identifies the unit that is full:

- Disk serial number: &5
- Disk type: &3
- Disk model: &4
- Device resource name: &26

Wait for the storage subsystem controller to reposition the data on the compressed disk unit. Do not power off the system. If you are frequently receiving this message that indicates that a compressed disk unit is full, do one or more of the following:

1. Save objects that are not needed from the auxiliary storage pool by specifying STG(*FREE) on the Save Object (SAVOBJ) command.
2. Delete objects that are not needed from the auxiliary storage pool.
3. Move one or more folders to a different auxiliary storage pool by saving the folder, deleting the folder, and restoring the folder to a different auxiliary storage pool.
4. Increase the storage capacity by adding disk units to the auxiliary storage pool. You can direct the system to immediately overflow data from the user auxiliary storage pool into the system auxiliary storage pool. This prevents you from having to wait for the storage subsystem controller to reposition the data on the compressed disk unit each time it becomes full. Use the Change ASP Attribute (CHGASPA) command to change the compression recovery policy to immediately overflow data to the system auxiliary storage pool whenever a compressed disk unit becomes full.

CPI1117

`Damaged job schedule &1 in library &2 deleted.`

This message is sent when the job schedule in the library was deleted because of damage.

CPI1136

`Mirrored protection still suspended.`

This message is sent each hour if mirrored protection is still suspended on one or more disk units.

CPI1138

`Storage overflow recovered.`

This message is sent when ASP &1 no longer has any objects that have overflowed into the system ASP for reason &2;

CPI1139

`Storage overflow recovery failed.`

This message is sent when an attempt to recover from storage overflow failed.

CPI1153

`System password bypass period ended.`

This message is sent when the system has been operating with the system password bypass period in effect. The bypass period has ended. Unless the correct system password is provided, the next IPL will not be allowed to complete successfully.

CPI1154

`System password bypass period will end in &5 days.`

This message is sent when the system password (during a previous IPL) was either not entered or not entered correctly, and the system bypass period was selected.

CPI1159

`System ID will expire with &1 more installs.`

This message is sent when the system ID is about to expire. The IBM Service Representative should be contacted.

CPI1160

`System ID has expired.`

This message is sent when the system identifier has expired. The IBM Service Representative should be contacted.

CPI1161

`Unit &1 with device parity protection not fully operational.`

Unit &1 is part of a disk unit subsystem with device parity protection. Unit &1 requires service. The data has been saved. Reduced performance, machine checks, and possible data loss can occur if this condition is not corrected.

CPI1162

`Unit &1 with device parity protection not fully operational.`

Unit &1 is part of a disk unit subsystem with device parity protection. Unit &1 is not fully operational for one of the following reasons:

- The service representative is repairing the unit.
- The unit is not operating, but there is not enough information to run problem analysis.

CPI1165

`One or more device parity protected units still not fully operational.`

One or more units within disk unit subsystems with device parity protection are still not fully operational due to errors.

CPI1166

`Units with device parity protection fully operational. operational.`

Units for all IOP subsystems that provide device parity protection are fully operational.

CPI1167

`Temporary I/0 processor error occurred.`

An error condition occurred on a I/O processor with disk devices.

CPI1168

`Error occurred on disk unit &1;`

Disk unit number &1 found an error. A damaged object can occur. A machine check can occur if the problem becomes worse. The following identifies the disk unit.

CPI1169

`Disk unit &1 not operating.`

Disk unit &1 has stopped operating. No data has been lost.

CPI1171

`Internal system object cannot be recovered.`

The internal system object that contains an index of jobs on the system was damaged. The system could not recover the object after &1 attempts.

No recovery action is necessary, but the performance of searching for jobs may be affected.

CPI1468

`System work control block table nearing capacity.`

The system sends this message when the number of entries in the system job tables approaches the maximum number allowed. Permitting the job tables to fill completely may prevent the successful submission of jobs or the completion of the subsequent IPL.

CPI22AA

`Unable to write audit record to QAUDJRN.`

An unexpected &1 exception occurred at instruction &2 of program QSYSAUDR when attempting to write an audit record of type &3 to the QAUDJRN audit journal. The action specified by the Auditing End Action (QAUDENDACN) system value will be performed.

CPI2209

`User profile &1 deleted because it was damaged.`

This message is sent when a user profile is deleted because it was damaged. This user profile may have owned objects prior to being deleted. These objects now have no owner. A Reclaim Storage (RCLSTG) command can be used to transfer the ownership of these objects to the QDFTOWN user profile.

CPI2239

`QAUDCTL system value changed to &1.`

During the install, QAUDCTL was changed to *NONE because the security auditing function was not available. The security auditing function is now available so the QAUDCTL system value has been changed to its original value.

CPI2283

`QAUDCTL system value changed to *NONE.`

This message is sent each hour after auditing has been turned off by the system because auditing failed. To turn auditing back on or to determine why auditing failed, you can change system value QAUDCTL to a value other than *NONE.

CPI2284

QAUDCTL system value changed to *NONE.

This message is sent during IPL if auditing was turned off by the system because auditing failed. To turn auditing back on or to determine why auditing failed, you can change system value QAUDCTL to a value other than *NONE.

CPI8A13

QDOC library nearing save history limit.

This message is sent when the number of objects in library QDOC is approaching the limit for the number of objects that the system supports storing in one library.

CPI8A14

QDOC library has exceeded save history limit.

This message is sent when the number of objects in Library QDOC has exceeded the limit for the number of objects that the system supports in one library.

CPI9014

Password received from device not valid.

This message is sent when a password has been received on a document interchange session that is not correct. It may indicate unauthorized attempts to access the system.

CPI9490

Disk error on device &25;

This message is sent when a disk error has been detected.

CPI94A0

Disk error on device &25;

This message is sent when a disk error has been detected.

CPI94CE

Error detected in bus expansion adapter, bus extension adapter, System Processor, or cables.

This message is sent when the system has detected a failure in the main storage. System performance may be degraded. Run problem analysis to determine the failing card.

CPI94CF

Main storage card failure is detected.

This message is sent when the system has detected a failure in the main storage. System performance may be degraded. Run problem analysis to determine the failing card.

CPI94FC

Disk error on device &25;

This message is sent when the 9336 Disk Unit has determined that one of its parts is exceeding the error threshold and is starting to fail.

CPI96C0

`Protected password could not be validated.`

The system sends this message when the protected password received for the user profile by the APPC sign-on transaction program was not correct. This message contains a reason code that identifies the error. Check the reason code to take appropriate action.

CPI96C1

`Sign-on request GDS variable was not correct.`

The system sends this message when the sign-on request GDS variable received by the APPC sign-on transaction program was not correct. Remote program have to sent the correct sign-on data.

CPI96C2

`User password could not be changed.`

This messages is sent if a security problem has occured.

CPI96C3

`Message &4 returned on system call.`

The system sends this message when a message returns on a system call by the APPC sign-on transaction program.

CPI96C4

`Password not correct for user profile.`

The system sends this message when the password specified is not correct.

CPI96C5

`User &4 does not exist.`

The system sends this message when the received user does not exist on the system.

CPI96C6

`Return code &4 received on call to CPI-Communications.`

The system sends this message when a call to CPI-Communications has sent a return code. Refer to the *Common Programming Interface Communications Reference* manual to see the return code description and determine why it is failing.

CPI96C7

`System failure in the APPC sign-on transaction program.`

The system sends this message when receiving an unexpected error. Run problem analysis to determine the failing.

CPP0DD9

`A system processor failure is detected.`

This message is sent when a system processor or system processor cache has failed. System performance may be degraded.

CPP0DDA

`A system processor failure is detected in slot 9.`

This message is sent when a system processor or system processor cache has failed. System performance may be degraded.

CPP0DDB

`A system processor failure is detected in slot 10.`

This message is sent when a system processor or system processor cache has failed. System performance may be degraded.

CPP0DDC

A system processor failure is detected.

This message is sent when the system has detected an error on the system processor. System performance may be degraded.

CPP0DDD

System processor diagnostic code detected an error.

This message is sent when a failure has been detected by the system-processor diagnostics during IPL, but the system is still able to function. System performance may be degraded.

CPP0DDE

A system processor error is detected.

This message is sent when a control failure is detected on a system processor. Hardware ECC is correcting the failure. However, if you performed an initial program load (IPL), control could not be initialized and the system would reconfigure itself without that processor.

CPP0DDF

A system processor is missing.

This message is sent when a processor is missing on a multi-processor system.

CPP1604

Attention Impending DASD failure. Contact your hardware service provider now.

This message is sent when an unrecoverable error resulting in data loss is about to occur on a disk device.

CPP29B0

Recovery threshold exceeded on device &25;

This message is sent when one of the parts in the 9337 disk unit is starting to fail.

CPP29B8

Device parity protection suspended on device &25;

This message is sent when one of the parts in the 9337 disk array is failing. Protection for implementation of RAID 5 technique has been suspended on the disk array.

CPP29B9

Power protection suspended on device &25;

This message is sent when one of the power modules in the 9337 disk array is failing. Power protection has been suspended on the disk array.

CPP29BA

Hardware error on device &25;

This message is sent when one of the parts in the 9337 disk array has failed. Service action is required.

CPP951B

Battery power unit fault.

This message is sent when the battery power unit has failed.

CPP9522

`Battery power unit fault.`

This message is sent when the battery power unit in the 5042 Expansion Unit or 5040 Extension Unit has failed.

CPP955E

`Battery power unit not installed.`

This message is sent when the battery power unit in the 9406 System Unit power supply is not installed.

CPP9575

`Battery power unit in 9406 needs to be replaced.`

This message is sent when the battery power unit in the 9406 System Unit has failed and needs to be replaced. It may still work, but more than the recommended number of charge-discharge cycles have occurred.

CPP9576

`Battery power unit in 9406 needs to be replaced.`

This message is sent when the battery power unit in the 9406 System Unit has failed and needs to be replaced. It may still work, but it has been installed longer than recommended.

CPP9589

`Test of battery power unit complete.`

This message is sent when testing has been completed for the battery power unit, and the results have been logged.

CPP9616

`Battery power unit not installed.`

This message is sent when the battery power unit has not been installed in the 5042 Expansion Unit or 5040 Extension Unit power supply.

CPP9617

`Battery power unit needs to be replaced.`

This message is sent when the battery power unit in the 5042 Expansion Unit or 5040 Extension Unit needs to be replaced. It may still work, but more than the recommended number of charge-discharge cycles have occurred.

CPP9618

`Battery power unit needs to be replaced.`

This message is sent when the battery power unit in the 5042 Expansion Unit or 5040 Extension Unit needs to be replaced. It may still work, but it has been installed longer than recommended.

CPP961F

`DC Bulk Module 3 fault.`

This message is sent when the dc bulk module 3 of the 9406 System Unit has failed.

CPP9620

`DC Bulk Module 2 fault.`

This message is sent when the dc bulk module 2 of the 9406 System Unit has failed.

CPP9621

```
DC Bulk Module 1 fault.
```

This message is sent when the dc bulk module 1 of the 9406 System Unit has failed.

CPP9622

```
DC Bulk Module 1 fault.
```

This message is sent when the dc bulk module 1 of the 5042 Expansion Unit or 5040 Extension Unit has failed. Other dc bulk modules can also cause this fault.

CPP9623

```
DC Bulk Module 2 fault.
```

This message is sent when the dc bulk module 2 of the 5042 Expansion Unit or 5040 Extension Unit has failed. Other DC bulk modules can also cause this fault.

CPP962B

```
DC Bulk Module 3 fault.
```

This message is sent when the dc bulk module 3 of the 5042 Expansion Unit or 5040 Extension Unit has failed. Other dc bulk modules can also cause this fault.

CPPEA02

```
*Attention*  Contact your hardware service provider now.
```

This message is sent when internal analysis of an exception indicates that hardware service is required now.

CPPEA04

```
*Attention*  Contact your hardware service provider.
```

This message is sent when internal analysis of an exception indicates that hardware redundancy has been lost due to a permanent failure.

CPPEA05

```
*Attention*  Contact your hardware service provider.
```

This message is sent when internal analysis of an exception indicates that data protection facilities have been lost due to a permanent failure.

CPPEA12

```
*Attention*  Contact your hardware service provider now.
```

This message is sent when internal analysis of an exception indicates that an I/O card is operating at a reduced performance level.

CPPEA13

```
*Attention*  Contact your hardware service provider.  Internal
analysis of exception data indicates that hardware service is
recommended to maintain system performance.  It is recommended that
you contact your hardware service provider to have the cache battery
pack of an I/O card replaced. Failure to do so could result in
reduced system performance.
```

CPPEA26

```
*Attention*  Contact your hardware service provider.
```

This message is sent when internal analysis of an exception indicates that hardware service is recommended to maintain system availability.

CPPEA32
Storage subsystem configuration error.

This message is sent when either too many devices or the wrong kind of devices have been configured under an I/O card.

CPPEA38
Attention Contact your hardware service provider now. A system error has occurred.

CPPEA39
Attention Contact your hardware service provider now. A critical system error has occurred. The system will automatically IPL using redundant resources.

Sample Program to Receive Messages from QSYSMSG

The following is a sample program which receives messages from the QSYSMSG message queue. The program consists of a single procedure which is receiving messages and handling message CPF1269. The reason code in the CPF1269 message is in binary format. This must be converted to a decimal value for the comparisons to the 704 and 705 reason codes. The procedure issues the ENDMOD command to prevent new jobs from being started until the situation is understood. It then sends the same message to a user-defined message queue to be reviewed by the security officer. It also sends a message to the system operator informing him of what occurred. If a different message is received, it is sent to the system operator.

A separate job would be started to call this sample program. The job would remain active, waiting for a message to arrive. The job could be ended using the ENDJOB command.

```
/*********************************************************************/
/*                                                                   */
/*  Sample program to receive messages from QSYSMSG                  */
/*                                                                   */
/*********************************************************************/
/*                                                                   */
/*  Program looks for message CPF1269 with a reason code of 704      */
/*  or 705.  If found then notify QSECOFR of the security failure.   */
/*  Otherwise resend the message to QSYSOPR.                         */
/*                                                                   */
/*  The following describes message CPF1269                          */
/*                                                                   */
/*      CPF1269:  Program start request received on communications   */
/*                device &1 was rejected with reason codes &6,; &7;  */
/*                                                                   */
/*      Message data from DSPMSGD CPF1269                            */
/*                                                                   */
/*        Data  type   offset length Description                     */
/*                                                                   */
/*         &1   *CHAR      1     10  Device                          */
/*         &2   *CHAR     11      8  Mode                            */
/*         &3   *CHAR     19     10  Job - number                    */
/*         &4   *CHAR     29     10  Job - user                      */
/*         &5   *CHAR     39      6  Job - name                      */
/*         &6   *BIN      45      2  Reason code - major            */
/*         &7   *BIN      47      2  Reason code - minor            */
/*         &8   *CHAR     49      8  Remote location name           */
/*         &9   *CHAR     57  *VARY  Unit of work identifier        */
/*                                                                   */
/*********************************************************************/

          PGM
```

```
            DCL        &MSGID   *CHAR LEN(  7)
            DCL        &MSGDTA  *CHAR LEN(100)
            DCL        &MSG     *CHAR LEN(132)

            DCL        &DEVICE  *CHAR LEN( 10)
            DCL        &MODE    *CHAR LEN(  8)
            DCL        &RMTLOC  *CHAR LEN(  8)

            MONMSG     CPF0000  EXEC(GOTO PROBLEM)
            /*************************************************************/
            /* Fetch messages from QSYSMSG message queue                 */
            /*************************************************************/
LOOP:       RCVMSG     MSGQ(QSYS/QSYSMSG) WAIT(*MAX) MSGID(&MSGID) +
                         MSG(&MSG) MSGDTA(&MSGDTA)

            IF         ((&MSGID *EQ 'CPF1269') /* Start failed msg */ +
              *AND      ((%BIN(&MSGDTA 45 2) *EQ 704)                 +
                *OR      (%BIN(&MSGDTA 45 2) *EQ 705)) )              +
            THEN(DO)
              /*********************************************************/
              /* Report security failure to QSECOFR                    */
              /*********************************************************/

              CHGVAR     &DEVICE %SST(&MSGDTA 1 10) /* Extract device */
              CHGVAR     &MODE   %SST(&MSGDTA 11 8) /* Extract mode   */
              CHGVAR     &RMTLOC %SST(&MSGDTA 49 8) /* Get loc name   */

              ENDMOD     RMTLOCNAME(&RMTLOC) MODE(&MODE)

              SNDPGMMSG  MSGID(&MSGID) MSGF(QCPFMSG) MSGDTA(&MSGDTA) +
                           TOMSGQ(QSECOFR)

              SNDPGMMSG  MSG('Device ' *CAT &DEVICE *TCAT ' Mode '  +
                           *CAT &MODE *TCAT ' had security failure, +
                           session max changed to zero')           +
                           TOMSGQ(QSYSOPR)
            ENDDO
            ELSE DO
              /*********************************************************/
              /* Other message - Resend to QSYSOPR                     */
              /*********************************************************/

              SNDPGMMSG  MSGID(&MSGID) MSGF(QCPFMSG) MSGDTA(&MSGDTA) +
                           TOMSGQ(QSYSOPR)

              /* SNDPGMMSG would fail if the message does            */
              /*   not have a MSGID or is not in QCPFMSG             */

              MONMSG     MSGID(CPF0000)  +
                           EXEC(SNDPGMMSG MSG(&MSG) TOMSGQ(QSYSOPR))
            ENDDO

            GOTO       LOOP    /* Go fetch next message              */


            /*************************************************************/
            /* Notify QSYSOPR of abnormal end                            */
            /*************************************************************/

PROBLEM:    SNDPGMMSG  MSG('QSYSMSG job has abnormally ended') +
                         TOMSGQ(QSYSOPR)
            MONMSG     CPF0000

            SNDPGMMSG  MSGID(CPF9898) MSGF(QCPFMSG) MSGTYPE(*ESCAPE) +
```

```
                   MSGDTA('Unexpected error occurred')
MONMSG     CPF0000

ENDPGM
```

Using the System Reply List

The system reply list allows you to specify that the system issue the reply to specified predefined inquiry messages so the display station user does not need to reply. Only inquiry messages can be automatically responded to.

The system reply list contains message identifiers, optional compare data, a reply value for each message, and a dump attribute. The system reply list applies only to predefined inquiry messages that are sent by a job that uses the system reply list. You specify that a job is to use the system reply list for inquiry messages on the INQMSGRPY(*SYSRPYL) parameter on the following commands:

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

When a predefined inquiry message is sent by a job that uses the system reply list, the system searches the reply list in ascending sequence number order for an entry that matches the message identifier and, optionally, the compare data of the reply message. If an entry is found, the reply specified is issued and the user is not required to enter a reply. If an entry is not found, the message is sent to the display station user for interactive jobs or system operator for batch jobs.

The system reply list is shipped with the system with the following initial entries defined:

Sequence Number	Message Identifier	Compare Value	Reply	Dump
10	CPA0700	*NONE	D	*YES
20	RPG0000	*NONE	D	*YES
30	CBE0000	*NONE	D	*YES
40	PLI0000	*NONE	D	*YES

These entries indicate that a reply of D is to be sent and a job dump is to be taken if the message CPA0700-CPA0799, RPG0000-RPG9999, CBE0000-CBE9999, or PLI0000-PLI9999 (which indicate a program failure) is sent by a job using the system reply list. For the system to use these entries, you must specify that the jobs are to use the system reply list.

To add other inquiry messages to the system reply list, use the Add Reply List Entry (ADDRPYLE) command. On this command you can specify the sequence number, the message identifier, optional compare data, compare data CCSID, reply action, and the dump attribute. The ADDRPYLE command function can be easily accessed by using the Work with System Reply List Entries (WRKRPYLE) command.

The following reply actions can be specified for the inquiry messages that are placed on the system reply list (the parameter value is given in parentheses):

- Send the default reply for the inquiry messages (*DFT). In this case, the default reply for the message is sent. The message is not displayed, and no default handling program is called.
- Require the workstation user or system operator to respond to the message (*RQD). If the message queue to which the message is sent (workstation message queue for interactive jobs and QSYSOPR for batch jobs) is in break mode, the message is displayed, and the workstation user must respond to the message. This option operates as if the system reply list were not being used.
- Send the reply specified in the system reply list entry (message reply, 32 characters maximum). In this case, the specified reply is sent as the response to the message. The message is not displayed, and no default handling program is called.

The following commands add entries to the system reply list for messages RPG1241, RPG1200, CPA4002, CPA5316, and any other inquiry messages:

- `ADDRPYLE SEQNBR(15) MSGID(RPG1241) RPY(C)`
- `ADDRPYLE SEQNBR(18) MSGID(RPG1200) RPY(*DFT) DUMP(*YES)`
- `ADDRPYLE SEQNBR(22) MSGID(CPA4002) RPY(*RQD) +  CMPDTA('QSYSPRT')`
- `ADDRPYLE SEQNBR(25) MSGID(CPA4002) RPY(G)`
- `ADDRPYLE SEQNBR(27) MSGID(CPA5316) RPY(I) DUMP(*NO) + CMPDTA('QSYSPRT' 21)`
- `ADDRPYLE SEQNBR(9999) MSGID(*ANY) RPY(*DFT)`

The system reply list now appears as follows:

Sequence Number	Message Identifier	Compare Value (b is a blank)	Compare Start Position	Reply	Dump
10	CPA0700		1	D	*YES
15	RPG1241		1	C	*NO
18	RPG1200		1	*DFT	*YES
20	RPG0000		1	D	*YES
22	CPA4002	'QSYSPRT'	1	*RQD	*NO
25	CPA4002		1	G	*NO
27	CPA5316	'QSYSPRT'	21	I	*NO
30	CBE0000		1	D	*YES
40	PLI0000		1	D	*YES
9999	*ANY		1	*DFT	*NO

For a job that uses this system reply list, the following occurs when the messages that were added to the reply list are sent by the job:

- For sequence number 15, whenever an RPG1241 message is sent by a job that uses the system reply list, a reply of C is sent and the job is not dumped.
- For sequence number 18, a generic message identifier is used so whenever an RPG1200 inquiry message is sent by the job, the default reply is sent. The default reply can be the default reply specified in the message description or the

system default reply. Before the default reply is sent, the job is dumped. The previous entry that was added overrides this entry for message RPG1241.

- For sequence number 22, if the inquiry message CPA4002 is sent with the compare data of QSYSPRT, the message is sent to the display station user, and the user must issue the reply.

  When a compare value is specified without a start position, the compare value is compared to the message data beginning in position 1 of the substitution data in the message.

  Sequence number 22 tests for a printer device name of QSYSPRT. For an example of testing one substitution variable with a different start position, see sequence number 27.
- For sequence number 25, if the inquiry message CPA4002 (verify alignment on printer &1;) is sent with the compare not equal to QSYSPRT, a reply of G is sent. The job is not dumped. Sequence number 22 requires an operator response to the forms alignment message if the printer device is QSYSPRT. Sequence number 25 defines that if the forms alignment inquiry message occurs for any other device, to assume a default response of G=Go.
- For sequence number 27, if the inquiry message CPA5316 is sent with the compare data of `TESTEDFILETSTLIBRARYQSYSPRT`, a reply of `I` is sent.

  When a compare value and a start position are specified, the compare value is compared with the message data beginning with the start position. In this case, position 21 is the beginning of the third substitution variable. For message CPA5316, the first four substitution variables are as follows:

&1	ODP file name	*CHAR	10
&2	ODP library name	*CHAR	10
&3	ODP device name	*CHAR	10
&4	Line number for first line	*BIN	2

  Therefore, sequence number 27 tests for an ODP device name of QSYSPRT before sending a reply.
- For sequence number `9999`, the message identifier of *ANY applies to any predefined inquiry message that is not matched by an entry with a lower sequence number, and the default reply for these inquiry messages is sent. If this entry were not included in the system reply list, the display station user would have to respond to all other inquiry messages that were not included in the system reply list.

When the compare value contains *CCHAR data, the message data that is from the sending function is converted to the CCSID of the message data that is stored in the system reply list before the compare is made. The system converts only data that is of type *CCHAR. IBM provides online information about the Add Message Description (ADDMSGD) command *CCHAR data. Refer to the *CL* section of the **Programming** category in the iSeries Information Center.

CAUTION:
The following restrictions apply when using *CCHAR data as compare data:

- **You cannot mix *CCHAR data with other data when adding this type of reply list entry.**
- **You cannot include the length of the *CCHAR data in the compare data.**

If you mix *CCHAR data or include the length of the *CCHAR data, unpredictable results may occur.

An entry remains on the system reply list until you use the Remove Reply List Entry (RMVRPYLE) command to remove it. You can use the Change Reply List Entry (CHGRPYLE) command to change the attributes of a reply list entry, and you can use the Work with System Reply List Entry (WRKRPYLE) command to display the reply entries currently in the reply list.

The job log receives a completion message indicating a successful change when the system reply list is updated using (ADDRPYLE), (CHGRPYLE), or (RMVRPYLE). The history log QHST also receives a completion message to record the change.

Reply Handling

The Reply Handling Exit Program allows a user-supplied exit program to be called when a reply is sent to an inquiry message. The exit program can accept, reject or replace the reply value. For details about the Reply Handling Exit Program, refer to Message Handling APIs in the *APIs* section of the**Programming** category in the **iSeries Information Center**.

Message Logging

The two types of logs for messages are:

- Job log
- History log

A job log contains information related to requests entered for a job. The QHST log contains system data, such as a history of job start and end activity on your system.

Job Log

Each job has an associated job log that can contain the following for the job:

- The commands in the job.
- The commands in a CL program if the CL program was created with the LOG(*YES) option or with the LOG(*JOB) option and a Change Job (CHGJOB) command was run with the LOGCLPGM(*YES) option (for more information on logging CL program commands, see “Logging CL Procedure Commands” on page 60).
- All messages and message help sent to the requester and not removed from the call message queues.

After the job ends, the job log can be written to either the output file QPJOBLOG or a database file. From the output file QPJOBLOG, the job log can be printed; from a database file, job log information can be queried using a database feature. You can also specify to not write job logs for jobs that have run successfully—a discussion about preventing job logs is presented later in this chapter.

You can use the QMHCTLJL API to write a job log to a database file. For information about the QMHCTLJL API, refer to the *APIs* section of the **Programming** category in the **iSeries Information Center**. When directing the job log to the database file, the system can generate one or two files. The primary file contains the essential information of a message such as message ID, message severity, message type, and message data. The secondary file contains the print images of the message text. Parameters on the QMHCTLJL API control the optional production of the secondary file. You can use database and query features on the

system to externally describe and process both files. See Appendix B, "Job Log Output Files," on page 403 for information on the formats of the primary and secondary files.

You can control what information the system writes in the job log. To do this, you specify the LOG parameter on the Create Job Description (CRTJOBD) command. You can change these levels by using the Change Job (CHGJOB) command or the Change Job Description (CHGJOBD) command. Three values make up the LOG parameter: Message level, message severity, and message text level. For more information on these commands, see the *CL* section of the **Programming** category of the iSeries Information Center.

The first value, message level, has the following levels:

Level **Description**

0 No data is logged.

1 The only information to be logged is all messages sent to the job's external message queue with a severity greater than or equal to the message severity specified. Messages of this type indicate when the job started, when it ended, and its status at completion.

2 The following information is logged:

- Level 1 logging information.
- Any requests that result in high-level messages with a severity greater than or equal to the severity specified. If the request is logged, all of its associated messages are also logged.

3 The following information is logged:

- Logging level 1 and 2 information.
- All requests.
- Commands run by a CL program if allowed by the `Log CL program commands` job attribute and the `Log` attribute of the CL program.

4 The following information is logged:

- All requests and all messages with a severity code greater than or equal to the severity specified, including trace messages.
- Commands run by a CL program if allowed by the `Log CL program commands` job attribute and the `Log` attribute of the CL program.

  Note: A high-level message is one that is sent to the program message queue of the program that receives the request. For example, QCMD is an IBM-supplied request processing program that receives requests.

The second value, message severity, specifies the severity level in conjunction with the log level that causes error messages to be logged in the job log. Values 0 through 99 are allowed.

The third value in the LOG parameter, message text level, specifies the level of message text that is written in the job log. The values are:

***SAME**
The current value for the message text level does not change.

***MSG** Only message text is written to the job log (message help is not included).

***SECLVL**
The message and the message help (cause and recovery) are written to the job log.

Before each new request is received by a request processing program, message filtering occurs. **Message filtering** is the process of removing messages from the job log based on the message logging level set for the job.

Filtering does not occur after every CL command is called within a program. Therefore, if a CL program is run interactively or submitted to batch, the filtering runs once after the program ends because the program is not a request processor.

Note: Since *NOLIST specifies that no job log is spooled for jobs that end normally, it is a waste of system resource in batch jobs to remove the messages from this log by specifying log level 0.

The following example shows how the logging level affects the information that is stored in the job message queue and, therefore, written to the job log, if one is requested. The example also shows that when the command is run interactively, filtering occurs after each command.

Note: Both high-level and detailed message logging levels are included in the examples. High-level messages are identified as *Message*; detailed messages are identified as *Detailed Message*.

1. The CHGJOB command specifies a logging level of 2 and a message severity of 50, and that only messages are to be written to the job log (*MSG).

```
                              Command Entry                        SYSTEM1
                                                        Request level:   1
Previous commands and messages:
     > CHGJOB LOG(2 50 *MSG)
```

2. PGMA sends three informational messages with severity codes of 20, 50, and 60 to its own call message queue and to the call message queue of the program called before the specified call (*PRV). The messages that PGMA sends to its own call message queue are called detailed messages. **Detailed messages** are those messages that are sent to the call message queue of the lower-level program call.

   PGMB sends two informational messages with severity codes of 40 and 50 to its own call message queue. These are detailed messages. PGMB also sends one informational message with a severity code of 10 to *PRV.

   Note that the CHGJOB command no longer appears on the display. According to logging level 2, only requests for which a message has been issued with a severity equal to or greater than that specified are saved for the job log, and no messages were issued for this request. If such a message had been issued, any detailed messages that had been issued would be saved for the job log and could be displayed by pressing F10.

```
                              Command Entry                        SYSTEM1
                                                          Request level:   1
Previous commands and messages:
      > CALL PGMA
        Message sev 20 - PGMA
        Message sev 50 - PGMA
        Message sev 60 - PGMA
      > CALL PGMB
        Message sev 10 - PGMB






                                                                      Bottom
Type command, press Enter.
===> ____________________________________________________________________
____________________________________________________________________________
____________________________________________________________________________
____________________________________________________________________________
 F3=Exit   F4=Prompt   F9=Retrieve   F10=Include detailed messages
 F11=Display full      F12=Cancel    F13=Information Assistant   F24=More keys
```

3. When F10=Include detailed messages is pressed from the Command Entry display, all the messages associated with the request are displayed. Press F10 again to exclude detailed messages.

```
                              Command Entry                        SYSTEM1
                                                          Request level:   1
All previous commands and messages:
   > CALL PGMA
     Detailed message sev  20 - PGMA
     Detailed message sev  50 - PGMA
     Detailed message sev  60 - PGMA
     Message sev 20 - PGMA
     Message sev 50 - PGMA
     Message sev 60 - PGMA
   > CALL PGMB
     Detailed message sev  40 - PGMB
     Detailed message sev  50 - PGMB
     Message sev 10 - PGMB

                                                                      Bottom
Type command, press Enter.
===> ____________________________________________________________________
____________________________________________________________________________
____________________________________________________________________________
____________________________________________________________________________
 F3=Exit   F4=Prompt   F9=Retrieve   F10=Exclude detailed messages
 F11=Display full      F12=Cancel    F13=Information Assistant    F24=More Keys
```

4. When another command is entered (in this example, CHGJOB), the `CALL PGMB` command and all messages (including detailed messages) are removed. They are removed because the severity code for the high-level message associated with this request was not equal to or greater than the severity code specified in the CHGJOB command. The `CALL PGMA` command and its associated messages remain because at least one of the high-level messages issued for that request has a severity code equal to or greater than that specified.

   On the following display, the CHGJOB command specifies a logging level of 3, a message severity of 40, and that both the first- and second-level text of a message are to be written to the job log. When another command is entered, the CHGJOB command remains on the display because logging level 3 logs all requests.

   PGMC sends two messages with severity codes of 30 and 40 to the call message queue of the program called before the specified call (*PRV).

PGMD sends a message with a severity of 10 to *PRV.

```
                              Command Entry                        SYSTEM1
                                                        Request level:   1
Previous commands and messages:
     > CALL PGMA
       Message sev 20 - PGMA
       Message sev 50 - PGMA
       Message sev 60 - PGMA
     > CHGJOB LOG(3 40 *SECLVL)
     > CALL PGMC
       Message sev 30 - PGMC
       Message sev 40 - PGMC
     > CALL PGMD
       Message sev 10 - PGMD


                                                                      Bottom
Type command, press Enter.
===> ____________________________________________________________________________
____________________________________________________________________________
____________________________________________________________________________
____________________________________________________________________________
 F3=Exit   F4=Prompt   F9=Retrieve   F10=Include detailed messages
 F11=Display full      F12=Cancel     F13=Information Assistant    F24=More Keys
```

5. When another command is entered after the CALL PGMD command was entered, the CALL PGMD command remains on the display, but its associated message is deleted. The message is deleted because its severity code is not equal to or greater than the severity code specified on the LOG parameter of the CHGJOB command.

   The command SIGNOFF *LIST is entered to print the job log.

```
                              Command Entry                        SYSTEM1
                                                        Request level:   1
Previous commands and messages:
     > CHGJOB LOG(3 40 *SECLVL)
     > CALL PGMC
       Message sev 30 - PGMC
       Message sev 40 - PGMC
     > CALL PGMD
     > CALL PGME









                                                                      Bottom
Type command, press Enter.
===> SIGNOFF *LIST___________________________________________________________
____________________________________________________________________________
____________________________________________________________________________
____________________________________________________________________________
 F3=Exit   F4=Prompt   F9=Retrieve   F10=Include detailed messages
 F11=Display full      F12=Cancel     F13=Information assistant    F24=More Keys
```

The job log, which follows, contains all the requests and all the messages that have remained on the Command Entry display. In addition, the job log contains the message help associated with each message, as specified by the CHGJOB command. Notice that the job log contains the message help of any message issued during the job, not just for the messages issued since the second CHGJOB command was entered.

```
 5722SS1 V5R3M0 040905                      Job Log                          SYSAS727 01/16/04 07:13:35         Page    1
  Job name . . . . . . . . . . :   QPADEV000C      User  . . . . . . :   JOHNDOE      Number . . . . . . . . . . . :   038518
  Job description  . . . . . . :   QDFTJOBD       Library . . . . . :   QGPL
MSGID      TYPE                 SEV  DATE      TIME            FROM PGM     LIBRARY     INST     TO PGM      LIBRARY     INST
CPF1124    Information          00   01/16/04  07:13:19.570504  QWTPIIPP     QSYS        0613     *EXT                    *N
                                  Message . . . . :   Job 038518/JOHNDOE/QPADEV000C started on 01/16/04 at
```

```
                                07:13:19 in subsystem QINTER in QSYS. Job entered system on 01/16/04 at
                                07:13:19.
*NONE      Request                01/16/04  07:13:24.318144  QMHGSD       QSYS        0010     QCMD        QSYS        0178
                              Message . . . . :  -CALL PGMA
MSG1001    Information          20   01/16/04  07:13:24.361064  PGMA         JOHNDOE     0029     PGMA        JOHNDOE     0029
                              From User . . . . . . . . . :   MARYJANE
                              Message . . . . :   Detailed message sev 20 - PGMA
                              MSG1001 second level text - PGMA
MSG1002    Information          50   01/16/04  07:13:24.361416  PGMA         JOHNDOE     0032     PGMA        JOHNDOE     0032
                              Message . . . . :   Detailed message sev 50 - PGMA
                              MSG1002 second level text - PGMA
MSG1003    Information          60   01/16/04  07:13:24.361592  PGMA         JOHNDOE     0036     PGMA        JOHNDOE     0036
                              Message . . . . :   Detailed message sev 60 - PGMA
                              MSG1003 second level text - PGMA
MSG1004    Information          20   01/16/04  07:13:24.361776  PGMA         JOHNDOE     003A     QCMD        QSYS        01A6
                              Message . . . . :   Message sev 20 - PGMA
                              MSG1004 second level text - PGMA
MSG1005    Information          50   01/16/04  07:13:24.362192  PGMA         JOHNDOE     0043     QCMD        QSYS        01A6
                              From User . . . . . . . . . :   MARYJANE
                              Message . . . . :   Message sev 50 - PGMA
                              MSG1005 second level text - PGMA
MSG1006    Information          60   01/16/04  07:13:24.362552  PGMA         JOHNDOE     004C     QCMD        QSYS        01A6
                              Message . . . . :   Message sev 60 - PGMA
                              MSG1006 second level text - PGMA
*NONE      Request                01/16/04  07:13:24.370240  QMHGSD       QSYS        0018     QCMD        QSYS        0178
                              Message . . . . :  -CHGJOB LOG(3 40 *SECLVL)
*NONE      Request                01/16/04  07:13:24.370672  QMHGSD       QSYS        001C     QCMD        QSYS        0178
                              Message . . . . :  -CALL PGMC
MSG100F    Information          30   01/16/04  07:13:24.379256  PGMC         JOHNDOE     *STMT    QCMD        QSYS        01A6
                              From User . . . . . . . . . :   MARYJANE
                              From module . . . . . . . . :   PGMC
                              From procedure  . . . . . . :   PGMC
                              Statement . . . . . . . . . :   8000
                              Message . . . . :   Message sev 30 - PGMC
                              MSG100F second level text - PGMC
MSG1010    Information          40   01/16/04  07:13:24.379608  PGMC         JOHNDOE     *STMT    QCMD        QSYS        01A6
                              From module . . . . . . . . . :   PGMC
                              From procedure  . . . . . . :   PGMC
                              Statement . . . . . . . . . :   8200
 5722SS1 V5R3M0 030905                          Job Log                          LPAR3TLM 01/16/04 07:13:35          Page    2
  Job name . . . . . . . . . . :   QPADEV000C      User  . . . . . . :   JOHNDOE      Number . . . . . . . . . . . :   038518
  Job description  . . . . . . :   QDFTJOBD        Library . . . . . :   QGPL
MSGID      TYPE                 SEV  DATE      TIME             FROM PGM     LIBRARY     INST     TO PGM      LIBRARY     INST
                              Message . . . . :   Message sev 40 - PGMD
                              MSG1010 second level text - PGMC
*NONE      Request                01/16/04  07:13:24.379928  QMHGSD       QSYS        0020     QCMD        QSYS        0178
                              Message . . . . :  -CALL PGMD
*NONE      Request                01/16/04  07:13:24.383568  QMHGSD       QSYS        0024     QCMD        QSYS        0178
                              Message . . . . :  -CALL PGME
*NONE      Request                01/16/04  07:13:35.166408  QMHGSD       QSYS        073E     QCMD        QSYS        0178
                              Message . . . . :  -signoff *list
CPF1164    Completion           00   01/16/04  07:13:35.173496  QWTMCEOJ     QSYS        00BD     *EXT                    *N
                              Message . . . . :   Job 038518/JOHNDOE/QPADEV000C ended on 01/16/04 at
                                07:13:35; 1 seconds used; end code 0 .
                              Cause . . . . . :   Job 038518/JOHNDOE/QPADEV000C completed on 01/16/04 at
                                07:13:35 after it used 1 seconds processing unit time.  The job had ending
                                code 0. The job ended after 1 routing steps with a secondary ending code of
                                0.  The job ending codes and their meanings are as follows:  0 - The job
                                completed normally. 10 - The job completed normally during controlled ending
                                or controlled subsystem ending. 20 - The job exceeded end severity (ENDSEV
                                job attribute). 30 - The job ended abnormally. 40 - The job ended before
                                becoming active. 50 - The job ended while the job was active. 60 - The
                                subsystem ended abnormally while the job was active. 70 - The system ended
                                abnormally while the job was active. 80 - The job ended (ENDJOBABN command).
                                90 - The job was forced to end after the time limit ended (ENDJOBABN
                                command). Recovery  . . . :   For more information, see the Work Management
                                topic in the Information Center, http://www.iseries.ibm.com/infocenter.
```

The headings at the top of each page of the printed job log identify the job to which the job log applies and the characteristics of each entry:

- The product ID, version, and date of the operating system
- The system name
- The date and time the job log was printed.
- The fully qualified name of the job (job name, user name, and job number).
- The name of the job description used to start the job.
- The section number. This is printed if the job log is printed in multiple sections because the job log wrapped and *PRTWRAP was specified for the job message queue full action.
- Headings for the information in the first line of each message entry.

The following information is printed for each message entry in the job log:

- The first line for each message contains the following information:
  - The message identifier or *NONE.
  - The message type.
  - The message severity. This is blank for request messages.

  - The date and time each message was sent.
  - The program name, library name, and instruction number of the program that sent the message.
  - The program name, library name, and instruction number of the program to which the message was sent. *EXT indicates that the message was sent to the external message queue of the job.
- If the message was sent by a user other than the user identified as the user in the qualified job name, the name of the user that sent the message is printed on a separate line. This can indicate one of the following situations:
  - The message was sent while the job was running under a different user profile. In the sample job log shown above, messages MSG1001, MSG1005, and MSG100F were sent while the job was running under a different user profile.
  - An inquiry message was replied to by another user.
  - A batch job was submitted by a user other than the user profile that the batch job runs under. In this case the name of the submitting user is included for each request message.
  - Another user changed the job attributes.
- If the sender is an ILE procedure, additional lines are printed to identify the module, procedure, and statement number. See "Sending or Receiving Program or Procedure" for additional information. In the sample job log shown above, for message MSG100F, PGMC is an ILE program.
- If the job is a multithreaded job, and messages were sent from more than one thread, the thread identifier is printed for each message.
- The message is printed on one or more lines.
- If the logging level indicates that the second-level text is to be included, the second level text appears on subsequent lines below the message.

Sending or Receiving Program or Procedure

When the sender or receiver is an ILE procedure, the message entry contains the full name of the procedure (procedure name, module name, and ILE program name). When the sender or receiver is an original program model (OPM) program, only the OPM program name is shown.

If the sender or receiver is an OPM program, the corresponding instruction number represents an instruction number. There is only one such number. If the sender or receiver is an ILE procedure, the instruction number represents a high level language statement number rather than an MI instruction number. If the ILE procedure has been optimized (maximum efficiency), there may be up to three numbers. It is not always possible to determine a single statement number for an optimized procedure. If there is more than one number given, each number represents a potential point where the procedure was when the message was sent. It is also possible that no number can be determined. If this the case, *N appears in the message rather than a number.

The logging levels affect a batch job log in the same way as shown in the preceding example. If the job uses APPC, the heading contains a line showing the unit of work identifier for APPC.

Additional Message Filtering

If the job log is directed to a database file through use of the QMHCTLJL API, additional message filtering can be specified. The message filtering specified through this API is applied when the job ends and the records for the messages are being written to the file. Up until this time, the messages which are now filtered

have appeared. Thus they can be seen while the job is running. When the job log is written, the messages which are filtered will have no records written to the file for them. Thus, even though they appear while the job is running they will not appear in the final file that is produced.

Displaying the Job Log

The way to display a job log depends on the status of the job.

- If the job has ended and the job log is not yet printed, use the Display Spooled File (DSPSPLF) command, as follows:

  ```
  DSPSPLF FILE(QPJOBLOG) JOB(001293/FRED/WS3)
  ```

  to display the job logs for job number 001293 associated with user FRED at display station WS3.
- If the job is still active (batch or interactive jobs) or is on a job queue and has not yet started, use the Display Job Log (DSPJOBLOG) command. For example, to display the job log of the interactive job for user JSMITH at display station WS1, enter:

  ```
  DSPJOBLOG JOB(nnnnnn/JSMITH/WS1)
  ```

  where `nnnnnn` is the job number.

To display the job log of your own interactive job, do one of the following:

- Enter the following command:

  ```
  DSPJOBLOG
  ```
- Enter the WRKJOB command and select option 10 (Display job log) from the Work with Job display.
- Press F10=Include detailed messages from the Command Entry display (this key displays the messages that are shown in the job log).
- If the input-inhibited light on your display station is on and remains on, do the following:
  1. Press the System Request key, then press the Enter key.
  2. On the System Request menu, select option 3 (Display current job).
  3. On the Display Job menu, select option 10 (Display Job log, if active or on job queue).
  4. On the Job Log display, DSPJOB appears as the processing request. Press F10 (Display detailed messages).
  5. On the Display All Messages display, press the Roll Down key to see messages that were received before you pressed the System Request key.
- Sign off the workstation, specifying LOG(*LIST) on the SIGNOFF command.

When you use the Display Job Log (DSPJOBLOG) command, you see the Job Log display. This display shows program names with special symbols, as follows:

>> The running command or the next command to be run. For example, if a program was called, the call to the program is shown.

> The command has completed processing.

. . The command has not yet been processed.

? Reply message. This symbol marks both those messages needing a reply and those that have been answered.

On the Job Log display, you can do the following:

- Press F10 to display detailed messages. This display shows the commands or operations that were run within an HLL program or within a CL program or procedure for which LOG is activated.
- Use the cursor movement keys to get to the end of the job log. To get to the end of the job log quickly, press F18 (Bottom). After pressing F18, you might need to roll down to see the command that is running.
- Use the cursor movement keys to get to the top of the job log. To get to the top of the job log quickly, press F17 (Top).

You may use the DSPJOBLOG command to direct the job to a database file instead of having it printed or displayed. There are two options available. In the first option, you may specify a file and member name on the command. In this option, the primary job log information is written to the database file specified on the command. In the second option you may use the command in conjunction with the information provided on the QMHCTLJL API which was run previously. In this option, the job log is written to the file(s) specified on the API call. With this option, both primary and secondary files can be produced and message filtering can be performed as the messages are written to the file. With both these options, when the DSPJOBLOG command completes, the output will not be displayed nor will there be a spooled file available for printing.

Preventing the Production of Job Logs

To prevent a job log from being produced at the completion of a batch job, you can specify *NOLIST for the message text-level value of the LOG parameter on the Batch Job (BCHJOB), Submit Job (SBMJOB), Change Job (CHGJOB), Create Job Description (CRTJOBD), or Change Job Description (CHGJOBD) command. If you specify *NOLIST for the message level value of the LOG parameter, the job log is not produced at the end of a job unless the job end code is 20 or greater. If the job end is 20 or greater, the job log is produced.

For an interactive job, the value specified for the LOG parameter on the SIGNOFF command takes precedence over the LOG parameter value specified for the job.

Job Log Considerations

The following suggestions apply to using job logs:

- To change the output queue for all jobs on the system, use the OUTQ or DEV parameter on the Change Printer File (CHGPRTF) command to change the file QSYS/QPJOBLOG. The following are two examples using each of the parameters:

```
CHGPRTF FILE(QSYS/QPJOBLOG)
        DEV (USRPRT)

or

CHGPRTF FILE(QSYS/QPJOBLOG)
        OUTQ(USROUTQ)
```

- To change the QPJOBLOG printer file to use output queue QEZJOBLOG, use the Operational Assistant cleanup function. When you want to use automatic cleanup of the job logs, the printer files must be directed to this output queue. For more information on the Operational Assistant cleanup function, see the *Getting Started with iSeries* topic **Systems management** category of the iSeries Information Center.
- To specify the output queue to which a job's job log is written, make sure that file QPJOBLOG has OUTQ(*JOB) specified. You can use the OUTQ parameter on any of the following commands: BCHJOB, CRTJOBD, CHGJOBD, or CHGJOB. The following is an example:

```
CHGJOB OUTQ(*JOB)
```
If you change the default OUTQ at the beginning of the job, all spooled files are affected. If you change it just before job completion, only the job log is affected. You cannot use the Override with Printer File (OVRPRTF) command to affect the job log.

- If the output queue for a job cannot be found, no job log is produced.
- To hold all job logs, specify HOLD(*YES) on the CHGPRTF command for the file QSYS/QPJOBLOG. The job logs are then released to the writer when the Release Spooled File (RLSSPLF) command is run. The following is an example:

```
CHGPRTF FILE(QSYS/QPJOBLOG)
        HOLD(*YES)
```

- If the system abnormally ends, the start prompt allows the system operator to specify whether the job logs are to be printed for any jobs that were active at the time of the abnormal end.
- To delete a job log, use the Delete Spooled File (DLTSPLF) command or the Delete option on the output queue display.
- If you used the USRDTA parameter on the Change Print File (CHGPRTF) command to change the user data value for the QSYS/QPJOBLOG file, the value specified will not be shown on the Work with Output Queue or Work with All Spooled Files displays. The value shown in the user data column is the job name of the job whose job log has printed.
- If job logs are being analyzed by programming technique, use the QMHCTLJL API to direct the job log to the database file(s). The format of the records in the database file is guaranteed while the printed format is not. If new fields need to be added to a job log record, they are added at the end of the record so existing programs will continue to work. Query features provided by the system can be used directly on the files.

Considerations for Interactive Job Logs

The IBM-supplied job descriptions QCTL, QINTER, and QPGMR all have a log level of LOG(4 0 *NOLIST); therefore, all messages and both first- and second-level text for the messages are written to the job log. However, the job logs are not printed unless you specify *LIST on the SIGNOFF command. To change the log level for interactive jobs, you can use the CHGJOB or CHGJOBD command.

If a display station user uses an IBM-supplied menu or the command entry display, all error messages are displayed. If the display station user uses a user-written initial program, any unmonitored message causes the initial program to end and a job log to be produced. However, if the initial program monitors for messages, it receives control when a message is received. In this case, it may be important to ensure that the job log is produced so you can determine the specific error that occurred. For example, assume that the initial program displays a menu that includes a sign-off option, which defaults to *NOLIST. The initial program monitors for all exceptions and includes a Change Variable (CHGVAR) command that changes the sign-off option to *LIST if an exception occurs:

```
        PGM
        DCLF MENU
        DCL &SIGNOFFOPT TYPE(*CHAR) LEN(7) VALUE(*NOLIST)
        .
        .
        .
        MONMSG MSG(CPF0000) EXEC(GOTO ERROR)
PROMPT: SNDRCVF RCDFMT(PROMPT)
        CHGVAR &IN41 '0'
        .
        .
```

```
            .
            IF (&OPTION *EQ '90') SIGNOFF LOG(&SIGNOFFOPT)
            .
            .
            .
            GOTO PROMPT
ERROR:      CHGVAR &SIGNOFFOPT '*LIST'
            CHGVAR &IN41 '1'
            GOTO PROMPT
            ENDPGM
```

If an exception occurs in the previous example, the CHGVAR command changes the option on the SIGNOFF command to *LIST and sets on an indicator. This indicator could be used to condition a constant that displays a message informing the display station user that an unexpected event has occurred and telling him what to do.

If the interactive job is running a CL program or procedure, the CL commands are logged only if the log level is 3 or 4 and one of the following is true:

- You specified LOG(*YES) on the Create Control Language Program (CRTCLPGM) command, the Create Control Language Module (CRTCLMOD) command, or the Create Bound CL Program (CRTBNDCL) command.
- You specified LOG(*JOB) on the Create Control Language Program (CRTCLPGM) command, the Create Control Language Module (CRTCLMOD) command, or the Create Bound CL Program (CRTBNDCL) command, and (*YES) is the current LOGCLPGM job attribute.

  You can set and change the LOGCLPGM job attribute by using the LOGCLPGM parameter on the SBMJOB, CRTJOBD, CRTJOBD, and CHGJOBD commands.

Considerations for Batch Job Logs

For your batch applications, you may want to change the amount of information logged. The log level (LOG(4 0 *NOLIST)) specified in the job description for the IBM-supplied subsystem QBATCH supplies a complete log if the job abnormally ends. If the job completes normally, no job log is produced.

If you want to print the job log in all cases, use the Change Job Description (CHGJOBD) command to change the job description, or specify a different LOG value on the BCHJOB or SBMJOB command. See “Job Log” on page 272 for a description of logging levels.

If a batch job is running a CL program or procedure, the CL commands are *always* logged if you specify LOG(*YES) when you create modules or programs using the following commands:

- Create Control Language Program (CRTCLPGM)
- Create Control Language Module (CRTCLMOD)
- Create Bound Control Language Program (CRTBNDCL)

CL commands are also logged if you specify LOGCLPGM(*YES) when you use the CHGJOB command and the SBMJOB command.

QHST History Log

The history log (QHST) consists of a message queue and a physical file known as a log-version. Messages sent to the history log message queue are written by the system to the current log version physical file.

- History log (QHST). Contains a high-level trace of system activities such as system, subsystem, and job information, device status, and system operator messages. Its message queue is QHST.

When a log-version is full, a new version of the log is automatically created. Each version is a physical file that is named in the following way:

```
Qxxxyydddn
```

where:

xxx Is a 3-character description of the log type (HST)

yyddd Is the Julian date of the first message in the log version

n Is a sequence number within the Julian date (A through Z and 0 through 9)

Note: The number of records in each version of the history log is specified in the system value QHSTLOGSIZ.

The text of the log version file contains the date and time of the first message and last message in the log version. The date and time of the first message are in positions 1-13 of the text; the date and time of the last message are in positions 14-26. The date and time are in the format cyymmddhhmmss, where:

c Is the century guard digit

yymmdd Is the date the message was sent

hhmmss Is the time the message was sent

You can create a maximum of 36 log versions with the same Julian date. If more than 36 log versions are created on the same day, the next available Julian day is used for subsequent log versions. If some of the older log versions are then deleted, it is possible to use the names again. Log versions are out of order when sequenced by name if log versions are deleted and names used again.

You can also write a program to process history log records. Because several versions of each log may be available, you must select the log-version to be processed. To determine the available log-versions, use the Display Object Description (DSPOBJD) command. For example, the following DSPOBJD command displays what versions of the history log are available:

```
DSPOBJD   OBJ(QSYS/QHST*)   OBJTYPE(*FILE)
```

You can delete logs on your system by using the delete option from the display that is presented on the Work with Objects (WRKOBJ) command.

You can display or print the information in a log using the Display Log (DSPLOG) command. You can select the information you want displayed or printed by specifying any combination of the following:

- Period of time
- Name of job that sent the log entry
- Message identifiers of entries

The following DSPLOG command displays all the available entries for the job OEDAILY in the current day:

```
DSPLOG JOB(OEDAILY)
```

The resulting display is:

```
                    Display History Log Contents

Job OEDAILY started
Database file OEMSTR in library OELIB expired
Job OEDAILY abnormally ended
Job OEDAILY started
Job OEDAILY ended

                                                                    Bottom
Press Enter to continue.

F3=Exit   F10=Display all   F12=Cancel
```

If you reset the system date or time to an earlier setting, or if you advanced the system date and time by more than 48 hours, a new log version is started. This ensures that all messages in a single log version are in chronological order.

Log versions created on a release prior to V3R6M0 may contain entries that are not in chronological order if the system date and time was reset to an earlier setting. Therefore, when you try to display the log-version, some entries may be missed. For example, if the log-version contains entries dated 1988 followed by entries dated 1987, and you want to display those 1987 entries, you specify the 1987 dates on the PERIOD parameter on the DSPLOG command but the expected entries are not displayed. You should always use the system date (QDATE) and the system time (QTIME), or you should specify the PERIOD parameter as follows:

```
PERIOD((start-time start-date) (*AVAIL *END))
```

The system writes the messages sent to a log message queue to the current version physical file when the message queue is full or when the DSPLOG command was used. If you want to ensure the current version is up-to-date, specify a fictitious message identifier, such as `###0000`, on the DSPLOG command. No messages are displayed, but the log-version physical file is made current.

If you print the information in a log using the Display Log command and output parameter *PRINT, (DSPLOG OUTPUT(*PRINT), only one line from each message is printed, using the first 105 characters of each message.

If you print the information in a log using the Display Log command and output parameter *PRTWRAP, (DSPLOG OUTPUT(*PRTWRAP), messages longer than 105 characters are wrapped to include additional lines to a limit of 2000 characters.

If you print the information in a log using the Display Log command and output parameter *PRTSECLVL, messages longer than 105 characters are wrapped to include additional lines to a limit of 2000 characters. The second-level message text is also printed if available, up to a maximum of 6000 characters.

If you display the information in a log using the Display Log (DSPLOG) command, only 105 characters of message text are shown. Any characters after 105 characters are truncated at the right.

Format of the History Log

A database file is used to store the messages sent to a log on the system. Because all records in a physical file have the same length and messages sent to a log have different lengths, the messages can span more than one record. Each record for a message has three fields:

- System date and time (a character field of length 8). This is an internal field. The converted date and time also are in the message.
- Record number (a 2-byte field). For example, the field contains hex `0001` for the first record, hex `0002` for the second record, and so on.
- Data (a character field of length 132).

The third field (data) of the first record has the following format:

Contents	Type	Length	Positions in Record
Job name	Character	26	11-36
Converted date and time[1]	Character	13	37-49
Message ID	Character	7	50-56
Message file name	Character	10	57-66
Library name	Character	10	67-76
Message type[2]	Character	2	77-78
Severity code	Character	2	79-80
Sending program name[3]	Character	12	81-92
Sending program instruction number[4]	Character	4	93-96
Receiving program name[3]	Character	10	97-106
Receiving program instruction number[4]	Character	4	107-110
Message text length	Binary	2	111-112
Message data length	Binary	2	113-114
Coded character set identifier (CCSID) for text or data[5]	Binary	4	115-118
Sending user profile	Character	10	119–128
Reserved	Character	14	129-142

Contents	Type	Length	Positions in Record
:			
1 The format is: cyymmddhhmmss where: **c** Is the century digit (c=0 if yy ≥ 40, c = 1 if yy < 40) **yymmdd** Is the year, month, and day that the message is sent **hhmmss** Is the hour, minute, and second that the message is sent 2 This has the same value as the RTNTYPE parameter on the Receive Message (RCVMSG) command. 3 If the sender or receiver is an ILE procedure, the entry in the history log contains only the ILE program name. The module name and procedure name are not included in the history log. 4 If the sender or receiver is an ILE procedure, the sending or receiving instruction number is 0. 5 This CCSID applies only to the message data that is defined as *CCHAR data if the message is a stored message. The rest of the message data can be considered 65 535. Otherwise, this is the CCSID of the immediate message.			

The third field (data) of the remaining records has the following format:

Contents	Type	Length
Message	Character	Variable[1]
Message data	Character	Variable[2]
:		
1 This length is specified in the first record (positions 111 and 112) and cannot exceed 132. 2 This length is specified in the first record (positions 113 and 114).		

A message is never split when a new version of a log is started. The first and last records of a message are always in the same QHST version.

For a description of the message data for a message, see "Defining Substitution Variables" on page 190.

Processing the QHST File

If you use an HLL program to process the QHST file, keep in mind that the length of the message can vary with each occurrence of a message. Because the message includes replaceable variables, the actual length of the message varies; therefore, the message data begins at a variable location for each use of the same message.

QHST Job Start and Completion Messages

The system performs special formatting of the job start and job completion messages. For message CPF1124 (job start) and message CPF1164 (job completion), the message data always begins in position 11 of the third record.

Job accounting provides more information than CPF1124 and CPF1164. For simple job accounting functions, use the CPF1164 message.

Performance information is not displayed as text on message CPF1164. Because the message is in the QHST log, users can write application programs to retrieve this data. The format of this performance information is as follows.

The performance information is passed as a variable length replacement text value. This means that the data is in a structure with the first entry being the length of the data. The size of the length field is not included in the length. The first data fields in the structure are the times and dates that the job entered the system and when the first routing step for the job was started. The times are in the format 'hh:mm:ss'. The separators are always colons. The dates are in the format defined in the system value QDATFMT and the separators are in the system value QDATSEP. The time and date the job entered the system precede the job start time and date in the structure.

The time and date the job entered the system are when the system becomes aware of a job to be initiated (a job structure is set aside for the job). For an interactive job, the job entry time is the time the password is recognized by the system. For a batch job, it is the time the BCHJOB or SBMJOB command is processed. For a monitor job, reader or writer, it is the time the corresponding start command is processed, and for autostart jobs it is during the start of the subsystem.

Following the times and dates are the total response time and the number of transactions. The total response time is in seconds and contains the accumulated value of all the intervals the job was processing between pressing the Enter key at the workstation and when the next display is shown. This information is similar to that shown on the WRKACTJOB display. This field is only meaningful for interactive jobs.

It is also possible in the case of a system failure or abnormal job end that the last transaction will not be included in the total. The job end code in this case would be a 40 or greater. The transaction count is also only meaningful for interactive jobs other than the console job and is the number of response time intervals counted by the system during the job.

The number of synchronous auxiliary I/O operations follows the number of transactions. This is the same as the AUXIO field that appears on the WRKACTJOB display except that this value is the total for the job. If the job ends with a end code of 70, this value may not contain the count for the final routing step. Additionally, if a job exists across an IPL (using a TFRBCHJOB command) it is ended before becoming active following an IPL, the value is 0.

The final field in the performance statistics is the job type. Values for this field are:

A Automatically started job

B Batch job

I Interactive job

M Subsystem monitor

R Spooling reader

S System job

W Spooling writer

X Start job

For messages in which the message data begins in a variable position, you can access the message data by doing the following:

- Determine the length of the variables in the message. For example, assume that a message uses the following five variables:

  ```
  Job name    *CHAR 10
  User name   *CHAR 10
  Job number  *CHAR 6
  Time        *CHAR 8
  Date        *CHAR 8
  ```

  These variables are fixed in the first 42 positions of the message data.
- To find the location of the message data, consider that:
  - The message always begins in position 11 of the second record.
  - The length of the message is stored in a 2-position field beginning at position 111 of the first record. This length is stored in a binary value so if the message length is 60, the field contains hex `003C`.

  Then, by using the length of the message and the start position of the message, you can determine the location of the message data.

Deleting QHST Files

Log-version physical files accumulate on a system and you should periodically delete old logs that are not needed. A log-version is created such that only the security officer is authorized to delete it.

The Operational Assistant* provides a cleanup function which includes the deletion of old QHST files. Another alternative is:

- As the security officer, specify:

  ```
  WRKOBJ  OBJ(QSYS/QHST*)  OBJTYPE(*FILE)
  ```

  Use option 4 to delete old unneeded files.

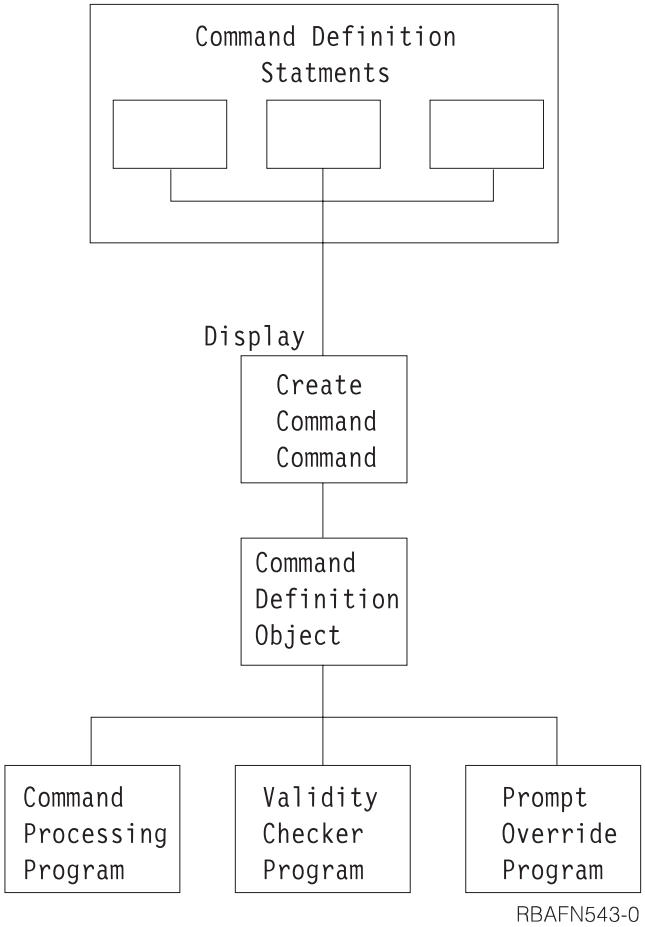
Chapter 9. Defining Commands

A CL command is a statement that requests that the system perform a function. Entering the command starts a program that performs the function. CL commands allow you to request a broad range of functions. You can use these IBM-supplied commands, change the default values that are supplied by IBM, and even define your own commands. This chapter describes how you can define and create your own commands. If you need help with the abbreviations of CL commands and keywords, see Appendix D, "Abbreviations of CL Commands and Keywords," on page 419.

When you define and create your own CL commands, you may also want to provide documentation for them. See Chapter 10, "Documenting Commands," on page 353 for information about how to create online help for your own commands, and how to create HTML pages from the help.

Overview of How to Define Commands

The following illustration shows the steps to create a command. The text that follows the illustration describes each step.



Writing your own validity checking and prompt override programs are optional steps.

© Copyright IBM Corp. 1997, 2004

Step Description

Command Definition Statements

The command definition statements contain the information that is necessary to prompt the workstation user for input, to validate that input, and to define the values to be passed to the program that is called when the command is run.

Command definition statements may exist in any file supported as input to the CRTCMD command. Table 9 contains the statements used for defining commands.

Table 9. Statements for Defining CL Commands

Statement Type	Statement Name	Description
Command	CMD	Specifies the prompt text, if any, for the command name
Parameter	PARM	Defines a parameter for a command
Element	ELEM	Defines an element in a list used as a parameter value
Qualifier	QUAL	Defines a qualifier of a name used as a parameter
Dependency	DEP	Defines a dependency for a parameter or among parameters
Prompt Control	PMTCTL	Defines the conditions under which certain parameters are prompted.

Create Command (CRTCMD) Command

The CRTCMD command processes the command definition statements to create the command definition object. The CRTCMD command may be run interactively or in a batch job.

Command Definition Object

The command definition object is the object that is checked by a system program to ensure that the command is valid and that the correct parameters were entered.

Validity Checking

The system performs validity checking on commands. You may also write your own validity checking program although it is not required.

The validity checking performed by the system ensures that:

- Values for the required parameters are entered.
- Each parameter value meets data type and length requirements.
- Each parameter value meets optional requirements specified in the command definition of:
  - A list of valid values
  - A range of values
  - A relational comparison to a value
- Conflicting parameters are not entered.

The system performs validity checking when:

- Commands are entered interactively from a display station.
- Commands are entered from a batch input stream using spooling.
- Commands are entered into a database file through the source entry utility (SEU).

- A command is passed to the QCMDEXC, QCMDCHK, or QCAPCMD program by a call from a HLL. See Chapter 6 for more information on the QCMDEXC program.
- A CL module or OPM program is created.
- Commands are run by a CL procedure or program or a REXX procedure.
- A command is run using the C language system function.

If you need more validity checking than the system performs, you can write a program called a *validity checking program* (see "Writing a Validity Checking Program" on page 345) or you can include the checking in the command processing program. You specify the names of both the command processing and validity checking programs on the CRTCMD command.

If a command has a validity checking program, the system passes the command parameter values to the validity checking program. This happens before the system calls the command processing program. A validity checking program runs during syntax checking during the following conditions:

- When running the command.
- When using the source entry utility (SEU) to enter commands into a CL source member and the programmer uses constants instead of variables for the parameters that are specified on the command.
- When compiling a CL program or procedure that uses constants instead of variables for all the parameters that are specified on the command.

When the program finds an error, the user receives a message to allow immediate correction of errors. The command processing program can assume that the data that is passed to it is correct. Refer to "Writing a Validity Checking Program" on page 345 for further information on writing validity checking programs.

Prompt Override Program

You can write prompt override programs to supply current values for parameter defaults when prompting the command. For example, prompt override programs are frequently used on Change commands to supply values for parameters with a default value of *SAME. See "Using Key Parameters and a Prompt Override Program" on page 325 for more details. A prompt override program is optional.

Command Processing Program

The command processing program (CPP) is the program that the command analyzer calls to perform the function requested. The CPP can be a CL program, another HLL program, or a REXX procedure. For example, it can be an application program that your command calls, or it can be a CL program or REXX procedure that contains a system command or series of commands.

The CPP must accept the parameters as defined by the command definition statements.

Command Exit Programs and Independent ASPs

Any exit program, including the command processing program, validity checking program, prompt override program, choices program, or prompt control program, needed by a command must be in the same independent auxiliary storage pool (ASP) as the command, or in the system ASP (ASP 1), or in a basic ASP (ASPs 2-32). The command must not be in one independent ASP and the exit programs in another independent ASP. Problems could occur when running the command if the independent ASP where these exit programs reside is not available (for example, if the independent ASP device is varied off).

Authority Needed for the Commands You Define

For users to use a command you create, they must have operational authority to the command and data authority to the command processing program and optional validity checking program. They also must have read authority to the library that contains the command, to the command processing program, and to the validity checking program. If the command processing program or the validity checking program refers to any service programs, the user must have execute authority to the service programs and to the service program libraries. The user must have execute authority to the programs that are listed below.

- Command Processing Program (CPP).
- Validity Checking Program (VCP).
- Any service programs that are used by the CPP or VCP.
- The libraries that contain the CPP, VCP, and service programs.

The user must also have the correct authority to any other commands run in the command processing programs. The user must also have authority to the files to open them.

Example of Creating a Command

If you want to create a command to allow the system operator to call a program to start the system, you would do the following. (This example assumes you are using IBM-supplied source files.)

1. Enter the command definition source statement into the source file QCMDSRC using the member name of STARTUP.

   ```
   CMD PROMPT('S Command for STARTUP')
   ```

2. Create the command by entering the following command.

   ```
   CRTCMD CMD(S) PGM(STARTUP) SRCMBR(STARTUP)
   ```

3. Enter the source statements for the STARTUP program (the command processing program).

   ```
   PGM
   STRSBS QINTER
   STRSBS QBATCH
   STRSBS QSPL
   STRPRTWTR DEV(QSYSPRT) OUTQ(QPRINT) WTR(WTR)
   STRPRTWTR DEV(WSPR2) OUTQ(WSPRINT) WTR(WTR2)
   SNDPGMMSG MSG('STARTUP procedure completed') MSGTYPE(*COMP)
   ENDPGM
   ```

4. Create the program using the Create Bound CL Program (CRTBNDCL) command.

   ```
   CRTBNDCL STARTUP
   ```

In the previous example, S is the name of the new command (specified by the CMD parameter). STARTUP is the name of the command processing program (specified by the PGM parameter) and also the name of the source member that contains the command definition statement (specified by the SRCMBR parameter). Now the system operator can either enter `S` to call the command or `CALL STARTUP` to call the command processing program.

How to Define Commands

To create a command, you must first define the command through command definition statements. Refer to the *CL* section of the **Programming** category of the iSeries Information Center for details. The general format of the command definition statements and a summary of the coding rules follow.

Statement
Coding Rules

CMD One and only one CMD statement must be used. The CMD statement can be placed anywhere in the source file.

PARM A maximum of 99 PARM statements is allowed. The order in which you enter the PARM statements into the source file determines the order in which the parameters are passed to the command processing program (CPP) and validity checking program (VCP). One PARM statement is required for each parameter that is to be passed to the command processing program. To specify a parameter as a key parameter, you must specify KEYPARM(*YES) for the PARM statement. The number of parameters coded with KEYPARM(*YES) should be limited to the number needed to uniquely define the object to be changed. To use key parameters, the prompt override program must be specified when creating the command. Key parameters cannot be defined with PMTCTL(*PMTRQS) or PMTCTL(label).

ELEM A maximum of 300 ELEM statements is allowed in one list. The order in which you enter the ELEM statements into the source file determines the order of the elements in the list. The first ELEM statement must have a statement label that matches the statement label on the TYPE parameter on the PARM or ELEM statement for the list.

QUAL A maximum of 300 qualifiers is allowed for a qualified name. The order in which you enter the QUAL statements into the source file determines the order in which the qualifiers must be specified and the order in which they are passed to the validity checking program and command processing program.

DEP The DEP statement must be placed after all PARM statements it refers to. Therefore, the DEP statements are normally placed near the end of the source file.

PMTCTL
The PMTCTL statement must be placed after all PARM statements it refers to. Therefore, the PMTCTL statements are normally placed at the end of the source file.

At least one PARM statement must precede any ELEM or QUAL statements in the source file. The source file in which you enter the command definition statements is used by the CRTCMD command when you create a command. For information on entering statements into a source file, see the ADTS for AS/400®: Source Entry Utility (SEU) book.

Using the CMD Statement

When you define a command, you must include one and only one CMD statement with your command definition statements.

When you define a command, you can provide command prompt text for the user. If the user chooses to be prompted for the command instead of entering the entire command, the user types in the command name and presses F4 (Prompt). The command prompt is then displayed with the command name and the heading prompt text on line 1 of the display.

If you want to specify prompt text for the command, use the PROMPT parameter on the CMD statement to specify the heading for the prompt. You then specify the

prompts for the parameters, elements of a list, and qualifiers on the PROMPT parameters for the PARM, ELEM, and QUAL statements.

On the PROMPT parameter for the CMD statement, you can specify the actual prompt heading text as a character string 30 characters maximum, or you can specify the message identifier of a message description. In the following example, a character string is specified for the command ORDENTRY.

```
CMD PROMPT('Order Entry')
```

Line 1 of the prompt looks like this after the user types in the command name and presses F4.

```
Order Entry (ORDENTRY)
```

If you do not provide prompt text for the command you are defining, you only need to use the word CMD for the CMD statement. However, you may want to use the PROMPT keyword for documentation purposes.

Defining Parameters

You can define as many as 99 parameters for each command. To define a parameter, you must use the PARM statement.

On the PARM statement, you specify the following:

- Name of the keyword for the parameter
- Whether or not the parameter is a key parameter
- Type of parameter value that can be passed
- Length of the value
- If needed, the default value for the parameter.

In addition, you must consider the following information when defining parameters. (The associated PARM statement parameter is given in parentheses.)

- Whether a value is returned by the command processing program (RTNVAL). If RTNVAL (*YES) is specified, a return variable must be coded on the command when it is called, if you want to have the value returned. If no variable is specified for a RTNVAL(*YES) parameter, a null pointer is passed to the command processing program.
- Whether the parameter is not to appear on the prompt to the user but is to be passed to the command processing program as a constant (CONSTANT).
- Whether the parameter is restricted (RSTD) to specific values (specified on the VALUES, SPCVAL, or SNGVAL parameter) or can include any value that matches the parameter type, length, value range, and a specified relationship.
- What the specific valid parameter values are (VALUES, SPCVAL, and SNGVAL).
- What tests should be performed on the parameter value to determine its validity (REL and RANGE).
- Whether the parameter is optional or required (MIN).
- How many values can be specified for a parameter that requires a simple list (MIN and MAX).
- Whether unprintable data (any character with a value of hexadecimal 00 through 3F or FF can be entered for the parameter value (ALWUNPRT).
- Whether a variable name can be entered for the parameter value (ALWVAR).
- Whether the value is a program name (PGM).
- Whether the value is a data area name (DTAARA).

- Whether the value is a file name (FILE).
- Whether the value must be the exact length specified (FULL).
- Whether the length of the value should be given with the value (VARY).
- Whether expressions can be specified for a parameter value (EXPR).
- Whether attribute information should be given about the value passed for the parameter (PASSATR).
- Whether to pass a value to the command processing program or validity checking program if the parameter being defined is not specified (PASSVAL).
- Whether the case value is preserved or the case value is converted to uppercase (CASE).
- Whether list within list displacements (LISTDSPL) are 2-byte or 4-byte binary values.
- What the message identifier is or what the prompt text for the parameter is (PROMPT).
- What valid values are shown in the possible choices field on the prompt display (CHOICE).
- Whether the choice values are provided by a program (CHOICEPGM).
- Whether prompting for a parameter is controlled by another parameter (PMTCTL).
- Whether values for a PMTCTL statement are provided by a program (for parameters referred to in CTL keywords) (PMTCTLPGM).
- Whether the value is to be hidden in the job log or hidden when the command is being prompted (DSPINPUT).

Naming the Keyword for the Parameter

The name of the keyword you choose for a parameter should be descriptive of the information being requested in the parameter value. For example, USER for user name, CMPVAL for compare value, and OETYPE for order entry type. The keyword can be as long as 10 alphanumeric characters, the first of which must be alphabetic.

Parameter Types

The basic parameter types are (parameter TYPE value given in parentheses):

- Decimal (*DEC). The parameter value is a decimal number, which is passed to the command processing program as a packed decimal value of the length specified on the LEN parameter. Values specified with more fractional digits than defined for the parameter are truncated.
- Logical (*LGL). The parameter value is a logical value, '1' or '0', which is passed to the command processing program as a character string of length 1 (`F1` or `F0`).
- Character (*CHAR). The parameter value is a character string, which can be enclosed in apostrophes and which is passed to the command processing program as a character string of the length specified on the LEN parameter. The value is passed with its apostrophes removed, is left-justified, and is padded with blanks.
- Name (*NAME). The parameter value is a character string that represents a basic name. The maximum length of the name is 256 characters. The first character is alphabetic (A-Z), $, #, or @. The remaining characters are the same as the first character, but can also include the numbers 0 through 9, underscores (_), and periods (.). The name can also be a string of characters that begin and end with double quotation marks (″). The system passes the value to the command processing program as a character string of the length specified in the LEN parameter. The value is left-justified and padded with blanks. Normally, you use

the *NAME type for object names. If you can enter a special value such as *LIBL or *NONE for the name parameter, you must describe the special value on the SPCVAL parameter. Then, if the display station user enters one of the allowed special values for the parameter, the system bypasses the rules for name verification.

- Simple name (*SNAME). The parameter value is a character string that follows the same naming rules as *NAME, except that no periods (.) are allowed.
- Communications name (*CNAME). The parameter value is a character string that follows the same naming rules as *NAME, except that no periods (.) or underscores (_) are allowed.
- Path name (*PNAME). The parameter value is a character string, which can be enclosed in apostrophes and which is passed to the command processing program as a character string of the length specified on the LEN parameter. The value is passed with its apostrophes removed, is left-justified, and is padded with blanks.
- Generic name (*GENERIC). The parameter value is a generic name, which ends with an asterisk (*). If the name does not end with an asterisk, then the generic name is assumed to be a complete object name. A generic name identifies a group of objects whose names all begin with the characters preceding the asterisk. For example, INV* identifies the objects whose names begin with INV, such as INV, INVOICE, and INVENTORY. The generic name is passed to the command processing program so that it can find the object names beginning with the characters in the generic name.
- Date (*DATE). The parameter value is a character string that is passed to the command processing program. The character string uses the format `cyymmdd` (`c` = century digit, `y` = year, `m` = month, `d` = day). The system sets the century digit based on the year specified on the date parameter for the command. If the specified year contained 4 digits, the system sets the century digit to 0 for years that start with 19. The system sets the century digit to 1 for years that start with 20. For years that are specified with 2 digits, the system sets the century digit to 0 if yy equals a number from 40 to 99. However, if yy equals a number from 00 through 39, the system sets the century digit to 1. The user must enter the date on the date parameter of the command in the format that is specified by the date format (DATFMT) job attribute. The date separator (DATSEP) job attribute determines the optional separator character to use for entering the date. Use the Change Job (CHGJOB) command to change the DATFMT and DATSET job attributes . The program reads dates with 2–digit years to be in the range of January 1, 1940, to December 31, 2039. Dates with 4–digit years must be in the range of August 24, 1928, to May 9, 2071.
- Time (*TIME). The parameter value is a character string. The system passes this string to the command processing program in the format `hhmmss` (`h` = hour, `m` = minute, `s` = second). The time separator (TIMSEP) job attribute determines the optional separator to use for entering the time. Use the Change Job (CHGJOB) command to change the TIMSEP job attribute.
- Hexadecimal (*HEX). The parameter value is a hexadecimal value. The characters specified must be 0 through F. The value is passed to the CPP as hexadecimal (EBCDIC) characters (2 hexadecimal digits per byte), and is right adjusted and padded with zeros. If the value is enclosed in apostrophes, an even number of digits is required.
- Zero elements (*ZEROELEM). The parameter value is considered to be a list of zero elements for which no value can be specified in the command. This parameter type is used to prevent a value from being entered for a parameter that is a list even though the command processing program (CPP) expects a value. For example, if two commands use the same CPP, one command could

pass a list for a parameter, and the other command may not have any values to pass. The parameter for the second command would be defined with TYPE(*ZEROELEM).

- Integer (*INT2 or *INT4). The parameter value is an integer that is passed as a 2-byte or 4-byte signed binary number. You can declare binary numbers in a CL procedure or program as variables of TYPE(*INT). You can also use TYPE(*CHAR) and process them with the %BINARY built-in function.
- Unsigned integer (*UINT2 or *UINT4). The parameter value is an integer that is passed as a 2-byte or 4-byte unsigned binary number. You can declare binary numbers in a CL procedure or program as variables of TYPE(*UINT). You can also use TYPE(*CHAR) and process them with the %BINARY built-in function.
- Null (*NULL). The parameter value is a null pointer, which is always passed to the command processing program as a place holder. The only PARM keywords valid for this parameter type are KWD, MIN, and MAX.
- Command string (*CMDSTR). The parameter value is a command. You can use CL variables to specify parameters in the command that are specified in the *CMDSTR parameter. However, you cannot use them to specify the entire *CMDSTR parameter. For example, "SBMJOB CMD(DSPLIB LIB(&LIBVAR))" is valid in a CL Program or procedure, but "SBMJOB CMD(&CMDVAR)" is not.
- Statement label. The statement label identifies the first of a series of QUAL or ELEM statements that further describe the qualified name or the mixed list being defined by this PARM statement.

The following parameter types are for IBM-supplied commands only.

- Expression (*X). The parameter value is a character string, variable name, or numeric value. The value is passed as a numeric value if it contains only digits, a plus or minus sign, and/or a decimal point; otherwise, it is passed as a character string.
- Variable name (*VARNAME). The parameter value is a variable name, which is passed to the command processing program as a character string. The value is left-justified and is padded with blanks. A variable is a name that refers to an actual data value during processing. A variable name can be as long as 10 alphanumeric characters (the first of which must be alphabetic) preceded by an ampersand (&); for example, &PARM. If the name of your variable does not follow the naming convention used on OS/400, you must enclose the name in apostrophes.
- Command (*CMD). The parameter value is a command. For example, the CL command IF has a parameter named THEN whose value must be another command.

Length of Parameter Value

You can also specify a length (LEN parameter) for parameter values of the following types. For parameter types of date or time, date is always 7 characters long and time is always 6 characters long. The following shows the maximum length for each parameter type and the default length for each parameter type for which you can specify a length.

Data Type	Maximum Length	Default Length
*DEC	24 (9 decimal positions)	15 (5 decimal positions)
*LGL	1	1
*CHAR	5000	32
*NAME	256	10

Data Type	Maximum Length	Default Length
*SNAME	256	10
*CNAME	256	10
*GENERIC	256	10
*HEX	256	1
*X	(256 24 9)	(1 15 5)
*VARNAME	11	11
*CMDSTR	20K	256
*PNAME	5000	32

The maximum length that is shown here is the maximum allowed length for these parameter types when the command runs. However, the maximum length that is allowed for character constants in the command definition statements is 32 characters. This restriction applies to the CONSTANT, DFT, VALUES, REL, RANGE, SPCVAL, and SNGVAL parameters. There are specific lengths for input fields available when prompting for a CL command. The input field lengths are 1 through 12 characters and 17, 25, 32, 50, 80, 132, 256, and 512 characters. If a particular parameter has a length that is not allowed,the input field displays with the next larger field length. The prompter displays a 512–character input field for parameters that can be longer than 512 characters.

Default Values

If you are defining an optional parameter, you can define a value on the DFT parameter to be used if the user does not enter the parameter value on the command. This value is called a *default value*. The default value must meet all the value requirements for that parameter (such as type, length, and special values). If you do not specify a default value for an optional parameter, the following default values are used.

Data Type	Default Value
*DEC	0
*INT2	0
*INT4	0
*UINT2	0
*UINT4	0
*LGL	'0'
*CHAR	Blanks
*NAME	Blanks
*SNAME	Blanks
*CNAME	Blanks
*GENERIC	Blanks
*DATE	Zeros ('F0')
*TIME	Zeros ('F0')
*ZEROELEM	0
*HEX	Zeros ('00')
*NULL	Null
*CMDSTR	Blanks
*PNAME	Blanks

Example of Defining a Parameter

The following example defines a parameter OETYPE for a command to call an order entry application.

```
PARM   KWD(OETYPE)  TYPE(*CHAR)  RSTD(*YES) +
       VALUES(DAILY WEEKLY MONTHLY)  MIN(1) +
       PROMPT('Type of order entry:')
```

The OETYPE parameter is required (MIN parameter is 1) and its value is restricted (RSTD parameter is *YES) to the values DAILY, WEEKLY, or MONTHLY. The PROMPT parameter contains the prompt text for the parameter. Since no LEN keyword is specified and TYPE(*CHAR) is defined, a length of 32 is the default.

Data Type and Parameter Restrictions

The following figure shows the valid combinations of parameters according to the parameter type. An X indicates that the combination is valid, a number refers to a restriction noted at the bottom of the table.

	LEN	RTNVAL	CONSTANT	RSTD	DFT	VALUES	REL	RANGE	SPCVAL	SNGVAL
*DEC	X	2	X	X	X	X	X	X	3	1
*LGL	X	2	X	X	X	X			3	1
*CHAR	X	2	X	X	X	X	X	X	3	1
*NAME	X		X	X	X	X	X	X	3	1
*SNAME	X		X	X	X	X	X	X	3	1
*CNAME	X		X	X	X	X	X	X	3	1
*PNAME	X	2	X	X	X	X	X	X	3	1
*GENERIC	X		X	X	X	X	X	X	3	1
*DATE			X	X	X	X	X	X	3	1
*TIME			X	X	X	X	X	X	3	1
*HEX	X		X	X	X	X	X	X	3	1
*ZEROELEM										
*INT2		2	X	X	X	X	X	X	3	1
*INT4		2	X	X	X	X	X	X	3	1
*UINT2		2	X		X		X	X	3	1
*UINT4		2	X		X		X	X	3	1
*CMDSTR	X		X		X					
*NULL	X									
STMT LABEL			X		X					X

Notes:

1. Valid only if the value for MAX is greater than 1. Also, To-values are ignored when CPP is a REXX procedure. Values passed as REXX procedure parameters are the values typed or the defaults for each parameter.
2. Not valid when the command CPP is a REXX procedure.
3. To-values are ignored when CPP is a REXX procedure. Values passed as REXX procedure parameters are the values typed or the default values for each parameter.

	MIN	MAX	ALWUNPRT	ALWVAR	PGM	DTAARA	FILE	FULL	EXPR	VARY
*DEC	X	X		X		X				
*LGL	X	X		X		X	X	1		
*CHAR	X	X	X	X	X	X	X	X	X	1
*NAME	X	X		X	X	X	X	X	X	1

	MIN	MAX	ALWUNPRT	ALWVAR	PGM	DTAARA	FILE	FULL	EXPR	VARY
*SNAME	X	X		X	X	X	X	X	X	1
*CNAME	X	X		X	X	X	X	X	X	1
*PNAME	X	X	X	X	X	X	X	X	X	1
*GENERIC	X	X		X	X	X	X	X	X	1
*DATE	X	X		X		X				
*TIME	X	X		X					X	
*HEX	X	X		X				X	X	
*ZEROELEM	X	X								
*INT2	X	X		X					X	
*INT4	X	X		X					X	
*UINT2	X	X		X					X	
*UINT4	X	X		X					X	
*CMDSTR	2	3		4						1
*NULL	2	3								
STMT LABEL	X	X			X					

Notes:

1. Parameter is ignored when CPP is a REXX procedure.
2. The value for MIN cannot exceed 1 for TYPE(*NULL).
3. The value for MAX cannot exceed 1 for TYPE(*NULL) or TYPE(*CMDSTR).
4. The ALWVAR value is ignored for this type of parameter. CL variables are not allowed when the parameter type is *CMDSTR.

	PASSATR	PASSVAL	CASE	LISTDSPL	DSPINPUT
*DEC	1	X		X	X
*LGL	1	X		X	X
*CHAR	1	X	3	X	X
*NAME	1	X		X	X
*SNAME	1	X		X	X
*CNAME	1	X		X	X
*PNAME	1	X	3	X	X
*GENERIC	1	X		X	X
*DATE	1	X		X	X
*TIME	1	X		X	X
*HEX	1	X		X	X
*ZEROELEM					
*INT2	1	X		X	X
*INT4	1	X		X	X
*UINT2	1	X		X	X
*UINT4	1	X		X	X
*CMDSTR	1			X	X
*NULL					
STMT LABEL		2			

	CHOICE	CHOICEPGM	PMTCTL	PMTCTLPGM	PROMPT	INLPMTLEN
*DEC	X	X	X	X	X	
*LGL	X	X	X	X	X	
*CHAR	X	X	X	X	X	4
*NAME	X	X	X	X	X	4
*SNAME	X	X	X	X	X	4

	CHOICE	CHOICEPGM	PMTCTL	PMTCTLPGM	PROMPT	INLPMTLEN
*CNAME	X	X	X	X	X	4
*PNAME	X	X	X	X	X	4
*GENERIC	X	X	X	X	X	4
*DATE	X	X	X	X	X	
*TIME	X	X	X	X	X	
*HEX	X	X	X	X	X	4
*ZEROELEM						
*INT2	X	X	X	X	X	
*INT4	X	X	X	X	X	
*UINT2	X	X	X	X	X	
*UINT4	X	X	X	X	X	
*CMDSTR	X	X	X	X	X	4
*NULL						
STMT LABEL	X	X	X	X	X	X

Notes:

1. Parameter is ignored when CPP is a REXX procedure.
2. PASSVAL passes a keyword with no blanks or other characters between parentheses when CPP is a REXX procedure.
3. Case (*MIXED) is allowed only with type *CHAR and *PNAME.
4. You can use INLPMTLEN(*PWD) only with types *CHAR, *NAME, *SNAME, *CNAME, and *PNAME.

The next figure shows the valid parameter combinations and restrictions for the PARM, ELEM, and QUAL statements. For example, the intersection of the row for LEN and the column for DFT are blank; therefore, there are no restrictions and combination of LEN(XX) and DFT(XX) is valid. However, the intersection of the row for DFT and the column for CONSTANT contains a 4 which refers to a note at the bottom of the table describing the restriction.

	LEN	RTNVAL	CONSTANT	RSTD	DFT	VALUES	REL	RANGE	SPCVAL	SNGVAL
LEN										
RTNVAL			1	1	1	1	1	1	1	1
CONSTANT		1			4					16
RSTD		1				7	9	9	7	7
DFT		1	4							
VALUES		1		7						
REL		1		9				9		
RANGE		1		9			9			
SPCVAL		1		7						
SNGVAL		1	21	7						
MIN					8					
MAX		2	2							10
ALWUNPRT										
ALWVAR		12								
PGM		1								
DTAARA		1								
FILE		1								
FULL		1								
EXPR		1	5							
VARY		3								

	LEN	RTNVAL	CONSTANT	RSTD	DFT	VALUES	REL	RANGE	SPCVAL	SNGVAL
PASSATR		3								
PASSVAL		13						11		
CASE										
LISTDSPL										
CHOICE			14							
CHOICEPGM										
PMTCTL			15							
PMTCTLPGM			15							
PROMPT			6							
INLPMTLEN		17	17	17						

Notes:

1. The RTNVAL parameter cannot be used with any of the following parameters: CONSTANT, RSTD, DFT, VALUES, REL, RANGE, SPCVAL, SNGVAL, PGM, DTAARA, FILE, FULL, or EXPR. The RTNVAL parameter cannot be used on any command using a REXX procedure as a CPP.
2. A MAX value greater than 1 is not allowed.
3. If RTNVAL(*YES) and PASSATR(*YES) are specified, VARY(*YES) must also be specified. If RTNVAL(*YES) and VARY(*YES) are specified, you must use either *INT2 or *INT4. Combinations of *INT2 and *INT4 are not valid.
4. The CONSTANT and DFT parameters are mutually exclusive.
5. The EXPR(*YES) and CONSTANT parameters are mutually exclusive.
6. The PROMPT parameter is not allowed.
7. If the RSTD parameter is specified, one of the following parameters must also be specified: VALUES, SPCVAL, or SNGVAL.
8. The MIN value must be 0.
9. The REL, RANGE, and RSTD(*YES) parameters are mutually exclusive.
10. Either the MAX value must be greater than 1 or the parameter type must be a statement label, or both.
11. The parameter may not refer to a parameter defined with the parameter PASSVAL(*NULL). A range between parameters is not valid on a PARM statement defined with PASSVAL(*NULL).
12. If RTNVAL(*YES) is specified, ALWVAR(*NO) cannot be specified.
13. PASSVAL(*NULL) is not allowed with RTNVAL(*YES) or a value greater than 0 for MIN.
14. The CHOICE and CONSTANT parameters are mutually exclusive.
15. CONSTANT is mutually exclusive with the PMTCTL and PMTCTLPGM parameters.
16. The CONSTANT parameter cannot be defined on the ELEM/QUAL statement if a SNGVAL parameter is defined on the PARM statement.
17. You cannot use the INLPMTLEN parameter with CONSTANT. You must specify INLPMTLEN(*CALC) or use it as the default if you have specified FULL(*YES), RTNVAL(*YES), or RSTD(*YES).

	MIN	MAX	ALWUNPRT	ALWVAR	PGM	DTAARA	FILE	FULL	EXPR	VARY
LEN										
RTNVAL		2		8	1	1	1	1	1	3
CONSTANT		2							4	
RSTD										

	MIN	MAX	ALWUNPRT	ALWVAR	PGM	DTAARA	FILE	FULL	EXPR	VARY
DFT	5									
VALUES										
REL										
RANGE										
SPCVAL										
SNGVAL		7								
MIN		6								
MAX	6									
ALWUNPRT										
ALWVAR										
PGM						9	9			
DTAARA					9		9			
FILE						9	9			
FULL										
EXPR										
VARY										
PASSATR										3
PASSVAL	10									
CASE										
LISTDSPL										
CHOICE										
CHOICEPGM										
PMTCTL	11									
PMTCTLPGM										
PROMPT										
INLPMTLEN								12		

Notes:

1. The RTNVAL parameter cannot be used with any of the following parameters: CONSTANT, RSTD, DFT, VALUES, REL, RANGE, SPCVAL, SNGVAL, PGM, DTAARA, FILE, FULL, or EXPR. The RTNVAL parameter cannot be used on any command using a REXX procedure as a CPP.
2. A MAX value greater than 1 is not allowed.
3. If RTNVAL(*YES) and PASSATR(*YES) are specified, VARY(*YES) must also be specified. If RTNVAL(*YES) and VARY(*YES) are specified, you must use either *INT2 or *INT4. Combinations of *INT2 and *INT4 are not valid.
4. The EXPR(*YES) and CONSTANT parameters are mutually exclusive.
5. The MIN value must be 0.
6. The value specified for the MIN parameter must not exceed the value specified for the MAX parameter.
7. Either the MAX value must be greater than 1 or the parameter type must be a statement label, or both.
8. If RTNVAL(*YES) is specified, ALWVAR(*NO) cannot be specified.
9. PGM(*YES), DTAARA(*YES), and a value other than *NO for the FILE parameters are mutually exclusive.
10. PASSVAL(*NULL) is not allowed with RTNVAL(*YES) or a value greater than 0 for MIN.
11. PMTCTL is not allowed with a value greater than 0 for MIN.

12. You must specify INLPMTLEN(*CALC) or use it as the default if you specified FULL(*YES), RTNVAL(*YES), or RSTD(*YES).

	PASSATR	PASSVAL	CASE	LISTDSPL	DSPINPUT
LEN					
RTNVAL	1	4			
CONSTANT				9	5
RSTD					
DFT					
VALUES					
REL					
RANGE		3			
SPCVAL					
SNGVAL					
MIN		4			
MAX					
ALWUNPRT					
ALWVAR					
PGM					
DTAARA					
FILE					
FULL					
EXPR					
VARY	1				
PASSATR					
PASSVAL					
CASE			10		
LISTDSPL				11	
CHOICE					
CHOICEPGM					
PMTCTL					
PMTCTLPGM					
PROMPT					
INLPMTLEN					

	CHOICE	CHOICEPGM	PMTCTL	PMTCTLPGM	PROMPT	INLPMTLEN
LEN						
RTNVAL						12
CONSTANT			7	7	2	12
RSTD						12
DFT						
VALUES						
REL						
RANGE						
SPCVAL						
SNGVAL						
MIN			8			
MAX						
ALWUNPRT						
ALWVAR						
PGM						
DTAARA						
FILE						

	CHOICE	CHOICEPGM	PMTCTL	PMTCTLPGM	PROMPT	INLPMTLEN
FULL						12
EXPR						
VARY						
PASSATR						
PASSVAL						
CASE						
LISTDSPL						
CHOICE		6				
CHOICEPGM	6					
PMTCTL						
PMTCTLPGM						
PROMPT						
INLPMTLEN						

Notes:

1. If RTNVAL(*YES) and PASSATR(*YES) are specified, VARY(*YES) must also be specified. If RTNVAL(*YES) and VARY(*YES) are specified, you must use either *INT2 or *INT4. Combinations of *INT2 and *INT4 are not valid.
2. The PROMPT parameter is not allowed.
3. The parameter may not refer to a parameter defined with the parameter PASSVAL(*NULL). A range between parameters is not valid on a PARM statement defined with PASSVAL(*NULL).
4. PASSVAL(*NULL) is not allowed with RTNVAL(*YES) or a value greater than 0 for MIN.
5. The CHOICE and CONSTANT parameters are mutually exclusive.
6. CHOICE(*PGM) requires a name for CHOICEPGM.
7. CONSTANT is mutually exclusive with the PMTCTL and PMTCTLPGM parameters.
8. PMTCTL is not allowed with a value greater than 0 for MIN.
9. CONSTANT is mutually exclusive with DSPINPUT(*NO) and DSPINPUT(*PROMPT).
10. The CASE parameter is valid only on PARM and ELEM statements. CASE is not valid on the QUAL statement.
11. The LISTDSPL parameter is valid only on the PARM statement.
12. You cannot use the INLPMTLEN parameter with CONSTANT. You must specify INLPMTLEN(*CALC) or use it as the default if you specified FULL(*YES), RTNVAL(*YES), or RSTD(*YES).

Defining Lists for Parameters

You can define a parameter to accept a list of values instead of just a single value. You can define the following types of lists:

- A simple list, which allows one or more values of the same type to be specified for a parameter
- A mixed list, which allows a set of separately defined values to be specified for a parameter
- A list within a list, which allows a list to be specified more than once for a parameter or which allows a list to be specified for a value within a mixed list

The following sample command source illustrates the different types of lists:

```
             CMD        PROMPT('Example of lists command')


   /* THE FOLLOWING PARAMETER IS A SIMPLE LIST. IT WILL ACCEPT UP TO   */
   /* 5 NAMES.                                                         */
             PARM       KWD(SIMPLST) TYPE(*NAME) LEN(10) DFT(*ALL) +
                         SPCVAL((*ALL)) MAX(5) PROMPT('Simple list +
                         of up to 5 names')


   /* THE FOLLOWING PARAMETER IS A MIXED LIST OF 3 VALUES, EACH OF A   */
   /* DIFFERENT TYPE AND/OR LENGTH. EACH ELEMENT MAY NOT BE REPEATED.  */
             PARM       KWD(MXDLST) TYPE(MLSPEC) PROMPT('This is a +
                  mixed list of 3 val')
MLSPEC:      ELEM       TYPE(*CHAR) LEN(4) PROMPT('Elem 1 of 3')
             ELEM       TYPE(*DEC) LEN(3 0) PROMPT('Second of three')
             ELEM       TYPE(*CHAR) LEN(10) PROMPT('Last of three +
                          elements')


/* THE FOLLOWING PARAMETER IS A LIST WITHIN A LIST. IT CONTAINS A   */
/* LIST OF UP TO 2 ELEMENTS, WHICH MAY BE REPEATED UP TO 3 TIMES.   */
             PARM       KWD(LWITHINL1) TYPE(LWLSPECA) MAX(3) +
                          PROMPT('Repeatable list of 2 elements')
LWLSPECA:    ELEM         TYPE(*CHAR) LEN(10) PROMPT('1st part of +
                            repeatable list')
             ELEM         TYPE(*DEC) LEN(5 0) PROMPT('2nd part of +
                            repeatable list')


/* THE FOLLOWING PARAMETER IS A LIST WITHIN A LIST. IT CONTAINS A   */
/* LIST OF UP TO 2 ELEMENTS, THE FIRST OF WHICH MAY BE REPEATED     */
/* UP TO 3 TIMES.                                                   */
             PARM       KWD(LWITHINL2) TYPE(LWLSPECB) MAX(1) +
                          PROMPT('Repeated simple within mixed')
LWLSPECB:    ELEM       TYPE(*CHAR) LEN(10) MAX(3) PROMPT('Simple +
                          list within a list')
             ELEM       TYPE(*DEC) LEN(5 0) PROMPT('Single parm +
                          within a list')
```

The following display shows the prompt for the preceding sample command:

```
                        Example of lists command (LSTEXAMPLE)

 Type choices, press Enter.

 Simple list of up to 5 names . . SIMPLST        *ALL
                           + for more values
 This is a mixed list of 3 val    MXDLST
   Elem 1 of 3  . . . . . . . . .
   Second of three  . . . . . . .
   Last of three elements . . . .
 Repeatable list of 2 elements    LWITHINL1
   1st part of repeatable list  .
   2nd part of repeatable list  .
                           + for more values
 Repeatable simple within mixed   LWITHINL2
   Simple list within a list  . .
                           + for more values
   Single parm within a list  . .


                                                                       Bottom
F3=Exit   F4=List   F5=Refresh   F12=Cancel    F13=Prompter help
F24=More keys
```

Defining a Simple List

A simple list can accept one or more values of the type specified by the parameter. For example, if the parameter is for the user name, a simple list means that more than one user name can be specified on that parameter.

```
USER(JONES SMITH MILLER)
```

If a parameter's value is a simple list, you specify the maximum number of elements the list can accept using the MAX parameter on the PARM statement. For a simple list, no command definition statements other than the PARM statement need be specified.

The following example defines a parameter USER for which the display station user can specify up to five user names (a simple list).

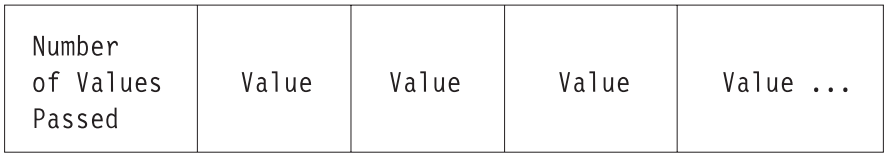
```
PARM     KWD(USER) TYPE(*NAME) LEN(10) MIN(0) MAX(5) +
         SPCVAL(*ALL) DFT(*ALL)
```

The parameter is an optional parameter as specified by MIN(0) and the default value is *ALL as specified by DFT(*ALL).

When the elements in a simple list are passed to the command processing program, the format varies depending on whether you are using CL or HLL, or REXX. The following section describes how the elements used in the previous example are passed using CL and HLL. For an explanation of the differences when using REXX, see "Using REXX for Simple Lists" on page 310.

Using CL or HLL for Simple Lists

When the command is run using CL or HLL, the elements in a simple list are passed to the command processing program in the following format.



RBAFN509-0

The number of values passed is specified by a binary value that is two characters long. This number indicates how many values were actually entered (are being passed), not how many can be specified. The values are passed by the type of parameter just as a value of a single parameter is passed (as described under "Defining Parameters" on page 294). For example, if two user names (BJONES and TBROWN) are specified for the USER parameter, the following is passed.

RBAFN510-0

The user names are passed as 10-character values that are left-adjusted and padded with blanks.

When a simple list is passed, only the number of elements specified on the command are passed. The storage immediately following the last element passed is not part of the list and must not be referred to as part of the list. Therefore, when the command processing program (CPP) processes a simple list, it uses the number of elements passed to determine how many elements can be processed.

Figure 13 on page 309 shows an example of a CL procedure using the binary built-in function to process a simple list.

```
PGM PARM (...&USER..)
.
.
.
/* Declare space for a simple list of up to five */
/* 10-character values to be received            */
DCL VAR(&USER) TYPE(*CHAR) LEN(52)
.
DCL VAR(&CT)    TYPE(*DEC)  LEN(3 0)
DCL VAR(&USER1) TYPE(*CHAR) LEN(10)
DCL VAR(&USER2) TYPE(*CHAR) LEN(10)
DCL VAR(&USER3) TYPE(*CHAR) LEN(10)
DCL VAR(&USER4) TYPE(*CHAR) LEN(10)
DCL VAR(&USER5) TYPE(*CHAR) LEN(10)
.
.
.
CHGVAR  VAR(&CT) VALUE(%BINARY(&USER 1 2))
.
IF (&CT > 0) THEN(CHGVAR &USER1 %SST(&USER 3 10))
IF (&CT > 1) THEN(CHGVAR &USER2 %SST(&USER 13 10))
IF (&CT > 2) THEN(CHGVAR &USER3 %SST(&USER 23 10))
IF (&CT > 3) THEN(CHGVAR &USER4 %SST(&USER 33 10))
IF (&CT > 4) THEN(CHGVAR &USER5 %SST(&USER 43 10))
IF (&CT > 5) THEN(DO)
/*  If CT is greater than 5, the values passed     */
/*  is greater than the program expects, and error */
/*  logic should be performed                      */
.
.
.
ENDDO
ELSE DO
/* The correct number of values are passed */
/* and the program can continue processing */
.
.
.
ENDDO
ENDPGM
```

Figure 13. Simple List Example

This same technique can be used to process other lists in a CL procedure or program.

For a simple list, a single value such as *ALL or *NONE can be entered on the command instead of the list. Single values are passed as an individual value. Similarly, if no values are specified for a parameter, the default value, if any is defined, is passed as the only value in the list. For example, if the default value *ALL is used for the USER parameter, the following is passed.

RBAFN511-0

*ALL is passed as a 10-character value that is left-adjusted and padded with blanks.

If no default value is defined for an optional simple list parameter, the following is passed:

RBAFN512-0

Using REXX for Simple Lists

When the same command is run, the elements in a simple list are passed to the REXX procedure in the argument string in the following format:

```
. . . USER(value1 value2  . . . valueN) . . .
```

where `valueN` is the last value in the simple list.

For example, if two user names (BJONES and TBROWN) are specified for the USER parameter, the following is passed:

```
. . . USER(BJONES TBROWN) . . .
```

When a simple list is passed, only the number of elements specified on the command are passed. Therefore, when the CPP processes a simple list, it uses the number of elements passed to determine how many elements can be processed.

The REXX example in Figure 14 produces the same result as the CL procedure in Figure 13 on page 309:

```
.
.
.
PARSE ARG . 'USER(' user ')' .
.
.
CT = WORDS(user)
IF CT > 0 THEN user1 = WORD(user,1) else user1 = '
IF CT > 1 THEN user2 = WORD(user,2) else user2 = '
IF CT > 2 THEN user3 = WORD(user,3) else user3 = '
IF CT > 3 THEN user4 = WORD(user,4) else user4 = '
IF CT > 4 THEN user5 = WORD(user,5) else user5 = '
IF CT > 5 THEN
  DO
    /* If CT is greater than 5, the values passed
       is greater than the program expects, and error
       logic should be performed */
.
.
.
  END
ELSE
  DO
    /* The correct number of values are passed
       and the program can continue processing */
  END
EXIT
```

Figure 14. REXX Simple List Example

This same procedure can be used to process other lists in a REXX program.

For a simple list, a single value such as *ALL or *NONE can be entered on the command instead of the list. Single values are passed as an individual value.

Similarly, if no values are specified for a parameter, the default value, if any is defined, is passed as the only value in the list. For example, if the default value *ALL is used for the USER parameter, the following is passed:

```
. . . USER(*ALL) . . .
```

If no default value is defined for an optional simple list parameter, the following is passed:

```
. . . USER() . . .
```

For more information about REXX procedures, see the REXX/400 Programmer's Guide and the REXX/400 Reference.

Defining a Mixed List

A mixed list accepts a set of separately defined values that usually have different meanings, are of different types, and are in a fixed position in the list. For example, `LOG(4 0 *SECLVL)` could specify a mixed list. The first value, `4`, identifies the message level to be logged; the second value, `0`, is the lowest message severity to be logged. The third value, `*SECLVL`, specifies the amount of information to be logged (both first- and second-level messages). If a parameter's value is a mixed list, the elements of the list must be defined separately using an Element (ELEM) statement for each element.

The TYPE parameter on the associated PARM statement must have a label that refers to the first ELEM statement for the list.

```
         PARM    KWD(LOG)    TYPE(LOGLST) ...

LOGLST:  ELEM    TYPE(*INT2)      ...
         ELEM    TYPE(*INT2)      ...
         ELEM    TYPE(*CHAR)      LEN(7)
```

The first ELEM statement is the only ELEM statement that can have a label. Specify the ELEM statements in the order in which the elements occur in the list.

Note that when the MAX parameter has a value greater than 1 on the PARM statement, and the TYPE parameter refers to ELEM statements, the parameter being defined is a list within a list.

Parameters that you can specify on the ELEM statement include TYPE, LEN, CONSTANT, RSTD, DFT, VALUES, REL, RANGE, SPCVAL, SNGVAL, MIN, MAX, ALWUNPRT, ALWVAR, PGM, DTAARA, FILE, FULL, EXPR, VARY, PASSATR, CHOICE, CHOICEPGM, and PROMPT.

In the following example, a parameter CMPVAL is defined for which the display station user can specify a comparison value and a starting position for the comparison (a mixed list).

```
      PARM   KWD(CMPVAL) TYPE(CMP) SNGVAL(*ANY) DFT(*ANY) +
               MIN(0)
CMP:  ELEM   TYPE(*CHAR) LEN(80) MIN(1)
      ELEM   TYPE(*DEC) LEN(2 0) RANGE(1 80) DFT(1)
```

When the elements in a mixed list are passed to the command processing program, the format varies depending on whether you are using CL or HLL, or REXX. The following section describes how the elements used in the previous example are passed using CL and HLL. For an explanation of the differences when using REXX, see "Using REXX for Mixed Lists" on page 313.

Using CL or HLL for Mixed Lists

When a command is run using CL or HLL, the elements in a mixed list are passed to the command processing program in the following format:

Number of Values in the Mixed List	Value of Element 1	Value of Element 2	. . .	Value of Element n

RBAFN513-0

The number of values in the mixed list is passed as a binary value of length 2. This value always indicates how many values have been defined for the mixed list, not how many were actually entered on the command. This value may be 1 if the SNGVAL parameter is entered or is passed as the default value. If the user does not enter a value for an element, a default value is passed. The elements are passed by their types just as single parameter values are passed (as described under "Defining Parameters" on page 294). For example, if, in the previous example the user enters a comparison value of QCMDI for the CMPVAL parameter, but does not enter a value for the starting position, whose default value is 1, the following is passed.

0002	QCMDI	1

RBAFN514-0

The data QCMDI is passed as an 80-character value that is left-adjusted and padded with blanks. The number of elements is sent as a binary value of length 2.

When the display station user enters a single value or when a single value is the default for a mixed list, the value is passed as the first element in the list. For example, if the display station user enters *ANY as a single value for the parameter, the following is passed.

0001	*ANY

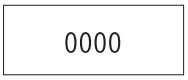
RBAFN515-0

*ANY is passed as an 80-character value that is left-adjusted and padded with blanks.

Mixed lists can be processed in CL programs. Unlike simple lists, the binary value does not need to be tested to determine how many values are in the list because this value is always the same for a given mixed list unless the SNGVAL parameter was passed to the command processing program. In this case, the value is 1. If the command is entered with a single value for the parameter, only that one value is passed. To process the mixed list in a CL procedure, you must use the substring built-in function (see Chapter 2).

In one case, only a binary value of 0000 is passed as the number of values for a mixed list. If no default value is defined on the PARM statement for an optional parameter and the first value of the list is required (MIN(1)), then the parameter itself is not required; but if any element is specified the first element is required. In this case, if the command is entered without specifying a value for the parameter,

the following is passed.



RBAFN512-0

An example of such a parameter is:

```
      PARM  KWD(KWD1)    TYPE(E1) MIN(0)
E1:   ELEM  TYPE(*CHAR)  LEN(10)  MIN(1)
      ELEM  TYPE(*CHAR)  LEN(2)   MIN(0)
```

If this parameter were to be processed by a CL procedure, the parameter value could be received into a 14-character CL variable. The first 2 characters could be compared to either of the following:

- a 2-character variable initialized to hex 0000 using the %SUBSTRING function.
- a decimal 0 using the %BINARY built-in function.

Using REXX for Mixed Lists

When a command is run using REXX, the elements in a mixed list are passed to the command processing program in the following format:

```
. . . CMPVAL(value1 value2  . . . valueN) . . .
```

where `valueN` is the last value in the mixed list.

If the user does not enter a value for an element, a default value is passed. For example, if in the previous example, the user enters a comparison value of QCMDI for the CMPVAL parameter, but does not enter a value for the starting position, whose default value is 1, the following is passed:

```
. . . CMPVAL(QCMDI 1)  . . .
```

Note that trailing blanks are not passed with REXX values.

When a display station user enters a single value or when a single value is the default for a mixed list, the value is passed as the first element in the list. For example, if the display station user enters *ANY as a single value for the parameter, the following is passed:

```
. . . CMPVAL(*ANY) . . .
```

Again note that trailing blanks are not passed with REXX values.

If no default value is defined on the PARM statement for an optional parameter and the first value of the list is required (MIN(1)), then the parameter itself is not required. But if any element is specified, the first element is required. In this case, if the command is entered without specifying a value for the parameter, the following is passed:

```
. . . CMPVAL() . . .
```

Defining Lists within Lists

A list within a list can be:

- A list that can be specified more than once for a parameter (simple or mixed list)
- A list that can be specified for a value within a mixed list

The following is an example of lists within a list.

```
STMT((START RESPND) (ADDDSP CONFRM))
```

The outside set of parentheses enclose the list that can be specified for the parameter (the outer list) while each set of inner parentheses encloses a list within a list (an inner list).

In the following example, a mixed list is defined within a simple list. A mixed list is specified, and the MAX value on the PARM statement is greater than 1; therefore, the mixed list can be specified up to the number of times specified on the MAX parameter.

```
        PARM KWD(PARM1) TYPE(LIST1) MAX(5)
LIST1:  ELEM TYPE(*CHAR) LEN(10)
        ELEM TYPE(*DEC) LEN(3 0)
```

In this example, the two elements can be specified up to five times. When a value is entered for this parameter, it could appear as follows:

```
PARM1((VAL1 1.0) (VAR2 2.0) (VAR3 3.0))
```

In the following example, a simple list is specified as a value in a mixed list. In this example, the MAX value on the ELEM statement is greater than 1; therefore, the element can be repeated up to the number of times specified on the MAX parameter.

```
        PARM KWD(PARM2) TYPE(LIST2)
LIST2:  ELEM TYPE(*CHAR) LEN(10) MAX(5)
        ELEM TYPE(*DEC) LEN(3 0)
```

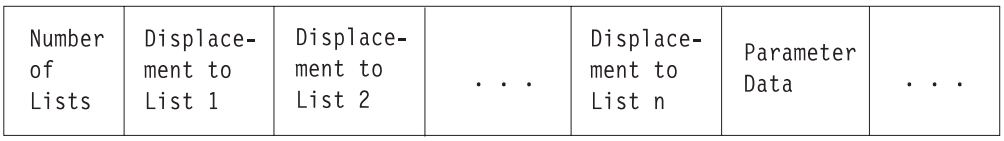
In this example, the first element can be specified up to five times, but the second element can be specified only once. When a value is entered for this parameter, it could appear as follows.

```
PARM2((NAME1 NAME2 NAME3) 123.0)
```

When lists within lists are passed to the command processing program, the format varies depending on whether you are using CL or HLL, or REXX. The following section describes how elements are passed using CL and HLL. For an explanation of the differences when using REXX, see "Using REXX for Lists within Lists" on page 316.

Using CL or HLL for Lists within Lists

When a command is run using CL or HLL, a list within a list is passed to the command processing program in the following format:



RBAFN534-0

The number of lists is passed as a binary value of length 2. Following the number of lists, the displacement to the lists is passed (not the values that were entered in the lists). Each displacement is passed as a binary value of length 2 or length 4 depending on the value of the LISTDSPL parameter.

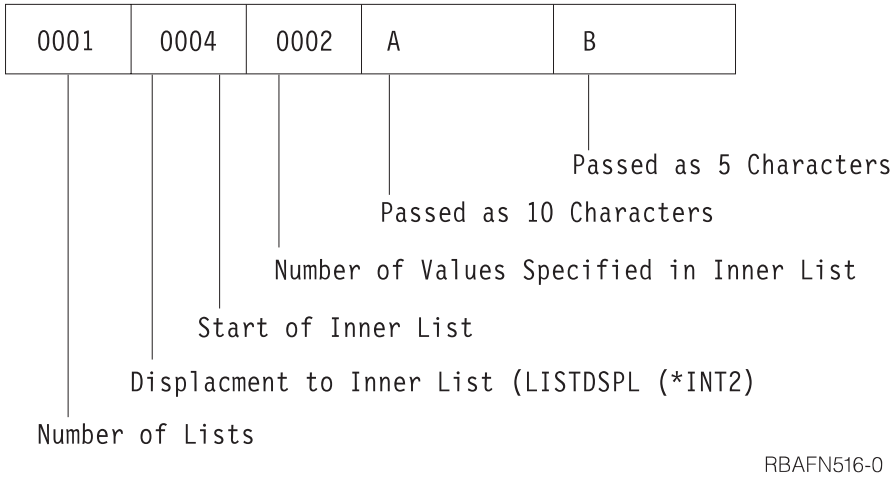
The following example shows a definition for a parameter KWD2 (which is a mixed list within a simple list) how the parameter can be specified by the display station user, and what is passed. The parameter definition is:

```
      PARM    KWD(KWD2)    TYPE(LIST) MAX(20) MIN(0) +
              DFT(*NONE)  SNGVAL(*NONE)  LISTDSPL(*INT2)
LIST: ELEM    TYPE(*CHAR)  LEN(10) MIN(1)       /*From value*/
      ELEM    TYPE(*CHAR)  LEN(5) MIN(0)        /*To value*/
```

The display station user enters the KWD2 parameter as:

```
KWD2((A B))
```

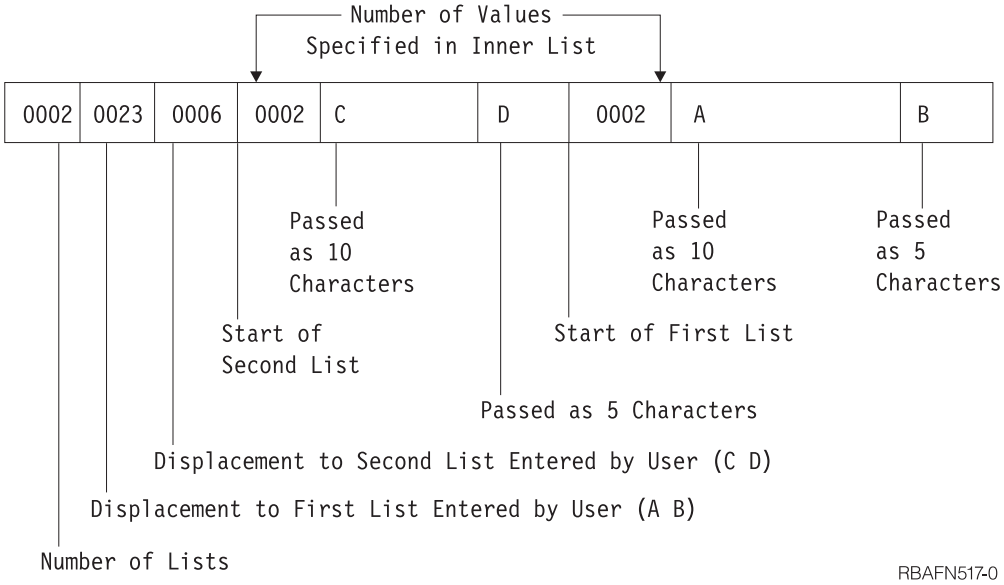
The following is passed to the command processing program:



If the display station user enters the following instead:

```
KWD2((A B) (C D))
```

the following is passed to the command processing program:



Lists within a list are passed to the command processing program in the order n (the last one entered by the display station user) to 1 (the first one entered by the display station user). The displacements, however, are passed from 1 to n.

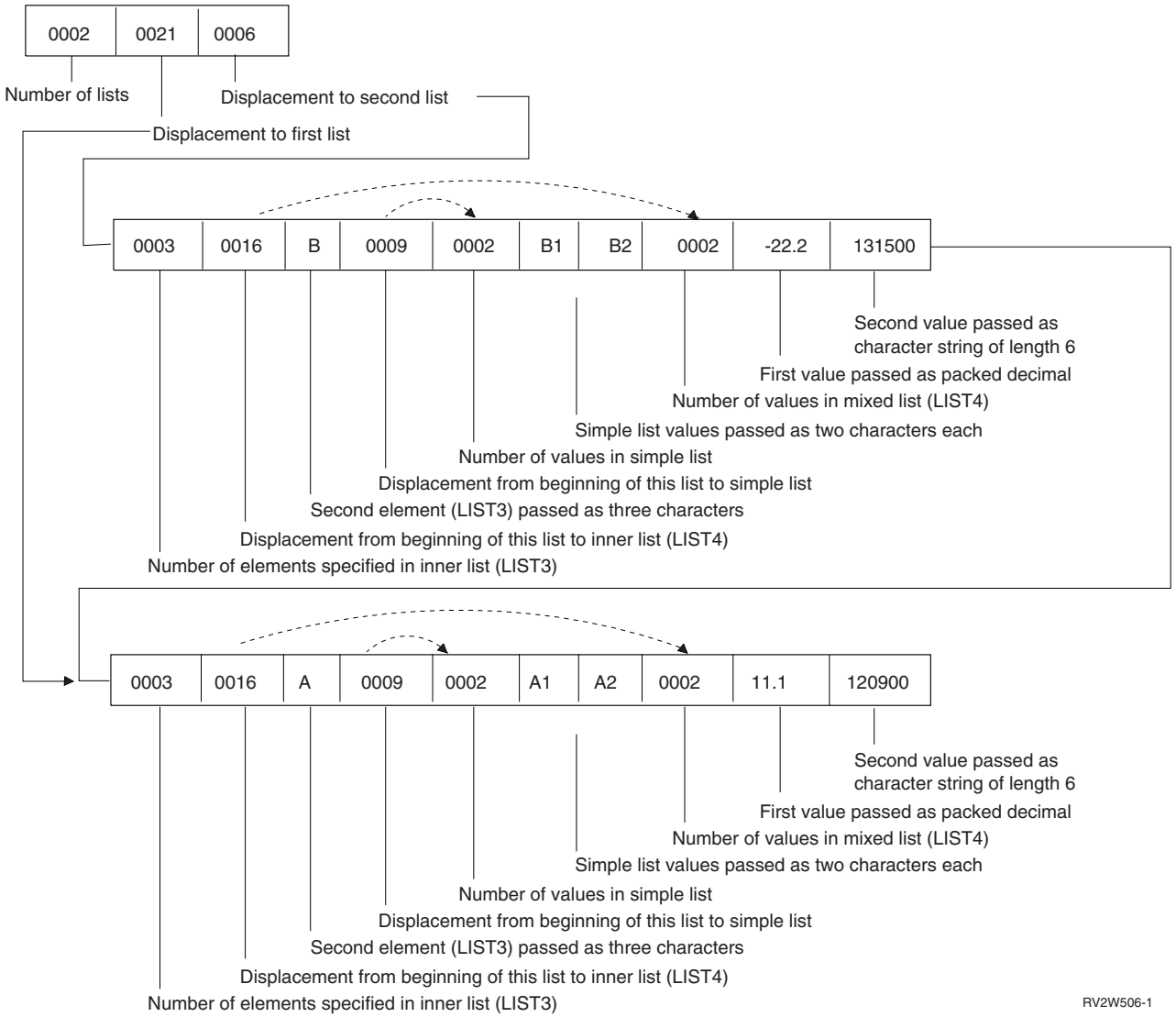
The following is a more complex example of lists within lists. The parameter definition is:

```
        PARM    KWD(PARM1)  TYPE(LIST3)  MAX(25)
LIST3:  ELEM    TYPE(LIST4)
        ELEM    TYPE(*CHAR)  LEN(3)
        ELEM    TYPE(*NAME)  LEN(2)  MAX(5)
LIST4:  ELEM    TYPE(*DEC) LEN(7 2)
        ELEM    TYPE(*TIME)
```

If the display station user enters the PARM1 parameter as:

```
PARM1(((11.1  120900)  A  (A1  A2))  ((-22.2  131500)  B  (B1  B2)))
```

the following is passed to the command processing program:



Using REXX for Lists within Lists

When a command is run using REXX, a list within a list is passed to the command processing program just as the values are entered for the parameters. Trailing blanks are not passed.

The following example shows a definition for a parameter KWD2, which is a mixed list within a simple list, how the parameter can be specified by the display station user, and what is passed. The parameter definition is:

```
       PARM    KWD(KWD2)    TYPE(LIST) MAX(20) MIN(0) +
               DFT(*NONE)  SNGVAL(*NONE)
LIST:  ELEM    TYPE(*CHAR)  LEN(10) MIN(1)       /*From value*/
       ELEM    TYPE(*CHAR)  LEN(5) MIN(0)        /*To value*/
```

The display station user enters the KWD2 parameter as:

```
KWD2((A B))
```

The following is passed to the command processing program:

```
KWD2(A B)
```

If the display station user enters the following instead:

```
KWD2((A B) (C D))
```

The following is passed to the command processing program:

```
KWD2((A B) (C D))
```

The following is a more complex example of lists within lists. The parameter definition is:

```
       PARM    KWD(PARM1)  TYPE(LIST3)  MAX(25)
LIST3: ELEM    TYPE(LIST4)
       ELEM    TYPE(*CHAR)  LEN(3)
       ELEM    TYPE(*NAME)  LEN(2)  MAX(5)
LIST4: ELEM    TYPE(*DEC) LEN(7 2)
       ELEM    TYPE(*TIME)
```

The display station user enters the PARM1 parameter as:

```
PARM1(((11.1  12D900)  A  (A1  A2))  ((-22.2  131500)  B  (B1  B2)))
```

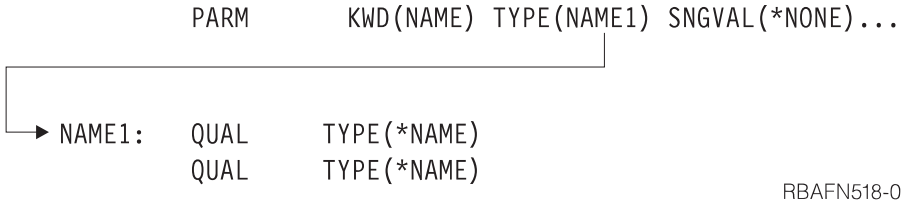
The following is passed to the command processing program:

```
PARM1(((11.1  12D900)  A  (A1  A2))  ((-22.2  131500)  B  (B1  B2)))
```

Defining a Qualified Name

A qualified name is the name of an object preceded by the name of the library in which the object is stored. If a parameter value or list item is a qualified name, you must define the name separately using Qualifier (QUAL) statements. Each part of the qualified name must be defined with a QUAL statement. The parts of a qualified name must be described in the order in which they occur in the qualified name. You must specify *NAME or *GENERIC in the first QUAL statement. The associated PARM or ELEM statement must identify the label that refers to the first QUAL statement for the qualified name.

The following command definition statements define the most common qualified name. A qualified object consists of the library name which contains an object followed by the name of the object itself. The QUAL statements must appear in the order in which they are to occur in the qualified name.



RBAFN518-0

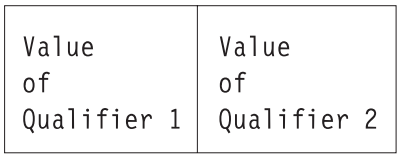
Many of the parameters that can be specified for the QUAL statement are the same as those described for the PARM statement (see “Defining Parameters” on page 294). However, only the following values can be specified for the TYPE parameter:

- *NAME
- *GENERIC
- *CHAR
- *INT2
- *INT4

When a qualified name is passed to the command processing program, the format varies depending on whether you are using CL or HLL, or REXX. The following section describes how qualified names are passed using CL and HLL. For an explanation of the differences when using REXX, see "Using REXX for a Qualified Name" on page 319.

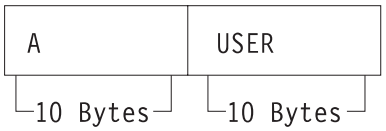
Using CL or HLL for a Qualified Name

A qualified name is passed to the command processing program in the following format when using CL or HLL:



RBAFN519-0

For example, if the display station user enters `NAME(USER/A)` for the previously defined QUAL statements, the name is passed to the command processing program as follows:



RBAFN520-0

Qualifiers are passed to the command processing program consecutively by their types and length just as single parameter values are passed (as described under "Defining Parameters" on page 294). The separator characters (/) are not passed. This applies regardless of whether a single parameter, an element of a mixed list, or a simple list of qualified names is passed.

If the display station user enters a single value for a qualified name, the length of the value passed is the total of the length of the parts of the qualified name. For example, if you define a qualified name with two values each of length 10, and if the display station user enters a single value, the single value passed is left-adjusted and padded to the right with blanks so that a total of 20 characters is passed. If the display station user enters *NONE as the single value, the following 20-character value is passed:

RBAFN521-0

Qualified names can be processed in CL programs using the Substring built-in function as shown in the following example.

The substring built-in function (%SUBSTRING or %SST) is used to separate the qualified name into two values.

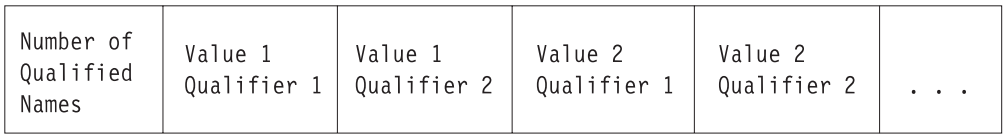
```
PGM PARM(&QLFDNAM)
DCL &QLFDNAM TYPE(*CHAR) LEN(20)
DCL &OBJ TYPE(*CHAR) LEN(10)
DCL &LIB TYPE(*CHAR) LEN(10)
CHGVAR &OBJ %SUBSTRING(&QLFDNAM 1 10) /* First 10 */
CHGVAR &LIB %SST(&QLFDNAM 11 10) /* Second 10 */
 .
 .
 .
ENDPGM
```

You can then specify the qualified name in the proper CL syntax. For example, `OBJ(&LIB/&OBJ)`.

You can also separate the qualified name into two values using the following method:

```
PGM PARM(&QLFDNAM)
DCL &QLFDNAM TYPE(*CHAR) LEN(20)
CHKOBJ (%SST(&QLFDNAM 11 10)/%SST(&QLFDNAM 1 10)) *PGM
 .
 .
 .
ENDPGM
```

A simple list of qualified names is passed to the command processing program in the following format:



RBAFN522-0

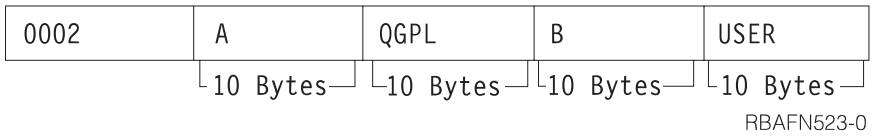
For example, assume that MAX(3) were added as follows to the PARM statement for the NAME parameter.

```
        PARM   KWD(NAME) TYPE(NAME1) SNGVAL(*NONE) MAX(3)
NAME1:  QUAL   TYPE(*NAME)
        QUAL   TYPE(*NAME)
```

If the display station user enters the following:

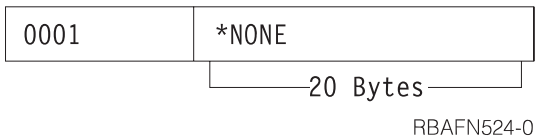
```
NAME(QGPL/A USER/B)
```

then the name parameter would be passed to the command processing program as follows.



RBAFN523-0

If the display station user enters the single value NAME(*NONE), the name parameter is passed as follows.



RBAFN524-0

Using REXX for a Qualified Name

When a command is run using REXX, a qualified name is passed to the command processing program just as the value is entered for the parameter. Trailing blanks are not passed.

For example, if a display station user enters the following for the QUAL statements defined previously in this section:

```
NAME(USER/A)
```

the qualified name is passed to the command processing program in the following format:

```
NAME(USER/A)
```

Qualifiers are passed to the command processing program consecutively by their types and length just as single parameter values are passed (as described under "Defining Parameters" on page 294).

If the display station user enters *NONE as the single value, the following 20-character value is passed:

```
NAME(*NONE)
```

The following example shows how a display station user would enter a simple list of qualified names:

```
NAME(QGPL/A USER/B)
```

Using REXX, the name parameter would be passed to the command processing program as the following:

```
NAME(QGPL/A USER/B)
```

Defining a Dependent Relationship

If a required relationship exists between parameters, and if parameter values must be checked when the command is run, use the Dependent (DEP) statement to define that relationship. Using the DEP statement, you can perform the functions that are listed below:

- Specify the controlling conditions that must be true before the parameter relationships defined in the PARM parameter need to be true (CTL).
- Specify the parameter relationships that require testing if the controlling conditions defined by CTL are true (PARM).
- Specify the number of parameter relationships that are defined on the associated PARM statement that must be true if the control condition is true (NBRTRUE).
- Specify the message identifier of an error message in a message file that the system is to send to the display station user if the parameter dependencies have not been satisfied.

In the following example, if the display station user specifies the TYPE(LIST) parameter, the display station user must also specify the ELEMLIST parameter.

```
DEP CTL(&TYPE *EQ LIST) PARM(ELEMLIST)
```

In the following example, the parameter &WRITER must never be equal to the parameter &NEWWTR. If this condition is not true, message USR0001 is issued to the display station user.

```
DEP CTL(*ALWAYS) PARM((&WRITER *NE &NEWWTR)) MSGID(USR0001)
```

In the following example, if the display station user specifies the FILE parameter, the display station user must also specify both the VOL and LABEL parameters.

```
DEP CTL(FILE) PARM(VOL LABEL) NBRTRUE(*EQ 2)
```

Possible Choices and Values

The prompter will display possible choices for parameters to the right of the input field on the prompt displays. The text to be displayed can be created automatically, specified in the command definition source, or created dynamically by an exit program. Text describing possible choices can be defined for any PARM, ELEM, or

QUAL statement, but because of limitations in the display format, the text is displayed only for values with a field length of 12 or less, 10 or less for all but the first qualifier in a group.

The text for possible choices is defined by the CHOICE parameter. The default for this parameter is *VALUES, which indicates that the text is to be created automatically from the values specified for the TYPE, RANGE, VALUES, SPCVAL, and SNGVAL keywords. The text is limited to 30 characters; if there are more values than can fit in this size, an ellipsis (...) is added to the end of the text to indicate that it is incomplete.

You can specify that no possible choices should be displayed (*NONE), or you can specify either a text string to be displayed or the ID of a text message which is retrieved from the message file specified in the PMTFILE parameter of the CRTCMD command.

You can also specify that an exit program to run during prompting to provide the possible choices text. This could be done if, for example, you want to show the user a list of objects that currently exist on the system. The same exit program can be used to provide the list of permissible values shown on the Specify Value for Parameter display. To specify an exit program, specify *PGM for the CHOICE parameter, and the qualified name of the exit program in the CHOICEPGM parameter on the PARM, ELEM, or QUAL statement.

The exit program must accept the following two parameters:

- **Parameter 1**: A 21-byte field that is passed by the prompter to the choice program, and contains the following:

  Positions
  Descriptions

  1-10 Command name. Specifies the name of the command being processed that causes the program to run.

  11-20 Keyword name. Specifies the keyword for which possible choices or permissible values are being requested.

  21 C or P character indicating the type of data being requested by prompter. The letter C indicates that this is a 30-byte field into which the text for possible choices is to be returned. The letter P indicates that this a 2000-byte field into which a permissible values list is to be returned.

- **Parameter 2**: A 30- or 2000-byte field for returning one of the following:
  - If C is in byte 21 of the first parameter, this indicates that the text for possible choices will return. Additionally, this is a 30-byte field where the program places the text to the right of the input field on the prompt display.
  - If P is in byte 21 of the first parameter (indicating that a permissible values list is to be returned), this is a 2000-byte field into which the program is to place the list. The first two bytes of the list must contain the number of entries (in binary) in the list. This value is followed by entries that consist of a 2-byte binary length followed by the value, which must be 1 to 34 bytes long.

    If a binary zero value is returned in the first two bytes, no permissible values are displayed.

    If a binary negative value is returned in the first two bytes, the list of permissible values is taken from the command.

If any exception occurs when the program is called, the possible choices text is left blank, and the list of permissible values is taken from the command.

Using Prompt Control

You can control which parameters are displayed for a command during prompting by using prompt control specifications. This control can simplify prompting for a command by displaying only the parameters that you want to see.

You can specify that a parameter be displayed depending on the value specified for other parameters. This specification is useful when a parameter has meaning only when another parameter (called a controlling parameter) has a certain value.

You can also specify that a parameter be selected for prompting only if additional parameters are requested by pressing a function key during prompting. This specification can be used for parameters that are seldom specified by the user, either because the default is normally used or because they control seldom-used functions.

If you want to show all parameters for a command that has prompt control specified, you can request that all parameters be displayed by pressing F9 during prompting.

Conditional Prompting

When prompting the user for a command, a parameter which is conditioned by other parameters is displayed if:

- It is selected by the value specified for the controlling parameter.
- The value specified for the controlling parameter is in error.
- A value was specified for the conditioned parameter.
- A function key was pressed during prompting to request that all parameters be displayed.

When a user is to be prompted for a conditioned parameter and no value has yet been specified for its controlling parameter, all parameters previously selected are displayed. When the user presses the Enter key, the controlling parameter is then tested to determine if the conditioned parameter should be displayed or not.

To specify conditional prompting in the command definition source, specify a label name in the PMTCTL parameter on the PARM statement for each parameter that is conditioned by another parameter. The label specified must be defined on a PMTCTL statement which specifies the controlling parameter and the condition being tested to select the parameter for prompting. More than one PARM statement can refer to the same label.

On the PMTCTL statement, specify the name of the controlling parameter, one or more conditions to be tested, and the number of conditions that must be true to select the conditioned parameters for prompting. If the controlling parameter has special value mapping, the value entered on the PMTCTL statement must be the to-value. If the controlling parameter is a list or qualified name, only the first list item or qualifier is compared.

In the following example, parameters OUTFILE and OUTMBR is selected only if *OUTFILE is specified for the OUTPUT parameter, and parameter OUTQ is selected only if *PRINT is specified for the OUTPUT parameter.

```
     PARM OUTPUT TYPE(*CHAR) LEN(1) DFT(*) RSTD(*YES) +
            SPCVAL((*) (*PRINT P) (*OUTFILE F))
     PARM OUTFILE TYPE(Q1) PMTCTL(OUTFILE)
     PARM OUTMBR TYPE(*NAME) LEN(10) PMTCTL(OUTFILE)
     PARM OUTLINK TYPE(*CHAR) LEN(10)
     PARM OUTQ TYPE(Q1) PMTCTL(PRINT)
     Q1: QUAL TYPE(*NAME) LEN(10)
         QUAL TYPE(*NAME) LEN(10) SPCVAL(*LIBL) DFT(*LIBL)
OUTFILE: PMTCTL CTL(OUTPUT) COND((*EQ F)) NBRTRUE(*EQ 1)
PRINT:   PMTCTL CTL(OUTPUT) COND((*EQ P)) NBRTRUE(*EQ 1)
```

In this previous example, the user is prompted for the OUTLINK parameter after the condition for OUTMBR parameter has been tested. In some cases, the user should be prompted for the OUTLINK parameter before the OUTMBR parameter is tested. To specify a different prompt order, either reorder the parameters in the command definition source or use the PROMPT keyword on the PARM statement for the OUTLINK parameter.

A label can refer to a group of PMTCTL statements. This allows you to condition a parameter with more than one controlling parameter. To specify a group of PMTCTL statements, enter the label on the first statement in the group. No other statements can be placed between the PMTCTL statements in the group.

Use the LGLREL parameter to specify the logical relationship between the statements in the group. The LGLREL parameter is not allowed on the first PMTCTL statement in a group. For subsequent PMTCTL statements, the LGLREL parameter specifies the logical relationship (*AND or *OR) to the PMTCTL statement or statements preceding it. Statements in a group can be logically related in any combination of *AND and *OR relationships (*AND relationships are checked first, then *OR relationships).

The following example shows how the logical relationship is used to group multiple PMTCTL statements. In this example, parameter P3 is selected when any one of the following conditions exists:

- *ALL is specified for P1.
- *SOME is specified for P1 and *ALL is specified for P2.
- *NONE is specified for P1 and *ALL is not specified for P2.

```
PARM P1 TYPE(*CHAR) LEN(5) RSTD(*YES) VALUES(*ALL *SOME *NONE)
PARM P2 TYPE(*NAME) LEN(10) SPCVAL(*ALL)
PARM P3 TYPE(*CHAR) LEN(10) PMTCTL(PMTCTL1)
PMTCTL1:PMTCTL CTL(P1) COND((*EQ *ALL))
        PMTCTL CTL(P1) COND((*EQ *SOME)) LGLREL(*OR)
        PMTCTL CTL(P2) COND((*EQ *ALL)) LGLREL(*AND)
        PMTCTL CTL(P1) COND((*EQ *NONE)) LGLREL(*OR)
        PMTCTL CTL(P2) COND((*NE *ALL)) LGLREL(*AND)
```

An exit program can be specified to perform additional processing on a controlling parameter before it is tested. The exit program can be used to condition prompting based on:

- The type or other attribute of an object
- A list item or qualifier other than the first one
- An entire list or qualified name

To specify an exit program, specify the qualified name of the program in the PMTCTLPGM parameter on the PARM statement for the controlling parameter. The exit program is run during prompting when checking a parameter. The

conditions on the PMTCTL statement are compared with the value returned by the exit program rather than the value specified for the controlling parameter.

When the system cannot find or successfully run the exit program, the system assumes any conditions that would use the returned value as true.

The exit program must be written to accept three parameters:

- A 20-character field. The prompter passes the name of the command in the first 10 characters and the name of the controlling parameter in the last 10 characters. This field should not be changed.
- The value of the controlling parameter. This field is in the same format as it is when passed to the command processing program and should not be changed.
- If the controlling parameter is defined as VARY(*YES) the value is not preceded by a length value. If the controlling parameter is PASSATR(*YES), the attribute byte is not included.
- A 32-character field into which the exit program places the value to be tested in the PMTCTL statements.

  The value being tested in the PMTCTL statement must be returned in the same format as the declared data type.

In the following example, OBJ is a qualified name which may be the name of a command, program, or file. The exit program determines the object type and returns the type in the variable &RTNVAL:

```
CMD
PARM OBJ TYPE(Q1) PMTCTLPGM(CNVTYPE)
Q1: QUAL TYPE(*NAME) LEN(10)
    QUAL TYPE(*NAME) LEN(10) SPCVAL(*LIBL) DFT(*LIBL)
PARM CMDPARM TYPE(*CHAR) LEN(10) PMTCTL(CMD)
PARM PGMPARM TYPE(*CHAR) LEN(10) PMTCTL(PGM)
PARM FILEPARM TYPE(*CHAR) LEN(10) PMTCTL(FILE)
CMD: PMTCTL CTL(OBJ) COND((*EQ *CMD) (*EQ *)) NBRTRUE(*EQ 1)
PGM: PMTCTL CTL(OBJ) COND((*EQ *PGM) (*EQ *)) NBRTRUE(*EQ 1)
FILE: PMTCTL CTL(OBJ) COND((*EQ *FILE) (*EQ *)) NBRTRUE(*EQ 1)
```

The source for the exit program is shown here:

```
PGM PARM(&CMD &PARMVAL &RTNVAL)
DCL &CMD *CHAR 20                /* Command and parameter name      */
DCL &PARMVAL *CHAR 20            /* Parameter value                 */
DCL &RTNVAL *CHAR 32             /* Return value                    */
DCL &OBJNAM *CHAR 10             /* Object name                     */
DCL &OBJLIB *CHAR 10             /* Object type                     */
CHGVAR &OBJNAM %SST(&PARMVAL 1 10)
CHGVAR &OBJLIB %SST(&PARMVAL 11 10)
CHGVAR &RTNVAL '*'               /* Initialize return value to error*/
CHKOBJ &OBJLIB/&OBJNAM *CMD      /* See if command exists           */
MONMSG CPF9801 EXEC(GOTO NOTCMD) /* Skip if no command              */
CHGVAR &RTNVAL '*CMD'            /* Indicate object is a command    */
RETURN                           /* Exit                            */
NOTCMD:
CHKOBJ &OBJLIB/&OBJNAM *PGM      /* See if program exists           */
MONMSG CPF9801 EXEC(GOTO NOTPGM) /* Skip if no program              */
CHGVAR &RTNVAL '*PGM'            /* Indicate object is a program    */
RETURN                           /* Exit                            */
NOTPGM:
CHKOBJ &OBJLIB/&OBJNAM *FILE     /* See if file exists              */
MONMSG CPF9801 EXEC(RETURN)      /* Exit if no file                 */
CHGVAR &RTNVAL '*FILE'           /* Indicate object is a file       */
ENDPGM
```

Additional Parameters

You can specify that a parameter which is not frequently used will not be prompted for unless the user requests additional parameters by pressing a function key during prompting. This is done by specifying PMTCTL(*PMTRQS) on the PARM statement for the parameter. When prompting for a command, parameters with PMTCTL(*PMTRQS) coded will not be prompted unless a value was specified for them or the user presses F10 to request the additional parameters.

The prompter displays a separator line before the parameters with PMTCTL(*PMTRQS) to distinguish them from the other parameters. By default, all parameters with PMTCTL(*PMTRQS) are prompted last, even though they are not defined in that order in the command definition source. You can override this by specifying a relative prompt number in the PROMPT keyword. If you do this, however, it can be difficult to see what parameters were added to the prompt when F10 is pressed.

Using Key Parameters and a Prompt Override Program

The prompt override program allows current values rather than defaults to be displayed when a command is prompted.

If a prompt override program is defined for a command, you can see the results of calling the prompt override program in the following two ways:

- Type the name of the command without parameters on any command line and press F4=Prompt. The next screen shows the key parameters for the command. Key parameters are parameters, such as the name of an object, that uniquely identify the object.

  Complete all fields shown and press the Enter key. The next screen shows all command parameters, and the parameter fields that are not key parameter fields contain current values rather than defaults (such as *SAME and *PRV).

  For example, if you type `CHGLIB` on a command line and press F4=Prompt, you see only the Library parameter. If you then type `*CURLIB` and press the Enter key, the current values for your current library are displayed.
- Type the name of the command and the values for all key parameters on any command line. Press F4=Prompt. The next screen shows all command parameters, and the parameter fields that are not key parameter fields will contain current values rather than defaults (such as *SAME and *PRV).

  For example, if you type `CHGLIB LIB(*CURLIB)` on a command line and press F4=Prompt, the current values for your current library are displayed.

When F10=Additional parameters is pressed, any parameters defined with PMTCTL(*PMTRQS) are displayed with current values. For more information about additional parameters, see “Additional Parameters.”

To exit the command prompt, press F3=Exit.

Procedure for Using Prompt Override Programs

To use a prompt override program, do the following:

1. Specify any parameters that are to be key parameters on the PARM statement in the command definition source. For information about the KEYPARM parameter, see the following section, “Identifying Key Parameters” on page 326.
2. Write a prompt override program. For information about creating prompt override programs, see “Writing a Prompt Override Program” on page 326.

3. Specify the name of the prompt override program on the PMTOVRPGM parameter when you create or change the command. For information about creating or changing commands that use the prompt override program, see "Specifying the Prompt Override Program When Creating or Changing Commands" on page 329.

Identifying Key Parameters

The number of key parameters should be limited to the number of parameters needed to uniquely define the object to be changed.

To ensure a key parameter is coded correctly in the command definition source, do the following:

- Specify KEYPARM(*YES) on the PARM statement in the command definition source.
- Define all parameters that specify KEYPARM(*YES) before all parameters that specify KEYPARM(*NO).

  Note: If a PARM statement specifies KEYPARM(*YES) after a PARM statement that specifies KEYPARM(*NO), the parameter is not treated as a key parameter and a warning message is issued.
- Do not specify a MAX value greater than one in the PARM statement.
- Do not specify a MAX value greater than one for ELEM statements associated with key parameters.
- Do not specify *PMTRQS or a prompt control statement for the PMTCTL keyword on the PARM statement.
- Place key parameters in the command definition source in the same order you want them to appear when prompted.

Writing a Prompt Override Program

A prompt override program needs to be passed certain information to return current values when a command is prompted. You must consider both the passed information and the returned values when you write a prompt override program.

For an example of CL source for a prompt override program, see "CL Sample for Using the Prompt Override Program" on page 329.

Parameters Passed to the Prompt Override Program: The prompt override program is passed the following parameters:

- A 20-character field. The first 10 characters of the field contain the name of the command and the last 10 characters contain the name of the library.
- A value for each key parameter, if any. If more than one key parameter is defined, the parameter values are passed in the order that the key parameters are defined in the command definition source.
- A 32676-byte (32K) space to hold the command string that is created by the prompt override program. The first two bytes of this field must contain the hexadecimal length of the command string that is returned. The actual command string follows the first two bytes.

  For example, when defining two key parameters for a command, four parameters pass to the prompt override program as follows:
  - One parameter for the command.
  - Two parameters for the key parameters.
  - One parameter for the command string space.

Information Returned from the Prompt Override Program: Based on the values passed, the prompt override program retrieves the current values for the parameters that are not key parameters. These values are placed into a command string, where the length of the string is determined and returned.

Use the following guidelines to ensure your command string is correctly defined:

- Use the keyword format for the command string just as you would on the command line.
- Do not include the command name and the key parameters in the command string.
- Precede each keyword with a selective prompt character to define how to display the parameter and what value to pass to the CPP. For information about using selective prompt characters, see "Selective Prompting for CL Commands" on page 170.

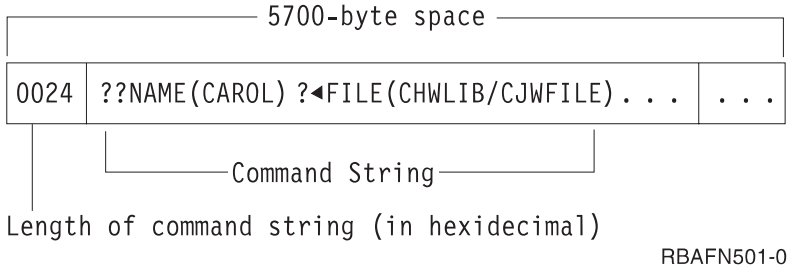
  When using selective prompts, do the following:

  – If a parameter is defined as MIN(1) in the command definition source (that is, the parameter is required), you must use the ?? selective prompt character for that keyword in the command string from the prompt override program.
  – Do not use the ?- selective prompt character in the prompt override program command string.

  The following example shows a command string returned from a prompt override program:

  ```
  ??Number(123456) ?<Qualifier(CLIB/CFILE) ?<LIST(ITEM1 ITEM2 ITEM3) ?<TEXT('Carol's file')
  ```

- Make sure that the specified value in the first two bytes of the space the program passes is the actual hexadecimal length of the command string.



- Include only the parameters in the command string whose current values you want displayed when the command is prompted. Parameters not included in the command string have their defaults displayed.
- Use character form for any numbers that appear in the command string. Do not use binary or packed form. Do not include any hexadecimal numbers in the command string.
- Do not put blank spaces between the library and the qualifier or the qualifier and the object. For example:

  ??KWD1(library /object)
  Not valid

  ??KWD1(library/ object)
  Not valid

  ??KWD1(library/object)
  Valid

  ??KWD1(library/object)
  Valid

- If you use special values or single values, make sure they are translated into the from-value defined in the command definition source.

  For example, a keyword has a special value defined as SPCVAL(*SPECIAL *) in the command definition source. *SPECIAL is the from-value and * is the to-value. When the current value is retrieved for this keyword, * is the value retrieved, but *SPECIAL must appear in the command string returned from the prompt override program. The correct from-value must be placed into the command string since more than one special value or single value can have the same to-value. For example, if `KWD1 SPCVAL((*SPC *) (*SPECIAL *))` is specified, the prompt override program must determine whether * is the to-value for *SPC or *SPECIAL.
- Define the length of fields used to retrieve text as follows:

  ```
  (2*(field length defined in command definition source)) + 2
  ```

  This length allows for the maximum number of quotation marks allowed in the text field. For example, if the TEXT parameter on the CHGxxx command is defined in the command definition source as LEN(50), then the parameter is declared as CHAR(102) in its prompt override program. For an example of how to define the length of fields used to retrieve text, see "CL Sample for Using the Prompt Override Program" on page 329.

  If the parameter for a text field is not correctly defined in the prompt override program and the text string retrieved by the prompt override program contains a quote, the command does not prompt correctly.
- Make sure that you double any embedded apostrophes, for example:

  ```
  ?<TEXT('Carol''s library')
  ```

Some commands can only be run in certain modes (such as DEBUG) or job status (such as *BATCH) but can still be prompted for from other modes or job statuses. When the command is prompted, the prompt override program is called regardless of the user's environment. If the prompt override program is called in a mode or environment that is not valid for the command, the defaults are displayed for the command and a value of 0 is returned for the length. Using the debug commands Change Debug (CHGDBG) and Add Program (ADDPGM) when not in debug mode are examples of this condition.

Allowing for Errors in a Prompt Override Program: If the prompt override program detects an error, it should do the following:

- Set the command string length to zero so that the defaults rather than current values are displayed when the command is prompted.
- Send a diagnostic message to the previous program on the call stack.
- Send escape message CPF0011.

For example, if you need a message saying that a library does not exist, add a message description similar to the following:

```
ADDMSGD      MSG('Library &2 does not exist') +
             MSGID(USR0012) +
             MSGF(QGPL/ACTMSG) +
             SEV(40) +
             FMT((*CHAR 4) (*CHAR 10))
```

Note: The substitution variable &1 is not in the message but is defined in the FMT parameter as 4 characters. &1 is reserved for use by the system and must always be 4 characters. If the substitution variable &1 is the only substitution variable defined in the message, you must ensure that the

fourth byte of the message data does not contain a blank when you send the message. The fourth byte is used by the system to manage messages during command processing and prompting.

This message can be sent to the calling program of the prompt override program by specifying the following in the prompt override program:

```
SNDPGMMSG    MSGID(USR0012) MSGF(QGPL/ACTMSG) +
             MSGDTA('0000' || &libname) MSGTYPE(*DIAG)
```

After the prompt override program sends all the necessary diagnostic messages, it should then send message CPF0011. To send message CPF0011, use the Send Program Message (SNDPGMMSG) command as follows:

```
SNDPGMMSG    MSGID(CPF0011) MSGF(QCPFMSG) +
             MSGTYPE(*ESCAPE)
```

When message CPF0011 is received, message CPD680A is sent to the calling program and displayed on the prompt screen to indicate that errors have been found. All diagnostic messages are placed in the user's job log.

Specifying the Prompt Override Program When Creating or Changing Commands

To use a prompt override program for a command you want to create, specify the program name when you use the Create Command (CRTCMD) command. You can also specify the program name when you change the command using the Change Command (CHGCMD) command. For both commands, specify the name of the prompt override program on the PMTOVRPGM parameter.

If key parameters are defined in the command definition source but the prompt override program is not specified when the command is created or changed, warning message CPD029B results. The key parameters are ignored, and when the command is prompted, it is displayed using the defaults specified in the command definition source.

Sometimes a prompt override program is specified when a command is created but when no key parameters are defined in the command definition source. In these cases, the prompt override program is called before the command is prompted; informational message CPD029A is sent when the command is created or changed.

CL Sample for Using the Prompt Override Program

The following example shows the command source for a command and the prompt override program. This command allows the ownership and text description of a library to be changed. The prompt override program for this command receives the name of the library; retrieves the current value of the library owner and the text description; and then places these values into a command string and returns it.

This prompt override program uses the ″?^″ selective prompt characters.

Sample Command Source

```
CHGLIBATR: CMD   PROMPT('Change Library Attributes')
           PARM  KWD(LIB) +
                   TYPE(*CHAR) MIN(1) MAX(1) LEN(10) +
                   KEYPARM(*YES) +
                   PROMPT('Library to be changed')
           PARM  KWD(OWNER) +
                   TYPE(*CHAR) LEN(10) MIN(0) MAX(1) +
                   KEYPARM(*NO) +
                   PROMPT('Library owner')
           PARM  KWD(TEXT) +
                   TYPE(*CHAR) MIN(0) MAX(1) LEN(50) +
                   KEYPARM(*NO) +
                   PROMPT('Text description')
```

Sample Prompt Override Program

```
 PGM PARM(&cmdname &keyparm1 &rtnstring)
/*************************************************************************/
/*                                                                       */
/*  Declarations of parameters passed to the prompt override program */
/*                                                                       */
/*************************************************************************/
 DCL  VAR(&cmdname)   TYPE(*CHAR) LEN(20)
 DCL  VAR(&keyparm1)   TYPE(*CHAR) LEN(10)
 DCL  VAR(&rtnstring) TYPE(*CHAR) LEN(5700)


/*************************************************************************/
/*                                                                       */
/* Return command string structure declaration                           */
/*                                                                       */
/*************************************************************************/

                     /* Length of command string generated         */
 DCL  VAR(&stringlen) TYPE(*DEC)  LEN(5 0) VALUE(131)
 DCL  VAR(&binlen)    TYPE(*CHAR) LEN(2)
                     /* OWNER keyword                              */
 DCL  VAR(&ownerkwd)  TYPE(*CHAR) LEN(8)  VALUE('?<OWNER(')
 DCL  VAR(&name)      TYPE(*CHAR) LEN(10)
                     /* TEXT keyword                               */
 DCL  VAR(&textkwd)   TYPE(*CHAR) LEN(8)  VALUE(' ?<TEXT(')
 DCL  VAR(&descript)  TYPE(*CHAR) LEN(102)


/*************************************************************************/
/*                                                                       */
/* Variables related to command string declarations                      */
/*                                                                       */
/*************************************************************************/
 DCL  VAR(&quote)      TYPE(*CHAR) LEN(1) VALUE('''')
 DCL  VAR(&closparen)  TYPE(*CHAR) LEN(1) VALUE(')')
```

```
/*********************************************************************/
/*                                                                   */
/*                  Start of operable code                           */
/*                                                                   */
/*********************************************************************/
/*********************************************************************/
/*                                                                   */
/* Monitor for exceptions                                            */
/*                                                                   */
/*********************************************************************/
    MONMSG  MSGID(CPF0000) +
            EXEC(GOTO CMDLBL(error))


/*********************************************************************/
/*                                                                   */
/* Retrieve the owner and text description for the library specified*/
/* on the LIB parameter.  Note:  This program assumes there are     */
/* no apostrophes in the TEXT description, such as (Carol's)  */
/*                                                                   */
/*********************************************************************/
    RTVOBJD OBJ(&keyparm1) OBJTYPE(*LIB) OWNER(&name) TEXT(&descript)

    CHGVAR VAR(%BIN(&binlen)) VALUE(&stringlen)


/*********************************************************************/
/*                                                                   */
/* Build the command string                                          */
/*                                                                   */
/*********************************************************************/
    CHGVAR VAR(&rtnstring) VALUE(&binlen)
    CHGVAR VAR(&rtnstring) VALUE(&rtnstring *TCAT &ownerkwd)
    CHGVAR VAR(&rtnstring) VALUE(&rtnstring *TCAT &name)
    CHGVAR VAR(&rtnstring) VALUE(&rtnstring *TCAT &closparen)
    CHGVAR VAR(&rtnstring) VALUE(&rtnstring *TCAT &textkwd)
    CHGVAR VAR(&rtnstring) VALUE(&rtnstring *TCAT &quote)
    CHGVAR VAR(&rtnstring) VALUE(&rtnstring *TCAT &descript)
    CHGVAR VAR(&rtnstring) VALUE(&rtnstring *TCAT &quote)
    CHGVAR VAR(&rtnstring) VALUE(&rtnstring *TCAT &closparen)

    GOTO CMDLBL(pgmend)
    ERROR:
    VALUE(0)
    CHGVAR VAR(%BIN(&rtnstring 1 2)) VALUE(&stringlen)
    VALUE(&binlen)
```

```
/*****************************************************************/
/*                                                               */
/* Send error message(s)                                         */
/*                                                               */
/* NOTE: If you wish to send a diagnostic message as well as CPF0011*/
/*       you will have to enter a valid error message ID in the  */
/*       MSGID parameter and a valid message file in the MSGF    */
/*       parameter for the first SNGPGMMSG command listed below. */
/*       If you do not wish to send a diagnostic message, do not */
/*       include the first SNDPGMMSG in your program.  However, in */
/*       error conditions, you must ALWAYS send CPF0011 so the   */
/*       second SNDPGMMSG command must be included in your program. */
/*                                                               */
/*****************************************************************/
    SNDPGMMSG MSGID(XXXXXXX) MSGF(MSGLIB/MSGFILE) MSGTYPE(*DIAG)
    SNDPGMMSG MSGID(CPF0011) MSGF(QCPFMSG) MSGTYPE(*ESCAPE)

PGMEND:
ENDPGM
```

Creating Commands

After you have defined your command through the command definition statements, you use the Create Command (CRTCMD) command to create the command. Besides specifying the command name, library name, and command processing program name for CL or high-level languages (HLL), or the source member, source file, command environment, and exit program for REXX, you can define the following attributes of the command:

- The validity checking used by the command
- The modes in which the command can be run
  - Production
  - Debug
  - Service
- Where the command can be used
  - Batch job
  - Interactive job
  - ILE CL module in a batch job
  - CL program in a batch job
  - ILE CL module in an interactive job
  - CL program in an interactive job
  - REXX procedure in a batch job
  - REXX procedure in an interactive job
  - As a command interpretively processed by the system through a call to QCMDEXC or QCAPCMD. (See Chapter 6, for information about QCMDEXC and QCAPCMD APIs.)
- The maximum number of parameters that can be specified by position
- The message file containing the prompt text
- The help panel group that is used as help for promptable parameters
- The help identifier name for the general help module used on this command
- The message file containing the messages identified on the DEP statement
- The current library to be active during command processing
- The product library to be active during command processing

- Whether an existing command with the same name, type, and library is replaced if REPLACE(*YES) is specified.
- The authority given to the public for the command and its description
- Text that briefly describes the command and its function

For commands with REXX CPPs, you can also specify the following:

- The initial command environment to handle commands when the procedure is started
- Exit programs to control running of your procedure

The following example defines a command named ORDENTRY to call an order entry application. The CRTCMD command defines the preceding attributes for ORDENTRY and creates the command using the parameter definitions contained in the member ORDENTRY in the IBM-supplied source file QCMDSRC. ORDENTRY contains the PARM statement used in the example under "Example of Defining a Parameter" on page 299.

```
CRTCMD     CMD(DSTPRODLB/ORDENTRY) +
           PGM(*LIBL/ORDENT) +
           TEXT('Calls order entry application')
```

The resulting command is:

```
ORDENTRY  OETYPE(value)
```

where the value can be DAILY, WEEKLY, or MONTHLY.

Once you have created a command, you can:

- Display the attributes of the command by using the Display Command (DSPCMD) command
- Change the attributes of the command by using the Change Command (CHGCMD) command
  - Delete the command by using the Delete Command (DLTCMD) command

Command Definition Source Listing

When you create a command, a source list is produced. The following shows a sample source list. The numbers refer to descriptions following the list.

```
 5722SS1 V5R3M0  031231 1              Command Definition       DSTPRODLB/ORDENTRY   SYSNAME  11/20/98 14:53:32 2 Page 1 3

Command name  . . . . . . . . . . . . . . . . . :  ORDENTRY
  Library . . . . . . . . . . . . . . . . . . . :    DSTPRODLB
Command processing program  . . . . . . . . . . :  ORDENT     4
  Library . . . . . . . . . . . . . . . . . . . :    *LIBL
Source file . . . . . . . . . . . . . . . . . . :  QCMDSRC
  Library . . . . . . . . . . . . . . . . . . . :    QGPL
Source file member  . . . . . . . . . . . . . . :  ORDENTRY    11/20/98  14:54:32
Validity checker program  . . . . . . . . . . . :  *NONE
Mode in which valid . . . . . . . . . . . . . . :  *PROD
                                                   *DEBUG
                                                   *SERVICE
Environment allowed . . . . . . . . . . . . . . :  *IREXX
                                                   *BREXX
                                                   *BPGM
                                                   *IPGM
                                                   *EXEC
                                                   *INTERACT
                                                   *BATCH
                                                   *BMOD
                                                   *IMOD
Allow limited user  . . . . . . . . . . . . . . :  *NO
Max positional parameters . . . . . . . . . . . :  *NOMAX
Prompt file . . . . . . . . . . . . . . . . . . :  *NONE
Message file  . . . . . . . . . . . . . . . . . :  QCPFMSG
  Library . . . . . . . . . . . . . . . . . . . :    *LIBL
Authority . . . . . . . . . . . . . . . . . . . :  *LIBCRTAUT
Replace command . . . . . . . . . . . . . . . . :  *YES
Enable graphical user interface . . . . . . . . :  *NO
Threadsafe  . . . . . . . . . . . . . . . . . . :  *NO
Multithreaded job action  . . . . . . . . . . . :  *SYSVAL
Text  . . . . . . . . . . . . . . . . . . . . . :  Calls order entry application
Help book name  . . . . . . . . . . . . . . . . :  *NONE
Help bookshelf. . . . . . . . . . . . . . . . . :  *NONE
```

```
Help panel group  . . . . . :   *NONE
Help identifier . . . . . :   *NONE
Help search index . . . . . :   *NONE
Current library . . . . . :   *NOCHG
Product library . . . . . :   *NOCHG
Prompt override program . . . . . :   *NONE
Compiler  . . . . . :   IBM AS/400 Command Definition Compiler 5

                                Command Definition Source
 6
SEQNBR  *...+... 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8 ...+... 9 ...+.  DATE   8
  100-              CMD        PROMPT('Order entry command')                                    11/20/98
7
  200-              PARM       KWD(OETYPE) TYPE(*CHAR) RSTD(*YES) +                             11/20/98
  300                            VALUES(DAILY WEEKLY MONTHLY) MIN(1) +                        11/20/98
  400                            PROMPT('Type of order entry:')                               11/20/98
                                      * * * * *  E N D  O F  S O U R C E  * * * * *

5769SS1 V4R4M0  990521               Command Definition               DSTPRODLB/ORDENTRY     11/20/98 14:54:32        Page    2
                                       Cross Reference

Defined Keywords 9
Keyword              Number          Defined      References
OETYPE                 001             200
                              * * * * *  E N D  O F  C R O S S  R E F E R E N C E  * * * * *

5769SS1 V4R4M0  990521               Command Definition               DSTPRODLB/ORDENTRY     11/20/98 14:54:32        Page    3
                                       Final Messages

Message              Sequence
ID                    Number          Sev        Text  10

                                       Message Summary
Total        Info        Error
    0           0            0   11
* CPC0202                             00        Command ORDENTRY created in library DSTPRODLB.  12
                               * * * * *  E N D  O F  C O M P I L A T I O N  * * * * *
```

Title:

1 The program number, version, release, modification level, and date of OS/400.

2 The date and time of this run.

3 The page number in the list.

Prologue

4 The parameter values specified (or defaults if not specified) on the CRTCMD command. If the source is not in a database file, the member name, date, and time are omitted.

5 The name of the create command definition compiler.

Source:

6 The sequence number of lines (records) in the source file. A dash following a sequence number indicates that a source statement begins at that sequence number. The absence of a dash indicates that a statement is the continuation of the previous statement. For example, a PARM source statement starts at sequence number 200 and continues at sequence numbers 300 and 400. (Note the continuation character of + on the PARM statement at sequence number 200 and 300.)

Comment source statements are handled like any other source statement and have sequence numbers.

7 The source statements.

8 The last date the source statement was changed or added. If the statement has not been changed or added, no date is shown. If the source is not in a database file, the date is omitted.

If an error is found during processing of the command definition statements and can be traced to a specific source statement, the error message is printed immediately following the source statement. An asterisk (*) indicates that the line contains an error message. The line contains the message identifier, severity, and the text of the message.

For more information about the command definition errors, see "Errors Encountered when Processing Command Definition Statements."

Cross-references:

9 The keyword table is a cross-reference list of the keywords validly defined in the command definition. The table lists the keyword, the position of the keyword in the command, the sequence number of the statement where the keyword is defined, and the sequence numbers of statements that refer to the keyword.

If valid labels are defined in the command definition, a cross-reference list of the labels (label table) is provided. The table lists the label, the sequence number of the statement where the label is defined, and the sequence numbers of statements that refer to the label.

Messages:

10 A list of the general error messages not listed in the source section that were encountered during processing of the command definition statements, if any. For each message, this section contains the message identifier, the sequence number of where the error occurred, the severity, and the message.

Message summary:

11 A summary of the number of messages issued during processing of the command definition statements. The total number is given along with totals by severity.

12 A completion message is printed following the message summary.

Errors Encountered when Processing Command Definition Statements

The types of errors that are caught during processing of the command definition statements include syntax errors, references to keywords and labels not defined, and missing statements. The following types of errors detected by the command definition compiler stop the command from being created (severity codes are ignored).

- Value errors
- Syntax errors

Even after an error that stops the command from being created is encountered, the command definition compiler continues to check the source for other errors. Syntax errors and fixed value errors prevent final checking that identifies errors in user names and values or references to keywords or labels. Checking for syntax errors and fixed value errors does continue. This lets you see and correct as many errors as possible before you try to create the command again. To correct errors made in the source statements, you use the EDTF (Edit File) command, or you can use the Source Entry Utility (SEU). For information about SEU, see the ADTS for AS/400:

Source Entry Utility book.

In the command definition source list, an error condition that relates directly to a specific source statement is listed after that command. See "Command Definition Source Listing" on page 333 for an example of these inline messages. Messages

that do not relate to a specific source statement but are more general in nature are listed in a messages section of the list, not inline with source statements.

Displaying a Command Definition

You can use the Display Command (DSPCMD) command to display or print the values that were specified as parameters on the CRTCMD command. The DSPCMD command displays the following information for your commands or for IBM-supplied commands:

- Qualified command name. The library name is the name of the library in which the command being displayed is located.
- Qualified name of the command processing program. The library name is the name of the library in which the command processing program resided when the command was created if a library name was specified on the CRTCMD or CHGCMD command. If a library name was not specified, *LIBL is displayed as the library qualifier. If the CPP is a REXX procedure, *REXX is shown.
- Qualified source file name, if the source file was a database file. The library name is the name of the library in which the source file was located when the CRTCMD command was processed. This field is blank if the source file was not a database file.
- Source file member name, if the source file was a database source file.
- If the CPP is a REXX procedure, the following information is shown:
  - REXX procedure member name
  - Qualified REXX source file name where the REXX procedure is located
  - REXX command environment
  - REXX exit programs
- Qualified name of the validity checking program. The library name is the name of the library in which the program resided when the command was created if a library name was specified on the CRTCMD or CHGCMD command. If a library name was not specified, *LIBL is displayed as the library qualifier.
- Valid modes of operation.
- Valid environments in which the command can be run.
- The positional limit for the command. *NOMAX is displayed if no positional limit exists for the command.
- Qualified name of the prompt message file. The library name is the name of the library in which the message file was located when the CRTCMD command was run. *NONE is displayed if no prompt message file exists for the command.
- Qualified name of the message file for the DEP statement. If a library name was specified for the message file when the command was created, that library name is displayed. If the library list was used when the command was created, *LIBL is displayed. *NONE is displayed if no DEP message file exists for the command.
- Qualified name of the help panel group.
- The help identifier name for the command.
- Qualified name for the prompt override program.
- Text associated with the command. Blanks are displayed if no text exists for the command.
- Indicator for whether the command prompt is enabled for conversion to a graphical user interface.
- Threadsafe indicator.

- Multithreaded job action, if the command is not threadsafe.

You can use the Retrieve Command Information (QCDRCMDI) API to return command attributes that were specified on the Create Command (CRTCMD) command when the command was created. You can also use the Retrieve Command Definition (QCDRCMDD) API to retrieve the structure of a command definition object, including parameter information, inter-parameter dependency information, and conditional prompting information. For more information about these APIs, see the *APIs* section of the **Programming** category of the iSeries Information Center.

Effect of Changing the Command Definition of a Command in a Procedure or Program

When a CL module or program is created, the command definitions of the commands in the procedure or program are used to generate the module or program. When the CL procedure or program is run, the command definitions are also used. If you specify a library name for the command in the CL procedure or program, the command must be in the same library at procedure creation time and at run time. If you specify *LIBL for the command in the CL procedure or program, the command is found, both at procedure creation and run time, using the library list (*LIBL).

You can make the following changes to the command definition statements for a command without re-creating the modules and programs that use the command. Some of these changes are made to the command definition statements source, which requires the command to be re-created. Other changes can be made with the Change Command (CHGCMD) command.

- Add an optional parameter in any position. Adding an optional parameter before the positional limit may affect any procedures, programs, and batch input streams that have the parameters specified in positional form.
- Change the REL and RANGE checks to be less restrictive.
- Add new special values. However, this could change the action of the procedure or program if the value could be specified before the change.
- Change the order of the parameters. However, changing the order of the parameters that precede the positional limit *will* affect any procedures, programs, and batch input streams that have the parameters specified in positional form.
- Increase the number of optional elements in a simple list.
- Change default values. However, this may affect the operation of the procedure or program.
- Decrease the number of required list items in a simple list.
- Change a parameter from required to optional.
- Change RSTD from *YES to *NO.
- Increase the length when FULL(*NO) is specified.
- Change FULL from *YES to *NO.
- Change the PROMPT text.
- Change the ALLOW value to be less restrictive.
- Change the name of the command processing program if the new command processing program accepts the correct number and type of parameters.
- Change the name of the validity checking program if the new validity checking program accepts the correct number and type of parameters.

- Change the mode in which the command can be run as long as the new mode does not affect the old mode of the same command that is used in a CL procedure or program.
- Change the TYPE to a compatible and less restrictive value. For example, change the TYPE from *NAME to *CHAR.
- Change the MAX value to greater than 1.
- Change the PASSATR and VARY values.

The following changes can be made to the command definition statements depending on what was specified in the CL procedure or program in which the command is used:

- Remove a parameter.
- Change the RANGE and REL values to be more restrictive.
- Remove special values.
- Decrease the number of elements allowed in a list.
- Change the TYPE value to be more restrictive or incompatible with the original TYPE value. For example, change the TYPE value from *CHAR to *NAME or change *PNAME to *CHAR.
- Add a SNGVAL parameter that was previously a list item.
- Change the name of an optional parameter.
- Remove a value from a list of values.
- Increase the number of required list items.
- Change a SNGVAL parameter to a SPCVAL parameter.
- Change a simple list to a mixed list of like elements.
- Change an optional parameter to a constant.
- Change RTNVAL from *YES to *NO, or from *NO to *YES.
- Change case value from *MIXED to *MONO.

The following changes can be made to the command definition statements, but may cause the procedure or program that uses the command to function differently:

- Change the meaning of a value.
- Change the default value.
- Change a SNGVAL parameter to a SPCVAL parameter.
- Change a value to a SNGVAL parameter.
- Change a list to a list within a list.
- Change case value from *MIXED to *MONO.

The following changes to the command definition statements require that the procedures or program using the command be re-created.

- Addition of a new required parameter.
- Removal of a required parameter.
- Changing the name of a required parameter.
- Changing a required parameter to a constant.
- Changing the command processing program to or from *REXX

In addition, if you specify *LIBL as the qualifier on the name of the command processing program or the validity checking when the command is created or

changed, you can move the command processing program or the validity checking to another library in the library list without changing the command definition statements.

Changing Command Defaults

You can change the default value of a command keyword by using the Change Command Default (CHGCMDDFT) command. Refer to the *CL* section of the **Programming** category of the iSeries Information Center for details. The keyword must have an existing default in order to allow a change to a new default value. You can change either an IBM-supplied command or a user-written command. You must use caution when changing defaults for IBM-supplied commands. The following are recommendations for changing defaults:

1. Use the Create Duplicate Object (CRTDUPOBJ) command to create a duplicate of the IBM-supplied command that you want to change in a user library. This allows other users on the system to use the IBM-supplied defaults if necessary.

   Use the Change System Library List (CHGSYSLIBL) command to move the user library ahead of QSYS or any other system-supplied libraries in the library list. This will allow the user to use the changed command without using the library qualifier.

   Changes to commands that are needed on a system-wide basis should be made in a user library. Additionally, you should add the user library name to the QSYSLIBL system value ahead of QSYS. The changed command is used system-wide. If you need to run an application that uses the IBM-supplied default, do so by using the Change System Library List (CHGSYSLIBL) command. Doing this removes the special library or library-qualify to the affected commands.
2. Installing a new release of a licensed program replaces all IBM-supplied commands for the licensed program on the machine. You should use a CL program to make changes to commands when installing a new release. This way you can run the CL program to duplicate the new commands to pick up any new keywords and make the command default changes.

   If an IBM-supplied command has new keywords, a copy of the command from a previous release may not run properly.

   The following is an example of a CL program that is used to delete the old version and create the new changed command:

```
PGM
DLTCMD USRQSYS/SIGNOFF
CRTDUPOBJ OBJ(SIGNOFF) FROMLIB(QSYS) OBJTYPE(*CMD) +
          TOLIB(USRQSYS) NEWOBJ(*SAME)
CHGCMDDFT CMD(USRQSYS/SIGNOFF) NEWDFT('LOG(*LIST)')
 .
 .
Repeat the DLTCMD, CRTDUPOBJ and CHGCMDDFT for each
command you want changed
 .
 .
ENDPGM
```

You can track changes you make to CL command defaults for use when you install a new release. To track changes, register an exit program for exit point QIBM_QCA_RTV_COMMAND. The exit program is called when you run the CHGCMDDFT command. One of the parameters passed to the exit program is the command string that is being run. You can save the command string to a source file and then compile the source file into a CL program. Finally, you use this program to reproduce the changes you have made to command defaults during

the previous release. For more information, see the Command Analyzer Retrieve exit program description in the *APIs* section of the **Programming** category of the iSeries Information Center.

The following steps can be used to build the NEWDFT command string for the CHGCMDDFT command. The USRQSYS/CRTCLPGM command is used in this example.

1. Create a duplicate copy of the command to be changed in a user library with the following command:
   ```
   CRTDUPOBJ OBJ(CRTCLPGM) FROMLIB(QSYS) OBJTYPE(*CMD) +
             TOLIB(USRQSYS)  NEWOBJ(*SAME)
   ```
2. Enter the command name to be changed in a source file referred to by the Source Entry Utility (SEU).
3. Press F4 to call the command prompter.
4. Enter any new default values for the keywords you want changed. In this example, `AUT(*EXCLUDE)` and `TEXT('Isn''t this nice text')` is entered.
5. Required keywords cannot have a default value; however, in order to get the command string in the source file, a valid value must be specified for each required keyword. Specify PGM1 for the PGM parameter.
6. Press the Enter key to put the command string into the source file. The command string returned would look like this:
   ```
   USRQSYS/CRTCLPGM PGM(PGM1) AUT(*EXCLUDE) +
   TEXT('Isn''t this nice text')
   ```
7. Remove the required keywords from the command string:
   ```
   USRQSYS/CRTCLPGM  AUT(*EXCLUDE) +
   TEXT('Isn''t this nice text')
   ```
   Remember that you may change only parameters, elements, or qualifiers that have existing default values. Specifying a value for a parameter, element, or qualifier that does not have an existing default value makes no default changes.
8. Insert the CHGCMDDFT at the beginning as shown in the example below:
   ```
   CHGCMDDFT USRQSYS/CRTCLPGM AUT(*EXCLUDE) +
   TEXT('Isn''t this nice text')
   ```
9. You must quote the input for the NEWDFT keyword as shown in the example below:
   ```
   CHGCMDDFT USRQSYS/CRTCLPGM 'AUT(*EXCLUDE) +
   TEXT('Isn''t this nice text')'
   ```
10. Since there are embedded apostrophes in the NEWDFT value, you must double them for the process to run properly:
    ```
    CHGCMDDFT USRQSYS/CRTCLPGM 'AUT(*EXCLUDE) +
    TEXT(''Isn''''t this nice text'')'
    ```
11. Now if you press F4 to call the command prompter, then F11 to request keyword prompting, you will see the following display:
    ```
    Command . . . . . :   CMD         R   CRTCLPGM
      Library . . . . . :                   USRQSYS
    New default parameter string:   NEWDFT      R   'AUT(*EXCLUDE)
    TEXT(''Isn''''t this nice text'')'
    ```
12. Now if you press the Enter key, the CHGCMDDFT command string is:
    ```
    CHGCMDDFT CMD(USRQSYS/CRTCLPGM) NEWDFT('AUT(*EXCLUDE) +
              TEXT(''Isn''''t  this nice text'')')
    ```
13. Press F1 to exit SEU and create and run the CL program or procedure.
14. The USRQSYS/CRTCLPGM will have default values of `*EXCLUDE for AUT` and `'Isn''t this nice text'` for TEXT.

Example 1

To provide a default value of *NOMAX for the MAXMBRS keyword of command CRTPF, do the following:

```
CRTPF FILE(FILE1) RCDLEN(96) MAXMBRS(1)
  .
  .
CHGCMDDFT CMD(CRTPF) NEWDFT('MAXMBRS(*NOMAX)')
```

Example 2

To provide a default value of 10 for the MAXMBRS keyword of the command CRTPF, do the following:

```
CRTPF FILE(FILE1) RCDLEN(96) MAXMBRS(*NOMAX)
  .
  .
CHGCMDDFT CMD(CRTPF) NEWDFT('MAXMBRS(10)')
```

Example 3

The following allows you to provide a default value of LIB001 for the first qualifier of the SRCFILE keyword and FILE001 for the second qualifier of the SRCFILE keyword for the command CRTCLPGM. The AUT keyword now have a default value of *EXCLUDE.

```
CRTCLPGM PGM(PROGRAM1) SRCFILE(*LIBL/QCMDSRC)
  .
  .
CHGCMDDFT CMD(CRTCLPGM) +
          NEWDFT('SRCFILE(LIB001/FILE001) AUT(*EXCLUDE)')
```

Example 4

The following provides a default value of 'Isn't this print text' for the PRTTXT keyword of the command CHGJOB. Since the NEWDFT keyword has embedded apostrophes, you must not double these apostrophes, or the process will not run correctly.

```
CHGJOB PRTTXT('Isn''t this print text')
   .
   .
CHGCMDDFT CMD(CHGJOB) +
          NEWDFT('PRTTXT(''Isn''''t this print text'')')
```

Example 5

The following provides a default value of QGPL for the first qualifier (library name) of the first list item of the DTAMBRS keyword for the command CRTLF. The new default value for the second list item of the DTAMBRS keyword (member name) is MBR1.

```
CRTLF FILE(FILE1) DTAMBRS(*ALL)
  .
  .
CHGCMDDFT CMD(CRTLF) +
          NEWDFT('DTAMBRS((QGPL/*N (MBR1)))')
```

Since *ALL is a SNGVAL (single value) for the entire DTAMBRS list, the defaults of *CURRENT for the library name and *NONE for the member name do not show up on the original command prompt display. The defaults *CURRENT and *NONE can be changed to a new default value but do not show up on the original prompt display because of the *ALL single value for the entire DTAMBRS list.

Example 6

Create a command that will display the spool files for a job:

```
CRTDUPOBJ OBJ(WRKJOB) FROMLIB(QSYS) +
          TOLIB(MYLIB) NEWOBJ(WRKJOBSPLF)
WRKJOBSPLF  OPTION(*SPLF)
```

```
  .
  .
CHGCMDDFT CMD(MYLIB/WRKJOBSPLF) +
          NEWDFT('OPTION(*SPLF)')
```

Writing a Command Processing Program or Procedure

A command processing program (CPP) can be a CL or HLL program, or a REXX procedure. Programs written in CL or HLL can also be called directly with the CALL CL command. REXX procedures can be called directly using the Start REXX Procedure (STRREXPRC) command. The command processing program does not need to exist when the Create Command (CRTCMD) command is run. If *LIBL is used as the library qualifier, the library list is used to find the command processing program when the created command is run.

Messages issued as a result of running the command processing program can be sent to the job message queue and automatically displayed or printed. You can send displays to the requesting display station.

Notes:

1. The parameters defined on the command are passed individually in the order they were defined (the PARM statement order).
2. Decimal values are passed to HLL and CL programs as packed decimal values of the length specified in the PARM statement.
3. Character, name, and logical values are passed to HLL and CL programs as a character string of the length defined in the PARM statement.

Writing a CL or HLL Command Processing Program

Figure 15 on page 343 shows the relationship between the Create Command (CRTCMD) command, the command definition statements, and the command processing program.

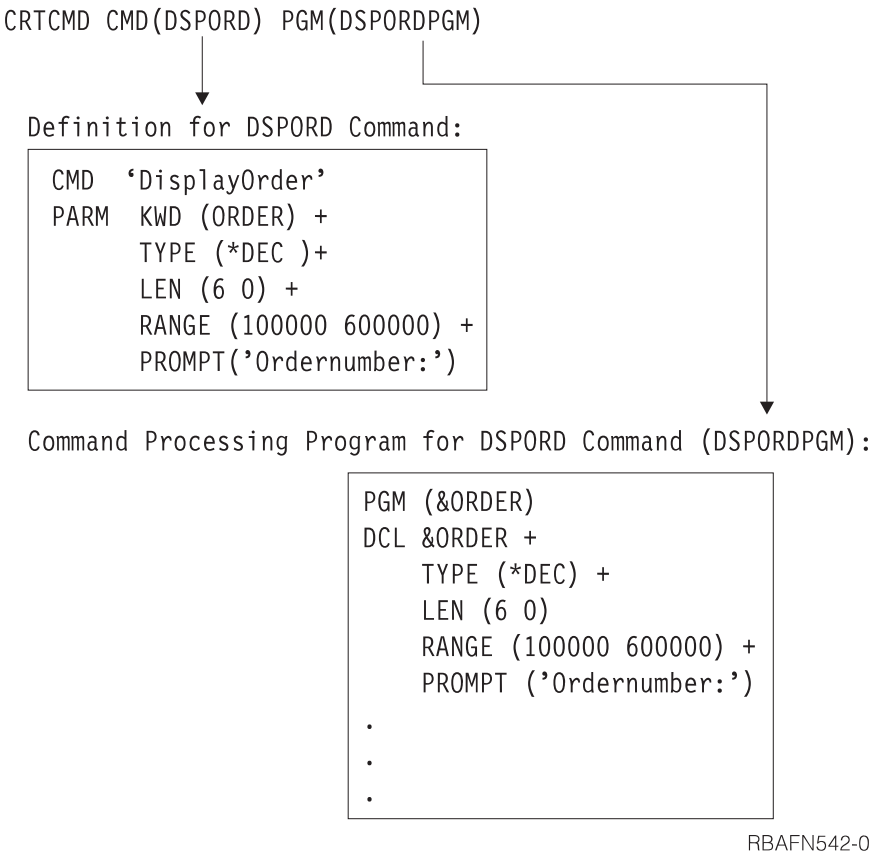


Figure 15. Command Relationships for CL and HLL

If the command processing program is a program written in CL, the variables that receive the parameter values must be declared to correspond to the type and length specified for each PARM statement. The following shows this correspondence. (Note the declare for the parameter ORDER in Figure 15.)

PARM Statement Type	PARM Statement Length	Declared Variable Type	Declared Variable Length
*DEC	x y[1]	*DEC	x y[1]
*LGL	1	*LGL	1
*CHAR	n	*CHAR	≤n[2]
*NAME	n	*CHAR	≤n[2]
*CNAME	n	*CHAR	≤n[2]
*SNAME	n	*CHAR	≤n[2]
*GENERIC	n	*CHAR	≤n[2]
*CMDSTR	n	*CHAR	≤n[2]
*DATE	7	*CHAR	7
*TIME	6	*CHAR	6
*INT2	n	*INT or *CHAR	2
*INT4	n	*INT or *CHAR	4
*UINT2	n	*UINT or *CHAR	2
*UINT4	n	*UINT or *CHAR	4

PARM Statement Type	PARM Statement Length	Declared Variable Type	Declared Variable Length
:			
[1]	x equals the length and y is the number of decimal positions.		
[2]	For character variables, if the length of the value passed is greater than the length declared, the value is truncated to the length declared. If RTNVAL(*YES) is specified, the length declared must equal the length defined on the PARM statement.		

A program written in CL used as a command processing program can process binary values (such as *INT2 or *INT4). The program can receive these values as character fields. In that case, the binary built-in function (%BINARY) can be used to convert them to decimal values. Otherwise, the CL program can declare them as integer variables.

The difference between *INT2 or *INT4 and *UINT2 or *UINT4 is that the *INT2 and *INT4 types are signed integers and the *UINT2 and *UINT4 types are unsigned integers. The default value for all integer types is 0. The *UINT2 and *UINT4 types have the same restrictions as the *INT and *INT4 types.

Note: The %BINARY built-in function is for use with signed integers. There is no corresponding function for unsigned integers.

For examples of command processing programs, see "Examples of Defining and Creating Commands" on page 346.

Writing a REXX Command Processing Procedure

Figure 16 shows the relationship between the Create Command (CRTCMD) command, the command definition statements, and the command processing procedure for REXX.

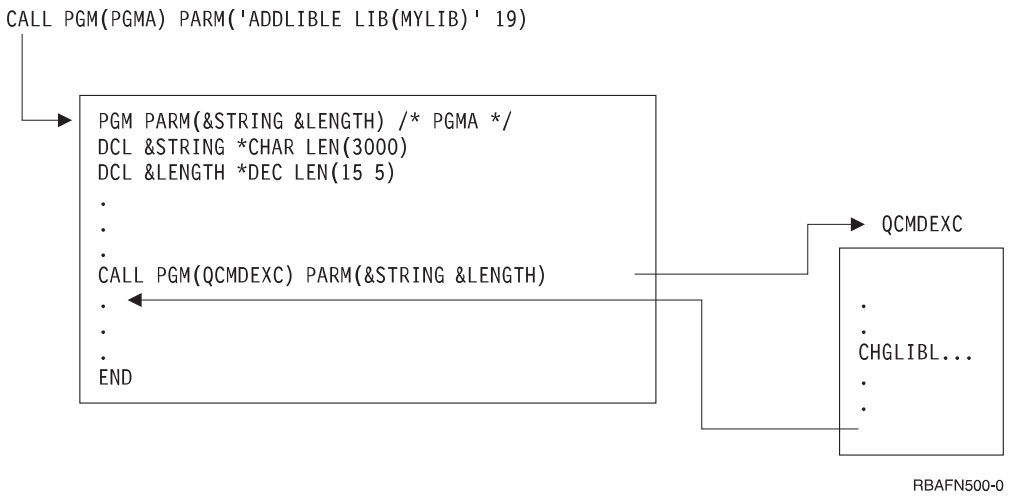


Figure 16. Command Relationships for REXX

Writing a Validity Checking Program

If you write a validity checking program for your command, specify the name of the validity checking program on the VLDCKR parameter on the Create Command (CRTCMD) command. The program does not have to exist when the CRTCMD command is run. If *LIBL is used as the library qualifier, the library list is used to find the validity checking program when the created command is run.

The following are two considerations for validity checking programs:

- The validity checking program is called only if the command syntax is correct. All parameters are passed to the program the same as they are passed to a command processing program.
- You should not use the validity checking program to change parameter values because the changed values are not always passed to the command processing program.

The remainder of this section describes how to send messages from a validity checking program that is written in CL.

If the validity checking program detects an error, it should send a diagnostic message to the previous call and then send escape message CPF0002. For example, if you need a message saying that an account number is no longer valid, you add a message description similar to the following to a message file:

```
ADDMSGD     MSG('Account number &2 no longer valid') +
            MSGID(USR0012) +
            MSGF(QGPL/ACTMSG) +
            SEV(40) +
            FMT((*CHAR 4) (*CHAR 6))
```

Note that the substitution variable &1 is not in the message but is defined in the FMT parameter as 4 characters. &1 is reserved for use by the system and must always be 4 characters. If the substitution variable &1 is the only substitution variable defined in the message, you must ensure that the fourth byte of the message data does not contain a blank when you send the message.

This message can be sent to the system by specifying the following in the validity checking:

```
SNDPGMMSG   MSGID(USR0012) MSGF(QGPL/ACTMSG) +
            MSGDTA('0000' || &ACCOUNT) MSGTYPE(*DIAG)
```

After the validity checking has sent all the necessary diagnostic messages, it should then send message CPF0002. The Send Program Message (SNDPGMMSG) command to send message CPF0002 looks like this:

```
SNDPGMMSG   MSGID(CPF0002) MSGF(QCPFMSG) +
            MSGTYPE(*ESCAPE)
```

When the system receives message CPF0002, it sends message CPF0001 to the calling program to indicate that errors have been found.

Message CPD0006 has been defined for use by the user-defined validity checking programs. An immediate message can be sent in the message data. Note in the following example that the message must be preceded by four character zeros.

The following shows an example of a validity checking program:

```
PGM PARM(&PARM01)
DCL VAR(&PARM01) TYPE(*CHAR) LEN(10)
IF COND(&PARM01 *EQ 'ERROR') THEN(DO)
SNDPGMMSG MSGID(CPD0006) MSGF(QCPFMSG) +
          MSGDTA('0000 DIAGNOSTIC MESSAGE FROM USER-DEFINED +
          VALIDITY CHECKER INDICATING THAT PARM01 IS IN ERROR.') +
          MSGTYPE(*DIAG)
SNDPGMMSG MSGID(CPF0002) MSGF(QCPFMSG) MSGTYPE(*ESCAPE)
ENDDO
ELSE
 .
 .
 .
ENDPGM
```

Examples of Defining and Creating Commands

This section contains examples of defining and creating commands.

Calling Application Programs

You can create commands to call application programs. If you create a command to call an application program, OS/400 performs validity checking on the parameters passed to the program. However, if you use the CALL command to call an application program, the application program must perform the validity checking.

For example, a label writing program (LBLWRT) writes any number of labels for a specific customer on either 1- or 2-part forms. When the LBLWRT program is run, it requires three parameters: the customer number, the number of labels, and the type of form to be used (ONE or TWO).

If the program were called directly from the display, the second parameter would be in the wrong format for the program. A numeric constant on the CALL command is always 15 digits with 5 decimal positions, and the LBLWRT program expects a 3-digit number with no decimal positions. A command can be created that provides the data in the format required by the program.

The command definition statements for a command to call the LBLWRT program are:

```
CMD PROMPT('Label Writing Program')
PARM  KWD(CUSNBR) TYPE(*CHAR) LEN(5) MIN(1) +
      PROMPT('Customer Number')
PARM  KWD(COUNT) TYPE(*DEC) LEN(3) DFT(20) RANGE(10 150) +
      PROMPT('Number of Labels')
PARM  KWD(FRMTYP) TYPE(*CHAR) LEN(3) DFT('TWO') RSTD(*YES) +
      SPCVAL(('ONE') ('TWO') ('1' 'ONE') ('2' 'TWO')) +
      PROMPT('Form Type')
```

For the second parameter, COUNT, a default value of 20 is specified and the RANGE parameter allows only values from 10 to 150 to be entered for the number of labels.

For the third parameter, FRMTYP, the SPCVAL parameter allows the display station user to enter `'ONE'`, `'TWO'`, `'1'`, or `'2'` for this parameter. The program expects the value `'ONE'` or `'TWO'`; however, if the display station user enters `'1'` or `'2'`, the command makes the necessary substitution for the FRMTYP parameter.

The command processing program for this command is the application program LBLWRT. If the application program were an RPG for OS/400 program, the following specifications would be made in the program to receive the parameters:

```
*ENTRY   PLIST
         PARM    CUST   5
         PARM    COUNT  30
         PARM    FORM   3
```

The CRTCMD command is:

```
CRTCMD CMD(LBLWRT) PGM(LBLWRT) SRCMBR(LBLWRT)
```

Substituting a Default Value

You can create a command that provides defaults for an IBM-supplied command and reduces the entries that the display station user must make. For example, you could create a Save Library on Tape (SAVLIBTAP) command that initializes a tape and saves a library on the tape device TAPE1. This command provides defaults for the standard Save Library (SAVLIB) command parameters and requires the display station user to specify only the library name.

The command definition statements for the SAVLIBTAP command are:

```
CMD PROMPT('Save Library to Tape')
PARM KWD(LIB) TYPE(*NAME) LEN(10) MIN(1) +
  PROMPT('Library Name')
```

The command processing program is:

```
PGM PARM(&LIB)
DCL &LIB TYPE(*CHAR) LEN(10)
INZTAP DEV(TAPE1) CHECK(*NO)
SAVLIB LIB(&LIB) DEV(TAPE1)
ENDPGM
```

The CRTCMD command is:

```
CRTCMD CMD(SAVLIBTAP) PGM(SAVLIBTAP) SRCMBR(SAVLIBTAP)
```

Displaying an Output Queue

You can create a command to display an output queue that defaults to display the output queue PGMR. The following command, DSPOQ, also allows the display station user to display any queue on the library list and provides a print option.

The command definition statements for the DSPOQ command are:

```
CMD PROMPT('WRKOUTQ.-Default to PGMR')
PARM  KWD(OUTQ) TYPE(*NAME) LEN(10) DFT(PGMR) +
      PROMPT('Output queue')
PARM  KWD(OUTPUT) TYPE(*CHAR) LEN(6) DFT(*) RSTD(*YES)
      VALUES(* *PRINT) PROMPT('Output')
```

The RSTD parameter on the second PARM statement specifies that the entry can only be one of the list of values.

The command processing program for the DSPOQ command is:

```
PGM PARM(&OUTQ &OUTPUT)
DCL &OUTQ TYPE(*CHAR) LEN(10)
DCL &OUTPUT TYPE(*CHAR) LEN(6)
WRKOUTQ OUTQ(*LIBL/&OUTQ) OUTPUT(&OUTPUT)
ENDPGM
```

The CRTCMD command is:

```
CRTCMD CMD(DSPOQ) PGM(DSPOQ) SRCMBR(DSPOQ)
```

The following command, DSPOQ1, is a variation of the preceding command. This command allows the workstation user to enter a qualified name for the output queue name, and the command defaults to *LIBL for the library name.

The command definition statements for the DSPOQ1 command are:

```
         CMD PROMPT('WRKOUTQ.-Default to PGMR')
         PARM    KWD(OUTQ) TYPE(QUAL1) +
                 PROMPT('Output queue:')
         PARM    KWD(OUTPUT) TYPE(*CHAR) LEN(6) RSTD(*YES) +
                 VALUES(* *PRINT) DFT(*) +
                 PROMPT('Output')
QUAL1:   QUAL TYPE(*NAME) LEN(10) DFT(PGMR)
         QUAL TYPE(*NAME) LEN(10) DFT(*LIBL) +
         SPCVAL(*LIBL)
```

The QUAL statements are used to define the qualified name that the user can enter for the OUTQ parameter. If the user does not enter a name, *LIBL/PGMR is used. The SPCVAL parameter is used because any library name must follow the rules for a valid name (for example, begin with A through Z), and the value *LIBL breaks these rules. The SPCVAL parameter specifies that if *LIBL is entered, OS/400 is to ignore the name validation rules.

The command processing program for the DSPOQ1 command is:

```
PGM PARM(&OUTQ &OUTPUT)
DCL &OUTQ TYPE(*CHAR) LEN(20)
DCL &OBJNAM TYPE(*CHAR) LEN(10)
DCL &LIB TYPE(*CHAR) LEN(10)
DCL &OUTPUT TYPE(*CHAR) LEN(6)
CHGVAR &OBJNAM %SUBSTRING(&OUTQ 1 10)
CHGVAR &LIB %SUBSTRING(&OUTQ 11 10)
WRKOUTQ OUTQ(&LIB/&OBJNAM) OUTPUT(&OUTPUT)
ENDPGM
```

Because a qualified name is passed from a command as a 20-character variable, the substring built-in function (%SUBSTRING or %SST) must be used in this program to put the qualified name in the proper CL syntax.

Displaying Messages from IBM Commands More Than Once

The CLROUTQ command issues the completion message CPF3417, which describes the number of entries deleted, the number not deleted, and the name of the output queue. If the CLROUTQ command is run within a CPP, the message is still issued but it becomes a detailed message because it is not issued directly by the CPP. For example, if a user-defined CLROUTQ command was issued from the Programmer Menu, the message would not be displayed. You can, however, receive an IBM message and reissue it from your CPP.

For example, you create a command named CQ2 to clear the output queue QPRINT2.

The command definition statements for the CQ2 command are:

```
CMD PROMPT ('Clear QPRINT2 output queue')
```

The CRTCMD command is:

```
CRTCMD CMD(CQ2) PGM(CQ2)
```

The CPP, which receives the completion message and displays it, is as follows:

```
PGM /* Clear QPRINT2 output queue CPP */
DCL &MSGID TYPE(*CHAR) LEN(7)
DCL &MSGDTA TYPE(*CHAR) LEN(100)
CLROUTQ QPRINT2
RCVMSG MSGID(&MSGID) MSGDTA(&MSGDTA) MSGTYPE(*COMP)
SNDPGMMSG MSGID(&MSGID) MSGF(QCPFMSG) MSGDTA(&MSGDTA) MSGTYPE(*COMP)
ENDPGM
```

The MSGDTA length for message CPF3417 is 28 bytes. However, by defining the variable &MSGDTA as 100 bytes, the same approach can be used on most messages because any unused positions are ignored.

Creating Abbreviated Commands

Example 1

You can create your own abbreviated commands to simplify IBM-supplied commands or to restrict the parameters allowed for users. For example, to allow users the ability to change only the printer device parameter, you can create your own Change Job (CJ) command. Following are three steps to create and implement your own CJ command:

- Step one: Command definition source statements

```
CMD   PROMPT('Change Job')

PARM  KWD(PRTDEV) +
      TYPE(*NAME) +
      LEN(10)  +
      SPCVAL(*SAME *USRPRF *SYSVAL *WRKSTN) +
      PROMPT('Printer Device')
```

- Step two: Processing program

```
PGM PARM(&PRTDEV)
DCL VAR(&PRTDEV)  TYPE(*CHAR)  LEN(10)
CHGJOB  PRTDEV(&PRTDEV)
ENDPGM
```

- Step three: CRTCMD command

```
CRTCMD  CMD(CJ)  PGM(CJ)  SRCMBR(CJ)
```

Example 2

You could create an abbreviated command called DW1 to start the printer writer W1.

The command definition statement is:

```
CMD /* Start printer writer command */
```

The command processing program is:

```
PGM
STRPRTWTR DEV(QSYSPRT) OUTQ(QPRINT) WTR(W1)
ENDPGM
```

The CRTCMD command is:

```
CRTCMD CMD(DW1) PGM(DW1) SRCMBR(DW1)
```

Deleting Files and Source Members

You can create a command to delete files and their corresponding source members in QDDSSRC.

The command definition statements for the command named DFS are:

```
CMD PROMPT('Delete File and Source')
PARM KWD(FILE) TYPE(*NAME) LEN(10) PROMPT('File Name')
```

The command processing program is written assuming that the name of the file and the source file member are the same. The program also assumes that both the file and the source file are on the library list. If the program cannot delete the file, an information message is sent and the command attempts to remove the source member. If the source member does not exist, an escape message is sent.

The command processing program is:

```
PGM PARM(&FILE)
DCL &FILE TYPE(*CHAR) LEN(10)
DCL &MSGID TYPE(*CHAR) LEN(7)
DCL &MSGDTA TYPE(*CHAR) LEN(80)
DCL &SRCFILE TYPE(*CHAR) LEN(10)
MONMSG MSGID(CPF0000) EXEC(GOTO ERROR) /* CATCH ALL */
DLTF &FILE
MONMSG MSGID(CPF2105) EXEC(DO) /* NOT FOUND */
RCVMSG MSGTYPE(*EXCP) MSGID(&MSGID) MSGDTA(&MSGDTA)
SNDPGMMSG MSGID(&MSGID) MSGF(QCPFMSG) MSGTYPE(*INFO) +
      MSGDTA(&MSGDTA)
GOTO TRYDDS
ENDDO
RCVMSG MSGTYPE(*COMP) MSGID(&MSGID) MSGDTA(&MSGDTA)
    /* DELETE FILE COMPLETED */
SNDPGMMSG MSGID(&MSGID) MSGF(QCPFMSG) MSGTYPE(*COMP) +
          MSGDTA(&MSGDTA) /* TRY IN QDDSSRC FILE */
TRYDDS:   CHKOBJ QDDSSRC OBJTYPE(*FILE) MBR(&FILE)
          RMVM QDDSSRC MBR(&FILE)
          CHGVAR &SRCFILE 'QDDSSRC'
          GOTO END
END:      RCVMSG MSGTYPE(*COMP) MSGID(&MSGID) MSGDTA(&MSGDTA)
              /* REMOVE MEMBER COMPLETED */
          SNDPGMMSG MSGID(&MSGID) MSGF(QCPFMSG) MSGTYPE(*COMP) +
          MSGDTA(&MSGDTA)
          RETURN
ERROR:    RCVMSG MSGTYPE(*EXCP) MSGID(&MSGID) MSGDTA(&MSGDTA)
              /* ESCAPE MESSAGE */
          SNDPGMMSG MSGID(&MSGID) MSGF(QCPFMSG) MSGTYPE(*ESCAPE) +
               MSGDTA(&MSGDTA)
ENDPGM
```

Deleting Program Objects

You can create a command to delete HLL programs and their corresponding source members.

The command definition statements for the command named DPS are:

```
CMD PROMPT ('Delete Program and Source')
PARM KWD(PGM) TYPE(*NAME) LEN(10) PROMPT('Program Name')
```

The command processing program is written assuming that the name of the program and the source file member are the same. Additionally, you have to use the IBM-supplied source files of QCLSRC, QRPGSRC, and QCBLSRC. The program also assumes that both the program and the source file are on the library list. If you cannot open the program, the system sends an information message, and the command attempts to remove the source member. If the source member does not exist, the system sends an escape message. The command processing program is:

```
PGM PARM(&PGM)
DCL &PGM TYPE(*CHAR) LEN(10)
DCL &MSGID TYPE(*CHAR) LEN(7)
DCL &MSGDTA TYPE(*CHAR) LEN(80)
```

```
DCL &SRCFILE TYPE(*CHAR) LEN(10)
MONMSG MSGID(CPF0000) EXEC(GOTO ERROR) /* CATCH ALL */
DLTPGM &PGM
MONMSG MSGID(CPF2105) EXEC(DO) /* NOT FOUND*/
RCVMSG MSGTYPE(*EXCP) MSGID(&MSGID) MSGDTA(&MSGDTA)
SNDPGMMSG MSGID(&MSGID) MSGF(QCPFMSG) MSGTYPE(*INFO) +
      MSGDTA(&MSGDTA)
GOTO TRYCL  /* TRY TO DELETE SOURCE MEMBER */
ENDDO
RCVMSG MSGTYPE(*COMP) MSGID(&MSGID) MSGDTA(&MSGDTA)
    /* DELETE PROGRAM COMPLETED */
SNDPGMMSG MSGID(&MSGID) MSGF(QCPFMSG) MSGTYPE(*COMP) +
     MSGDTA(&MSGDTA) /* TRY IN QCLSRC */
TRYCL:   CHKOBJ QCLSRC OBJTYPE(*FILE) MBR(&PGM)
         MONMSG MSGID(CPF9815) EXEC(GOTO TRYRPG) /* NO CL MEMBER */
         RMVM QCLSRC MBR(&PGM)
         CHGVAR &SRCFILE 'QCLSRC'
         GOTO END
TRYRPG:   /* TRY IN QRPGSRC FILE */
         CHKOBJ QRPGSRC OBJTYPE(*FILE) MBR(&PGM)
         MONMSG MSGID(CPF9815) EXEC(GOTO TRYCBL) /* NO RPG MEMBER */
         RMVM QRPGSRC MBR(&PGM)
         CHGVAR &SRCFILE 'QRPGSRC'
         GOTO END
TRYCBL:   /* TRY IN QCBLSRC FILE */
         CHKOBJ QCBLSRC OBJTYPE(*FILE) MBR(&PGM)
         /* ON LAST SOURCE FILE LET CPF0000 OCCUR FOR A NOT FOUND +
         CONDITION */
         RMVM QCBLSRC MBR(&PGM)
         CHGVAR &SRCFILE 'QCBLSRC'
         GOTO END
TRYNXT:   /* INSERT ANY ADDITIONAL SOURCE FILES */
          /* ADD MONMSG AFTER CHKOBJ IN TRYCBL AS WAS +
            DONE IN TRYCL AND TRYRPG */
END:      RCVMSG MSGTYPE(*COMP) MSGID(&MSGID) MSGDTA(&MSGDTA)
               /*REMOVE MEMBER COMPLETED */
          SNDPGMMSG MSGID(&MSGID) MSGF(QCPFMSG) MSGTYPE(*COMP) +
             MSGDTA(&MSGDTA)
          RETURN
ERROR:    RCVMSG MSGTYPE(*EXCP) MSGID(&MSGID) MSGDTA(&MSGDTA)
              /* ESCAPE MESSAGE */
          SNDPGMMSG MSGID(&MSGID) MSGF(QCPFMSG) MSGTYPE(*ESCAPE) +
               MSGDTA(&MSGDTA)
          ENDPGM
```

Chapter 10. Documenting Commands

If you choose to define and create your own CL commands, as discussed in Chapter 9, "Defining Commands," on page 289, you may also decide to create online command help to describe your commands. This chapter describes how you can create online help for your own commands, and how to create HTML command documentation using the online command help.

Commands and Command Help

Command prompting and online command help are powerful features of CL commands. If you have developed one or more of your own CL commands, you know that all of the command prompting capabilities used by IBM CL commands are available for you to use for your commands. The same is true for command help; you can create help to describe your commands for users.

The first step is to understand how the connections work between commands and command help.

- Command help information is stored in panel group objects. The symbolic object type for a panel group is *PNLGRP. A help panel group consists of help modules. Each help module has a help module name.
- There are two parameters on the CRTCMD (Create Command) command that make the connection between a command (*CMD) object and an online help panel group: **HLPID** (Help identifier) and **HLPPNLGRP** (Help panel group).
- There are four types of command help modules in an online help panel group:
  1. Command-level help module
  2. Parameter-level help module
  3. Command examples help module
  4. Command error messages help module

When you try to view the help for a CL command (for instance, by pressing the F1 (Help) key while prompting the command), OS/400 determines whether the command has a help panel group associated with the command. If the command was created specifying a panel group name for the HLPPNLGRP parameter, or was changed by the CHGCMD (Change Command) command to have a help panel group associated with the command, the help stored in the panel group is retrieved, formatted, and displayed. OS/400 attempts to retrieve help from the following help modules in the help panel group:

- A command-level help module that has the same name as the value specified for the HLPID parameter when the command was created or changed. For example, if command STRPAY was created specifying HLPID(STRPAY), OS/400 looks for a help module named STRPAY.
- A parameter-level help module for each command parameter, with the exception of constant parameters. The help module name must be the command's HLPID value followed by a forward slash character followed by the parameter keyword name. For example, if command STRPAY has a HLPID value of STRPAY and a parameter named TITLE, OS/400 looks for a parameter-level help module named STRPAY/TITLE.
- A help module containing one or more examples of using the command. The help module name must be the command's HLPID value followed by a forward

© Copyright IBM Corp. 1997, 2004

slash character followed by COMMAND/EXAMPLES. For example, if command STRPAY has a HLPID value of STRPAY, OS/400 looks for a command examples help module named STRPAY/COMMAND/EXAMPLES.

- A help module containing the list of monitorable messages that may be signaled from the command. The help module name must be the command's HLPID value followed by a forward slash character followed by ERROR/MESSAGES. For example, if command STRPAY has a HLPID value of STRPAY, OS/400 looks for a command examples help module named STRPAY/ERROR/MESSAGES.

When viewing extended CL command help on a 5250 terminal (or using 5250 emulator software), command-level and parameter-level help sections are required, and the examples and error messages help sections are optional. If the command-level help section or any parameter-level help section is not found in the online help panel group, diagnostic message CPF6E01 (Help information is incomplete) is displayed at the bottom of the help display. If the examples or error messages help modules are not found, no diagnostic message is displayed.

Writing Command Help

Once you understand how a command (*CMD) object can be connected to an online help panel group (*PNLGRP) object, you can move on to actually writing the help text that goes into the four types of help modules for a command. iSeries online help for commands is written in a tag language known as UIM (user interface manager). The UIM source is compiled using the CRTPNLGRP (Create Panel Group) command to create a *PNLGRP object. For more information about help panel groups and the UIM panel group definition language, see Appendix A of the Application Display Programming book.

Generating UIM Source for Command Help

As a simpler alternative to learning UIM syntax, you can use the Generate Command Documentation (GENCMDDOC) command. This command allows you to create a file containing UIM source; the file provides a template for the online command help. To create the UIM source, you need to specify *UIM for the GENOPT (Generation Options) parameter. The information used to create the template is retrieved from the command object (*CMD) you specify. Using GENCMDDOC to create the UIM source can simplify writing online help for your CL commands.

For more information about GENCMDDOC, see the command's documentation in the CL topic in the iSeries Information Center.

The following is an example of using the GENCMDDOC command to generate a UIM template:

```
GENCMDDOC   CMD(MYLIB/MYCMD)
            TODIR('/QSYS.LIB/MYLIB.LIB/QPNLSRC.FILE')
            TOSTMF(*CMD)  GENOPT(*UIM)
```

In the example, the command retrieves information from the command object named MYCMD in library MYLIB and generates UIM source into member MYCMD of source file QPNLSRC in library MYLIB. After the template UIM source has been generated, you will need to edit the UIM source as follows:

- Ensure that the command-level help module describes the purpose of the command. The start of the first sentence is provided followed by a **<...>** marker that you need to replace with the appropriate text. Ensure that restrictions that

apply to running the command are described, such as commands that must be run first, or special authorizations needed. Some example restrictions are provided; you should edit these to match your command's restrictions.

- Ensure that each parameter-level help module explains the purpose of the parameter. The start of the first sentence is provided followed by a **<...>** marker that you need to replace with the appropriate text. You may also want to describe any inter-parameter dependencies or restrictions; you can use the :NT. (Note) and :ENT. (End of Note) UIM tags for these parameter-level restrictions.

  The parameter-level help module also should provide a description of the possible choices for the parameter. Headings for each special value or single value are provided, but you need to replace the **<...>** markers with the parameter value descriptions.
- Ensure that examples are provided as needed. The command examples help module is primed with headings for two examples. If the command has no parameters, you could edit this help module to have only a single example. If the command has many parameters, or provides several distinct functions, you can copy the headings to create additional command examples. You need to edit the command examples to insert your own commands' parameter keywords and parameter values and to replace the **<...>** markers with the descriptions of what the example commands would do.
- Ensure that error message text is provided as needed. The command error messages help module is primed with headings showing how you can use the &MSG. built-in function to imbed the message text of error messages sent by the command. You need to edit the list of messages to contain the actual message identifiers of messages signaled from your command, along with the message file that contains the message descriptions.

Once you've edited the UIM source to tailor it to your command, you can create the online help panel group by using the CRTPNLGRP (Create Panel Group) command. The following is an example of using the CRTPNLGRP command:

```
CRTPNLGRP   PNLGRP(MYLIB/MYCMD)
            SRCFILE(MYLIB/QPNLSRC)  SRCMBR(MYCMD)
```

This command attempts to create a panel group from the UIM source in member MYCMD in source physical file QPNLSRC in library MYLIB. If there are no severe errors found when compiling the UIM source, a panel group (*PNLGRP) object named MYCMD is created in library MYCMD. The command generates a spooled file, which can be viewed to see informational, warning, and severe errors found by the UIM compiler.

Sharing Common Help

The previous section describes how to create a panel group that contains help modules for just one command. Because of the naming scheme used for connecting command online help modules with a command, you could store the help modules for many commands into a single panel group. When help modules for several related commands are in a single panel group, you can use :IMHELP. (Imbed Help) UIM tag to share common help information. For example, you might have several commands that have a parameter named OUTPUT, which has the same general description on each command. Sharing the same command help information ensures consistency across all the commands that have this parameter.

Another option for sharing online help text is to use the :IMPORT. (Import) UIM tag, which lets you define one or more help modules that are in another panel group that can be imbedded dynamically using :IMHELP tags.

For more information about the :IMHELP. and :IMPORT. UIM tags, see Appendix A of the Application Display Programming book.

Organizing Help Text into Help Modules

You can choose how to organize the help text that goes into the help modules in your panel group. Using the UIM capability to imbed help from one help module into one or more other help modules by using the :IMHELP. tag, you can divide the help text for any of the four types of command help modules defined earlier amongst two or more help modules. The most common reason to split the help text into smaller help modules is to facilitate sharing of common help text.

When defining help modules that will be imbedded into other help modules, be sure to choose help module names that will not be confused with one of the four types of online command help modules.

Generating HTML Source for Command Documentation

In addition to creating help information for your command, you may want to create command documentation that can be viewed when not connected to an iSeries system, or printed. OS/400 lets you generate command documentation in Hyper Text Markup Language (HTML) source that can be displayed using an internet browser, or imported into many word processing programs. To do this, you can use the Generate Command Documentation (GENCMDDOC) command, specifying *HTML for the GENOPT (Generation Options) parameter. The command generates a file containing information retrieved from the command object (*CMD) you specify and any command help panel group (*PNLGRP) objects that already exist for the command.

For more information about using GENCMDDOC, see the command's documentation in the CL topic in the iSeries Information Center.

To see what the HTML output looks like and how it matches the online help, compare the help for any command and the CL command documentation provided in the iSeries Information Center. IBM builds the Information Center command documentation from the online command help.

The following is an example of using the GENCMDDOC command to generate HTML source:

```
GENCMDDOC   CMD(MYLIB/MYCMD)
```

In this example, the command retrieves information from the command object named MYCMD in library MYLIB and generates HTML source into stream file MYLIB_MYCMD.HTML in the current working directory of the job. The generated stream file can be viewed in HTML source format using the DSPF (Display File), or in HTML browser format using any standard internet browser software, or by command.

Chapter 11. Debugging Programs

Debugging ILE Programs

Debugging allows you to detect, diagnose, and eliminate errors in a program. You can debug your ILE programs by using the ILE source debugger. This chapter describes how to use the ILE source debugger.

This chapter describes how to:

- Prepare your ILE program for debugging
- Start a debug session
- Add and remove programs from a debug session
- View the program source from a debug session
- Set and remove conditional and unconditional breakpoints
- Step through a program
- Display the value of variables
- Change the value of variables
- Display the attributes of variables
- 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 commands:

- Use the Start Debug (STRDBG) command and retain the default *NO for the UPDPROD parameter.
- Use the Change Debug (CHGDBG) command.

See the *CL* section of the **Programming** category of the iSeries Information Center for more information.

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

The ILE Source Debugger

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

- Debug any ILE CL 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
- Set and remove conditional and unconditional breakpoints
- Step through a specified number of statements
- Display or change the value of variables

© Copyright IBM Corp. 1997, 2004

- Display the attributes of a variable

When a program stops because of a breakpoint or a step command, the applicable 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 use the debug options (DBGVIEW) when you create a module object or program object using Create CL Module (CRTCLMOD) or Create Bound CL (CRTBNDCL). After you set the breakpoints or other ILE source debugger options, you can call 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 case, lower case, or mixed case.

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

Table 10 summarizes these debug commands. The online help for the ILE source debugger describes the debug commands and explains their allowed abbreviations.

Table 10. 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.
BREAK	Permits you to enter either an unconditional or conditional breakpoint at a position in the program being tested. Use BREAK *position* WHEN *expression* to enter a conditional breakpoint.
SBREAK	Permits you to enter a service entry point at a position in the program being tested. A service entry point is a type of breakpoint established in a program to facilitate the system debugger in gaining control of a spawned job. The breakpoint is only signaled when the job within which the service entry point was hit is not currently under debug.
CLEAR	Permits you to remove conditional and unconditional breakpoints.
DISPLAY	Allows you to display the names and definitions assigned by using the EQUATE command. It also allows you to display a different source module than the one currently shown on the Display Module Source display. The module object must exist in the current program object.
EQUATE	Allows you to assign an expression, variable, or debug command to a name for shorthand use.
EVAL	Allows you to display or change the value of a variable or to display the value of expressions.
QUAL	Allows you to define the scope of variables that appear in subsequent EVAL commands.
STEP	Allows you to run one or more statements of the program being debugged.
FIND	Searches 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.

Table 10. ILE source debugger commands (continued)

Debug Command	Description
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 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.
SET™	Specifies if case sensitive or case insensitive searching is performed for all subsequent FIND requests in the current debug session. It also allows you to change the update production files value.
WATCH	Displays a list of the currently active watch conditions.

Preparing a Program Object for a Debug Session

Before you can use the ILE source debugger, you must use either the CRTCLMOD or CRTBNDCL command and specify the DBGVIEW option.

For each ILE CL module object that you want to debug, you can create one of three views:

- Root source view
- Listing view
- Statement view

Using a Root Source View

A root source view contains the source statements of the source member.

To use the root source view with the ILE source debugger, the ILE CL compiler creates the root source view while the module object (*MODULE) is being created.

Note: The module object is created by 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.

To debug an ILE CL module object by using a root source view, use the *SOURCE or *ALL option on the DBGVIEW parameter for either the CRTCLMOD or CRTBNDCL commands.

One way to create a root source view, is as follows:

```
CRTCLMOD
MODULE(MYLIB/MYPGM) SRCFILE(MYLIB/QCLLESRC) SRCMBR(MYPGM) TEXT('CL Program')
DBGVIEW(*SOURCE)
```

The Create CL Module (CRTCLMOD) command with *SOURCE for the DBGVIEW parameter creates a root source view for module object *MYPGM*.

Using a Listing View

A listing view is similar to the source code portion of the compile listing or spool file produced by the ILE CL compiler.

To debug an ILE CL module object by using a listing view, use the *LIST or *ALL option on the DBGVIEW parameter for either the CRTCLMOD or CRTBNDCL commands when you create the module.

One way to create a listing view is as follows:

```
CRTCLMOD
MODULE(MYLIB/MYPGM) SRCFILE(MYLIB/QCLLESRC) SRCMBR(MYPGM) TEXT('CL Program')
DBGVIEW(*LIST)
```

Using a Statement View

A statement view does not contain any CL source data. However, breakpoints can be added by using procedure names and statement numbers found in the compiler listing. To debug an ILE CL module object using a statement view, you need a copy of the compiler listing.

Note: No data is shown in the Display Module Source display when a statement view is used to debug an ILE CL module object.

To debug an ILE CL module object by using a statement view, use the *STMT, *SOURCE, *LIST, or *ALL option on the DBGVIEW parameter for either the CRTCLMOD or CRTBNDCL commands when you create the module.

One way to create a statement view is as follows:

```
CRTCLMOD
MODULE(MYLIB/MYPGM) SRCFILE(MYLIB/QLSRC) SRCMBR(MYPGM) TEXT('CL Program')
DBGVIEW(*STMT)
```

Starting the ILE Source Debugger

After you create the 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.

Initially, you can add as many as twenty (20) program objects and twenty (20) service programs to a debug session. Do this by using the Program (PGM) and Service Program (SRVPGM) parameters on the STRDBG command. The program objects can be any combination of ILE or original program model (OPM) programs. To start a debug session with three program objects, type:

```
STRDBG PGM(*LIBL/MYPGM1 *LIBL/MYPGM2 *LIBL/MYPGM3) SRVPGM(*LIBL/SRVPGM1 *LIBL/SRVPGM2)
DBGMODSRC(*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 for ILE program objects. The first module object bound to the program object with debug data is shown.

The option to use the ILE source debugger to debug OPM programs exists for users. OPM programs contain source debug data when created. Do this only by specifying either the OPTION(*SRCDBG) or the OPTION(*LSTDBG) parameter of the Create CL Program (CRTCLPGM) command. The source debug data is actually part of the program object.

To add OPM programs that are created containing source debug data to the ILE source debugger, use the Program (PGM) and OPM Source Level Debug (OPMSRC) parameters on the STRDBG command. To start a debug session with an OPM program created with source debug data, type:

```
STRDBG PGM(*LIBL/MYOPMPGM) OPMSRC(*YES) DSPMODSRC(*YES)
```

Adding Program Objects to a Debug Session

You can add more program objects 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 Table 10 on page 358 for a list of ILE source debugger commands. 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.

```
                          Work with Module List
                                                              System:   SYSTEM01
Type options, press enter.
  1=Add program   4=Remove program   5=Display module source
  8=Work with module breakpoints

Opt     Program/module     Library        Type
 1       weekday2             *LIBL         *PGM
         DSPWKDAY           MYLIB         *PGM
         DSPWKDAY                         *MODULE     Selected
         AABP1                            *MODULE

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

Figure 17. Adding an ILE Program Object to a Debug Session. When the Enter is pressed, program WEEKDAY2 is added to the debug session

```
                          Work with Module List
                                                                System:   SYSTEM01
Type options, press enter.
  1=Add program   4=Remove program   5=Display module source
  8=Work with module breakpoints

Opt     Program/module     Library        Type
         WEEKDAY2          *LIBL          *PGM
         WEEKDAY2           MYLIB         *PGM
         WEEKDAY2                         *MODULE
         DSPWKDAY           MYLIB         *PGM
         DSPWKDAY                         *MODULE     Selected
         AABP1                            *MODULE




                                                                          Bottom
Command
===>
F3=Exit   F4=Prompt   F5=Refresh   F9=Retrieve   F12=Cancel
Program WEEKDAY2 added to source debugger.
```

Figure 18. Adding an ILE Program Object to a Debug Session. The information message at the bottom of the display shows that program WEEKDAY2 was added to the debug session.

When you have finished adding program objects to the debug session, press F3 (Exit) from the Work with Module List display to return to the Display Module Source display. You can also use option 5 (Display Module Source) to select and display a module.

To add OPM programs to a debug session, use the Add Program (ADDPGM) command. A debug session can include up to twenty (20) OPM programs at any given time. You can add OPM programs that contain source debug data to the debug session by using option 1 (Add program) on the Work with Module List display. (This is true provided the debug session allows OPM source level debugging.) You can allow OPM source level debugging by starting the debug session and by using the OPMSRC parameter on the STRDBG command. If the OPMSRC parameter was not specified on the STRDBG command, activate OPM source level debugging. Do this by using the OPM Source Level Debug (OPMSRC) parameter on the Change Debug (CHGDBG) command. Alternately, you can change the value of the OPM source debug support option by using the SET debug command.

Removing Program Objects from a Debug Session

You can remove program objects 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 you want to remove, on the Work with Module List display. See Figure 19 on page 363. The Work with Module List display can be accessed from the Display Module Source display by pressing F14 (Work with Module List). To remove a service program, change the default program type from *PGM to *SRVPGM.

```
                          Work with Module List
                                                                System:   SYSTEM01
Type options, press enter.
  1=Add program   4=Remove program   5=Display module source
  8=Work with module breakpoints

Opt     Program/module     Library        Type
                *LIBL     *PGM
 4        WEEKDAY2           MYLIB        *PGM
    WEEKDAY2                             *MODULE
    DSPWKDAY          MYLIB              *PGM
    DSPWKDAY                             *MODULE     Selected
    AABP1                                *MODULE

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

Figure 19. Removing an ILE Program Object from a Debug Session. When the Enter key is pressed, program WEEKDAY2 is removed from the debug session.

```
                          Work with Module List
                                                                System:   SYSTEM01
Type options, press enter.
  1=Add program   4=Remove program   5=Display module source
  8=Work with module breakpoints

Opt     Program/module     Library        Type
                *LIBL     *PGM
          DSPWKDAY           MYLIB          *PGM
          DSPWKDAY                          *MODULE     Selected
          AABP1                             *MODULE

                                                                          Bottom
Command
===>
F3=Exit   F4=Prompt   F5=Refresh   F9=Retrieve   F12=Cancel
Program WEEKDAY2 removed from source debugger.
```

Figure 20. Removing an ILE Program Object from a Debug Session

When you have finished removing program objects 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 from a debug session.

To remove OPM programs from a debug session, use the Remove Program (RMVPGM) command. If OPM source level debugging is active, OPM programs that are created with source debug data may be listed on the Work with Module List display. You can remove these programs from the debug session by using option 4 (Remove program) on the Work with Module List display.

Viewing the Program Source

The Display Module Source display shows the source of a program object one module object at a time. A module object's source can be shown if the module object was compiled using one of the following debug view options:

- DBGVIEW(*ALL)
- DBGVIEW(*SOURCE)
- DBGVIEW(*LISTING)

There are two methods to change what is shown on the Display Module Source display:

- Change a view
- Change a module

When you change a view, the ILE source debugger maps to equivalent positions in the view you are changing to. When you change the module, the executable statement on the displayed view is stored in memory and is viewed when the module is displayed again. Line numbers that have breakpoints set are highlighted. When a breakpoint, step, or message causes the program to stop and the display to be shown, the source line where the event occurred is highlighted.

Changing a Module Object

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 Display Module Source display is shown in Figure 21.

To select a module object, type 5 (Display module source) next to the module object you want to show.

```
                          Display Module Source

Program:   DSPWKDAY       Library:   MYLIB          Module:   DSPWKDAY
    24      500-              CALL       PGM(WEEKDAY2) PARM(&DAYOFWK)
    25      600-              IF         COND(&DAYOFWK *EQ 1) THEN(CHGVAR +
    26      700                            VAR(&WEEKDAY) VALUE('Sunday'))
    27      800-              ELSE       CMD(IF COND(&DAYOFWK *EQ 2) THEN(CHGV
    28      900                            VAR(&WEEKDAY) VALUE('Monday')))
    29     1000-              ELSE       CMD(IF COND(&DAYOFWK *EQ 3) THEN(CHGV
    30     1100                           VAR(&WEEKDAY) VALUE('Tuesday')))
    31     1200-              ELSE       CMD(IF COND(&DAYOFWK *EQ 4) THEN(CHGV
    32     1300                           VAR(&WEEKDAY) VALUE('Wednesday')))
    33     1400-              ELSE       CMD(IF COND(&DAYOFWK *EQ 5) THEN(CHGV
    34     1500                           VAR(&WEEKDAY) VALUE('Thursday')))
    35     1600-              ELSE       CMD(IF COND(&DAYOFWK *EQ 6) THEN(CHGV
    36     1700                           VAR(&WEEKDAY) VALUE('Friday')))
    37     1800-              ELSE       CMD(IF COND(&DAYOFWK *EQ 7) THEN(CHGV
    38     1900                           VAR(&WEEKDAY) VALUE('Saturday')))
                                                                      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 21. Display a Module View

After you select the module object that you want to view, press Enter. The selected module object is shown in the Display Module Source display.

An alternate method of changing a module object 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 or service program object that has been added to the debug session.

Changing the View of a Module Object

Several views of an ILE CL module object are available depending on the values you specify when you create an ILE CL module object. These views are:

- Root source view
- Listing view
- Statement 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 22. The current view is listed at the top of the window, and the other views that are available are shown below. Each module object in a program object can have a different set of views available, depending on the debug options used to create it.

To select a view, type 1 (Select) next to the view you want to show.

```
                          Display Module Source
.........................................................................
:                              Select View                              :
:                                                                       :
:  Current View . . . :   CL Root Source                                :
:                                                                       :
:  Type option, press Enter.                                            :
:    1=Select                                                           :
:                                                                       :
:  Opt     View                                                         :
:          CL Root Source                                               :
:   1      CL Listing View                                              :
:                                                                       :
:                                                                       :
:                                                                Bottom :
:  F12=Cancel                                                           :
:                                                                       :
:.......................................................................:
                                                                  More...
Debug . . .

F3=End Program   F6=Add/Clear breakpoint   F10=Step   F11=Display variable
F12=Resume   F17=Watch variable   F18=Work with watch   F24=More keys
```

Figure 22. Changing a View of a Module Object

After you select the view of the module object that you want to show, press Enter and the selected view of the module object is shown in the Display Module Source display.

Setting and Removing Breakpoints

You can use breakpoints to stop a program object at a specific point when it is running. An **unconditional breakpoint** stops the program object at a specific statement. A **conditional breakpoint** stops the program object when a specific condition at a specific statement is met.

When the program object stops, the Display Module Source display is shown. The appropriate module object is shown with the source positioned at the line where the breakpoint occurred. This line is highlighted. At this point, you can evaluate variables, set more breakpoints, and run any of the debug commands.

You should know the following characteristics about breakpoints before using them:

- When a breakpoint is bypassed, for example with the GOTO statement, that breakpoint isn't processed.
- 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 to program objects
  – Removing breakpoints from program objects
  – Displaying breakpoint information
  – Resuming the running of a program object after a breakpoint has been reached.

Setting and Removing Unconditional Breakpoints

You can set or remove an unconditional breakpoint by using:

- F6 (Add/Clear breakpoint) from the Display Module Source display
- F13 (Work with Module Breakpoints) from the Display Module Source display
- The BREAK debug command to set a breakpoint
- The CLEAR debug command to remove a breakpoint

The simplest way to set and remove an unconditional breakpoint is to use F6 (Add/Clear breakpoint) from the Display Module Source display. To set an unconditional breakpoint using F6, place your cursor on the line to which you want to add the breakpoint and press F6. An unconditional breakpoint is then set on the line. To remove an unconditional breakpoint, place your cursor on the line from which you want to remove the breakpoint and press F6. The breakpoint is then removed from the line.

Repeat the previous steps for each unconditional breakpoint you want to set.

Note: If the line on which you want to set a breakpoint is not a runnable statement, the breakpoint is set at the next runnable statement.

After the breakpoints are set, press F3 (Exit) 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 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 breakpoints is to use the BREAK and CLEAR debug commands.

To set an unconditional breakpoint by using the BREAK debug command, type:

```
BREAK line-number
```

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.

To remove an unconditional breakpoint by 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.

If using the statement view, there is no line numbers displayed. To set unconditional breakpoints in the statement view, type:

```
BREAK procedure-name/statement-number
```

on the debug command line. *Procedure-name* is the name of your CL module. *Statement-number*(from the compiler listing) is the statement number where you wanted to stop.

Setting and Removing Conditional Breakpoints

You can set or remove a conditional breakpoint by using:

- The Work with Breakpoints display
- The BREAK debug command to set a breakpoint
- The CLEAR debug command to remove a breakpoint

Using the Work with Breakpoints Display:

Note: The relational operators supported for conditional breakpoints are <, >, =, <=, >=, and <> (not equal).

One way you can set or remove conditional breakpoints is through the Work with Module Breakpoints display. 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 23 on page 368. To set a conditional breakpoint, type the following:

- 1 (Add) in the *Opt* field,
- the debugger line number where you want to set the breakpoint in the *Line* field,
- a conditional expression in the *Condition* field,

and press Enter. For example, to set a conditional breakpoint at debugger line 35, as shown in Figure 23 on page 368, type the following:

- 1 (Add) in the *Opt* field,
- 35 in the *Line* field,
- type &I=21 in the *Condition* field,

and press Enter.

To remove a conditional breakpoint, type 4 (Clear) in the *Opt* field next to the breakpoint you want to remove, and press Enter. You can also remove unconditional breakpoints in this manner.

```
                    Work with Module Breakpoints
                                                          System:   SYSTEM01
Program  . . . :   MYPGM               Library  . . . :   MYLIB
  Module . . . :     MYMOD             Type . . . . . :   *PGM

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

Opt     Line       Condition
 1      35____     &I=21_____________________
 _      ______     __________________________
```

Figure 23. Setting a Conditional Breakpoint

Repeat the previous steps for each conditional breakpoint you want to set or remove.

Note: If the line on which you want to set a breakpoint is not a runnable statement, the breakpoint is set at the next runnable statement.

After you specify all breakpoints that you want to set or remove, press F3 (Exit) to return to the Display Module Source display.

Then press F3 (Exit) 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.

Call the program object. When a statement with a conditional breakpoint is reached, the conditional expression associated with the breakpoint is evaluated before the statement is run. If the result is false, the program object continues to run. If the result is true, the program object stops, and the Display Module Source display is shown. At this point, you can evaluate variables, set more breakpoints, and run any of the debug commands.

Using the BREAK and CLEAR Debug Commands: An alternate method of setting and removing conditional breakpoints is to use the BREAK and CLEAR debug commands.

To set a conditional breakpoint by 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. *expression* is the conditional expression that is evaluated when the breakpoint is encountered. The relational operators supported for conditional breakpoints are <, >, =, <=, >=, and <> (not equal).

In non-numeric conditional breakpoint expressions, the shorter expression is implicitly padded with blanks before the comparison is made. This implicit padding occurs before any National Language Sort Sequence (NLSS) translation. See “National Language Sort Sequence (NLSS)” on page 369 for more information on NLSS.

To remove a conditional breakpoint by using the CLEAR debug command, type:

```
CLEAR line-number
```

on the debug command line. *Line-number* is number in the currently displayed view of the module object from which you want to remove a breakpoint.

In the statement view, no line numbers are displayed. To set conditional breakpoints in the statement view, type:

```
BREAK procedure-name/statement-name WHEN expression
```

on the debug command line. *Procedure-name* is the name of your CL module. *Statement-number*(from the compiler listing) is the statement number where you want to stop.

National Language Sort Sequence (NLSS): Non-numeric conditional breakpoint expressions are divided into the following two types:

- Char- 8: each character contains 8 bits
- Char-16: each character contains 16 bits (DBCS)

NLSS applies only to non-numeric conditional breakpoint expressions of type Char-8. See Table 11 on page 370 for the possible combinations of non-numeric conditional breakpoint expressions.

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

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

Note: The name of the sort sequence table is saved during compilation. At debug time, the source debugger uses the name saved from the compilation to access the sort sequence table. If the sort sequence table specified at compilation time resolves to something other than *HEX or *JOBRUN, it is important the sort sequence table does *not* get altered before debugging is started. If the table cannot be accessed because it is damaged or deleted, the source debugger uses the *HEX sort sequence table.

Table 11. Non-numeric Conditional Breakpoint Expressions

Type	Possibilities
Char-8	• Character variable compared to character variable • Character variable compared to character literal [1] • Character variable compared to hex literal [2] • Character literal [1] compared to character variable • Character literal [1] compared to character literal [1] • Character literal [1] compared to hex literal [2] • Hex literal [2] compared to character variable [1] • Hex literal [2] compared to character literal [1] • Hex literal [2] compared to hex literal [2]
Char 16	• DBCS character variable compared to DBCS character variable • DBCS character variable compared to graphic literal [3] • DBCS character variable compared to hex literal [2] • Graphic literal [3] compared to DBCS character variable • Graphic literal [3] compared to Graphic literal [3] • Graphic literal [3] compared to hex literal [2] • Hex literal [2] compared to DBCS character variable • Hex literal [2] compared to Graphic literal [3]

:

1 Character literal is of the form 'abc'.

2 Hexadecimal literal is of the form X'hex digits'.

3 Graphic literal is of the form G'<so>DBCS data<si>'. Shift-out is represented as <so> and shift-in is represented as <si>.

Conditional Breakpoint Examples:

```
CL declarations:   DCL    VAR(&CHAR1) TYPE(*CHAR) LEN(1)
                   DCL    VAR(&CHAR2) TYPE(*CHAR) LEN(2)
                   DCL    VAR(&DEC1) TYPE(*DEC) LEN(3 1)
                   DCL    VAR(&DEC2) TYPE(*DEC) LEN(4 1)


Debug command:     BREAK 31 WHEN &DEC1 = 48.1

Debug command:     BREAK 31 WHEN &DEC2 > &DEC1

Debug command:     BREAK 31 WHEN &CHAR2 <> 'A'

Comment:           'A' is implicitly padded to
                   the right with one blank character before
                   the comparison is made.


Debug command:     BREAK 31 WHEN %SUBSTR(&CHAR2 2 1) <= X'F1'

Debug command:     BREAK 31 WHEN %SUBSTR(&CHAR2 1 1) >= &CHAR1

Debug command:     BREAK 31 WHEN %SUBSTR(&CHAR2 1 1) < %SUBSTR(&CHAR2 2 1)
```

The %SUBSTR built-in function allows you to substring a character string variable. The first argument must be a string identifier, the second argument is the starting position, and the third argument is the number of single byte or double byte characters. Arguments are delimited by one or more spaces.

Removing All Breakpoints

You can remove all breakpoints, conditional and unconditional, from a program object that has a module object shown on the Display Module Source display by using the CLEAR PGM debug command. To use the debug command, type:

```
CLEAR PGM
```

on the debug command line. The breakpoints are removed from all of the modules bound to the program or service program.

Stepping through the Program Object

After a breakpoint is encountered, you can run a specified number of statements of a program object, then stop the program again and return to the Display Module Source display. The program object begins running on the next statement of the module object in which the program stopped. Typically, a breakpoint is used to stop the program object.

You can step through a program object by using:

- F10 (Step) or F22 (Step into) on the Display Module Source display
- The STEP debug command

Using F10 or F22 on the Display Source Display

The simplest way to step through a program object one statement at a time is to use F10 (Step) or F22 (Step into) on the Display Module Source display. When you press F10 (Step) or F22 (Step into), then next statement of the module object shown in the Display Module Source display is run, and the program object is stopped again.

Note: You cannot specify the number of statements to step through when you use F10 (Step) or F22 (Step into). Pressing F10 (Step) or F22 (Step into) performs a single step.

Another way to step through a program object is to use the STEP debug command. The STEP debug command allows you to run more than one statement in a single step.

Using the STEP Debug Command

The default number of statements to run, using the STEP debug command, is one. To step through a program object using the STEP debug command, type:

```
STEP number-of-statements
```

on the debug command line. *Number-of-statements*is the number of statements of the program object that you want to run in the next step before the program object is stopped again. For example, if you type

```
STEP 5
```

on the debug command line, the next five statements of your program object are run, then the program object is stopped again and the Display Module Source display is shown.

Step Over and Step Into

When a CALL statement to another program object is encountered in a debug session, you can do either of the following:

- Step over the called program object, or
- Step into the called program object.

If you choose to **step over** the called program object, then the CALL statement and the called program object are run as a single step. The called program object is run to completion before the calling program object is stopped at the next step. Step over is the default step mode.

If you choose to **step into** the called program object, then each statement in the called program object is run as a single step. If the next step at which the running program object is to stop falls within the called program object, the called program object is stopped at this point. The called program object is then shown in the Display Module Source display if the called program object is compiled with debug data and you have the correct authority to debug it.

Stepping over Program Objects

You can step over program objects by using:

- F10 (Step) on the Display Module Source display
- The STEP OVER debug command

Using F10(Step)

You can use F10 (Step) on the Display Module Source display to step over a called program object in a debug session. If the next statement to be run is a CALL statement to another program object, then pressing F10 (Step) will cause the called program object to run to completion before the calling program object is stopped again.

Using the Step Over Debug Command

Alternatively, you can use the STEP OVER debug command to step over a called program object in a debug session. To use the STEP OVER debug command, type:

```
STEP number-of-statements OVER
```

on the debug command line. *Number-of-statements* is the number of statements of the program object that you want to run in the next step before the program object is halted again. If one of the statements that are run contains a CALL statement to another program object, the ILE source debugger steps over the called program object.

Stepping into Program Objects

You can step into program objects by using:

- F22 (Step into) on the Display Module Source display
- The STEP INTO debug command

Using F22(Step Into)

You can use F22 (Step into) on the Display Module Source display to step into a called program object in a debug session. If the next statement to be run is a CALL statement to another program object, pressing F22 (Step into) causes the first statement in the called program object to be run. The called program object is then shown in the Display Module Source display.

Note: The called program object must have debug data associated with it in order for it to be shown in the Display Module Source display.

Using the Step Into Debug Command

Alternatively, you can use the STEP INTO debug command to step into a called program object in a debug session. To use the STEP INTO debug command, type:

```
STEP number-of-statements INTO
```

on the debug command line. *Number-of-statements* is the number of statements of the program object that you want to run in the next step before the program object is stopped again. If one of the statements that are run contains a CALL statement to another program object, the debugger steps into the called program object. Each statement in the called program object is counted in the step. If the step ends in the called program object then the called program object is shown in the Display Module Source display. For example, if you type

```
STEP 5 INTO
```

on the debug command line, the next five statements of the program object are run. If the third statement is a CALL statement to another program object, then two statements of the calling program object are run and the first three statements of the called program object are run.

Displaying Variables

You can display the value of variables 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. However, you do not need to specifically define the scope of the variables contained in a CL module because they are all of global scope.

Using F11(Display Variable)

To display a variable using F11 (Display variable), place your cursor on the variable that you want to display and press F11. The current value of the variable is shown on the message line at the bottom of the Display Module Source display.

```
                          Display Module Source

Program:   DSPWKDAY       Library:   MYLIB          Module:   DSPWKDAY
     4               DCL        VAR(&MSGTEXT) TYPE(*CHAR) LEN(20)
     5               CALL       PGM(WEEKDAY2) PARM(&DAYOFWK)
     6               IF         COND(&DAYOFWK *EQ 1) THEN(CHGVAR +
     7                            VAR(&WEEKDAY) VALUE('Sunday'))
     8               ELSE       CMD(IF COND(&DAYOFWK *EQ 2) THEN(CHGVAR +
     9                            VAR(&WEEKDAY) VALUE('Monday')))
    10               ELSE       CMD(IF COND(&DAYOFWK *EQ 3) THEN(CHGVAR +
    11                            VAR(&WEEKDAY) VALUE('Tuesday')))
    12               ELSE       CMD(IF COND(&DAYOFWK *EQ 4) THEN(CHGVAR +
    13                            VAR(&WEEKDAY) VALUE('Wednesday')))
    14               ELSE       CMD(IF COND(&DAYOFWK *EQ 5) THEN(CHGVAR +
    15                            VAR(&WEEKDAY) VALUE('Thursday')))
    16               ELSE       CMD(IF COND(&DAYOFWK *EQ 6) THEN(CHGVAR +
    17                            VAR(&WEEKDAY) VALUE('Friday')))
    18               ELSE       CMD(IF COND(&DAYOFWK *EQ 7) THEN(CHGVAR +
                                                                    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
&DAYOFWK = 3.
```

Figure 24. Displaying a Variable using F11 (Display variable). Using the EVAL Debug Command

You can also use the EVAL debug command to determine the value of a variable. 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. If the value cannot be shown on a single line, it is shown on the Evaluate Expression display.

For example, to display the value of the variable *&DAYOFWK*; on line 7 of the module object shown in Figure 24 on page 373, type:

```
EVAL &DAYOFWK
```

The message line of the Display Module Source display shows `&DAYOFWK = 3.` as in Figure 24 on page 373.

Display logical variable example

```
CL declarations:   DCL    VAR(&LGL1) TYPE(*LGL) VALUE('1')


Debug command:     EVAL &LGL1

Result:            &LGL1 = '1'
```

Display character variable examples

```
CL declarations:    DCL    VAR(&CHAR1) TYPE(*CHAR) LEN(10) VALUE('EXAMPLE')

Debug command:     EVAL &CHAR1

Result:            &CHAR1 = 'EXAMPLE   '


Debug command:     EVAL %SUBSTR(&CHAR1 5 3)

Result:            %SUBSTR(&CHAR1 5 3) = 'PLE'


Debug command:     EVAL %SUBSTR(&CHAR1 7 4)

Result:            %SUBSTR(&CHAR1 7 4) = 'E   '
```

The %SUBSTR built-in function allows you to substring a character string variable. The first argument must be a string identifier, the second argument is the starting position, and the third argument is the number of single byte or double byte characters. Arguments are delimited by one or more spaces.

Display decimal variable example

```
CL declarations:    DCL    VAR(&DEC1) TYPE(*DEC) LEN(4 1) VALUE(73.1)

CL declarations:    DCL    VAR(&DEC2) TYPE(*DEC) LEN(3 1) VALUE(12.5)

Debug command:     EVAL &DEC1

Result:            &DEC1 = 073.1

Debug command:     EVAL &DEC2

Result:            &DEC2 = 12.5
```

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, on the debug command line, type:

```
EVAL variable-name: x  number-of-bytes
```

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 *number-of-bytes* indicates the number of bytes displayed. 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.

```
CL declaration:  DCL    VAR(&CHAR1) TYPE(*CHAR) LEN(10) VALUE('ABC')
                 DCL    VAR(&CHAR2) TYPE(*CHAR) LEN(10) VALUE('DEF')

Debug command:    EVAL &CHAR1:X 32

Result:
00000     C1C2C340 40404040 4040C4C5 C6404040 ABC       DEF
00010     40404040 00000000 00000000 00000000     ............
```

Changing the Value of Variables

You can change the value of variables by using the EVAL command with the assignment operator (=).

The scope of the variables used in the EVAL command is defined by using the QUAL command. However, you do not need to specifically define the scope of the variables contained in a CL module because they are all of global scope.

You can use the EVAL debug command to assign numeric, character, and hexadecimal data to variables provided they match the definition of the variable.

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 or literal value that you want to assign to *variable-name*. For example,

```
EVAL &COUNTER = 3.0
```

changes the value of *&COUNTER*; to 3.0 and shows

```
&COUNTER = 3.0 = 3.0
```

on the message line of the Display Module Source display. The result is preceded by the variable-name and value you are changing.

When you assign values to a character variable, the following rules apply:

- If the length of the source expression is less than the length of the target expression, the data is left justified in the target expression and the remaining positions are filled with blanks.
- If the length of the source expression is greater than the length of the target expression, the data is left justified in the target expression and truncated to the length of the target expression.

Note: DBCS variables can be assigned any of the following:

- Another DBCS variable
- A graphic literal of the form G‘<so>DBCS data<si>’
- A hexadecimal literal of the form X'hex digits'

Change logical variable examples

```
CL declarations:   DCL    VAR(&LGL1) TYPE(*LGL) VALUE('1')
                   DCL    VAR(&LGL2) TYPE(*LGL)


Debug command:     EVAL &LGL1

Result:            &LGL1 = '1'


Debug command:     EVAL &LGL1 = X'F0'

Result:            &LGL1 = X'F0' = '0'


Debug command:     EVAL &LGL2 = &LGL1

Result:            &LGL2 = &LGL1 = '0'
```

Change character variable examples

```
CL declarations:   DCL    VAR(&CHAR1) TYPE(*CHAR) LEN(1) VALUE('A')
                   DCL    VAR(&CHAR2) TYPE(*CHAR) LEN(10)


Debug command:     EVAL &CHAR1 = 'B'

Result:            &CHAR1 = 'B' = 'B'


Debug command:     EVAL &CHAR1 = X'F0F1F2F3'

Result:            &CHAR1 = 'F0F1F2F3' = '0'


Debug command:     EVAL &CHAR2 = 'ABC'

Result:            &CHAR2 = 'ABC' = 'ABC       '


Debug command:     EVAL %SUBSTR(CHAR2 1 2) = %SUBSTR(&CHAR2 3 1)

Result:            %SUBSTR(CHAR2 1 2) = %SUBSTR(&CHAR2 3 1) = 'C '

Comment:           Variable &CHAR contains 'C C       '
```

The %SUBSTR built-in function allows you to substring a character string variable. The first argument must be a string identifier, the second argument is the starting position, and the third argument is the number of single byte or double byte characters. Arguments are delimited by one or more spaces.

Change decimal variable examples

```
CL declarations:    DCL    VAR(&DEC1) TYPE(*DEC) LEN(3 1) VALUE(73.1)
                    DCL    VAR(&DEC2) TYPE(*DEC) LEN(2 1) VALUE(3.1)


Debug command:     EVAL &DEC1 = 12.3

Result:            &DEC1 = 12.3 = 12.3


Debug command:     EVAL &DEC1 = &DEC2

Result:            &DEC1 = &DEC2 = 03.1
```

Attributes of a Variable Examples

The Attribute (ATTR) debug command permits you to display the attributes of a variable. The attributes are the size (in bytes) and type of the variable as recorded in the debug symbol table on Table 10 on page 358.

The following is an example using the ATTR debug command.

```
CL declaration:   DCL    VAR(&CHAR2) TYPE(*CHAR) LEN(10)

Debug command:     ATTR &CHAR2

Result:            TYPE = FIXED LENGTH STRING, LENGTH = 10 BYTES


CL declaration:   DCL    VAR(&DEC) TYPE(*DEC) LEN(3 1)

Debug command:     ATTR &DEC

Result:            TYPE = PACKED(3,1), LENGTH = 2 BYTES
```

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* that 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.

Source Debug National Language Support for ILE CL

The following conditions exist when you are working with source debug National Language Support for ILE CL.

- When a view is displayed on the Display Module Source display, the source debugger converts all data to the Coded Character Set Identifier (CCSID) of the debug job.
- When assigning literals to variables, the source debugger does not perform CCSID conversion on quoted literals (for example 'abc'). Also, quoted literals are case sensitive.

Working with *SOURCE View

The following condition is true only when you are working with the CL Root Source View:

- If the source file CCSID is **different** from the module CCSID, the source debugger may not recognize a CL identifier containing variant characters (#, @, $)

The CCSID of the module can be found using the Display Module (DSPMOD) CL command. If you need to work with CL Root Source View and the source file CCSID is different from the module CCSID, you can take one of the following actions:

- Ensure the CCSID of CL source is the same as the CCSID of the compile-time job.
- Change the CCSID of the compile-time job to 65 535 and compile.
- Use the CL Listing View if the previous two options are not possible.

See the ILE Concepts book, Chapter 10, "Debugging Considerations" for more information on national language Restrictions for Debugging.

Using COPY, SAVE, RESTORE, CRTDUPOBJ, and CHKOBJITG while Debugging

Breakpoints or steps may be **temporarily removed** from a program while debugging when you use certain control language (CL) commands to specify your library or program. Breakpoints and steps are *restored* when the CL command completes running. A CPD190A message will be in the job log when the breakpoints or steps are removed; another CPD190A message will be in the job log when the breakpoints or steps are restored.

The following are the CL commands which can cause the breakpoint or step to be temporarily removed:

CHKOBJITG	CPY	CPROBJ	RSTLIB	SAVLIB
	CPYLIB	CRTDUPOBJ	RSTOBJ	SAVOBJ
				SAVSYS
				SAVCHGOBJ

Note: When the CL commands are operating on the program, you will receive error message CPF7102 when you issue the BREAK or STEP command.

Debugging OPM Programs

Testing functions are designed to help you write and maintain your applications. It lets you run your programs in a special testing environment while closely observing and controlling the processing of these programs in the testing environment. You can interact with your programs using the testing functions described in this chapter. These functions are available through a set of commands that can be used interactively or in a batch job. The functions allow you to:

- Trace a program's processing sequence and show the statements processed and the values of program variables at each point in the sequence.
- Stop at any statement in a program (called a breakpoint) and receive control to perform a function such as displaying or changing a variable value or calling another user-defined program.

No special commands specifically for testing are contained in the program being tested. The same program being tested can be run normally without changes. All test commands are specified within the job the program is in, not as a permanent part of the program being tested. With the testing commands, you interact with the programs symbolically in the same terms as the high-level language (HLL) program was written in. You refer to variables by their names and statements by their numbers. (These are the numbers used in the program's source list.) In addition, the test functions are only applicable to the job they are set up in. The same program can be used at the same time in another job without being affected by the testing functions set up.

Debug Mode

To begin testing, your program must be put in debug mode. Debug mode is a special environment in which the testing functions can be used in addition to the normal system functions. Testing functions cannot be used outside debug mode. To start debug mode, you must use the Start Debug (STRDBG) command. In addition to placing your program in debug mode, the STRDBG command lets you specify certain testing information such as the programs that are being debugged. Your program remains in debug mode until an End Debug (ENDDBG) or Remove Program (RMVPGM) command is encountered or your current routing step ends.

Note: If the System Debug Manager function in iSeries Navigator has been used to select Debug in order to check the system, then issuing the Start Debug (STRDBG) command will cause the graphical interface to be presented. In this case, whenever one of the users specified in the user list issues the STRDBG command, they will see the iSeries Navigator System Debug Manager rather than the Command Entry display. If the ENDDBG command is entered and *Debug* in the System Debug Manager is not currently selected, then issuing the STRDBG command will again start the system debugger using the Command Entry display.

The following STRDBG command places the job in debug mode and adds program CUS310 as the program to be debugged.

```
STRDBG  PGM(CUS310)
```

| You can use the ILE source debugger to debug OPM programs. To create OPM
| programs that contain source debug data, specify the OPTION(*SRCDBG)
| parameter or the OPTION(*LSTDBG) parameter on the Create CL Program
| (CRTCLPGM) command. The source debug data is actually part of the program
| object.

To add OPM programs that get created containing source debug data to the ILE source debugger, use the Program (PGM) and OPM Source Level Debug (OPMSRC) parameters on the STRDBG command. To start a debug session with an OPM program created with source debug data, type:

```
STRDBG PGM(*LIBL/MYOPMPGM) OPMSRC(*YES) DSPMODSRC(*YES)
```

For more information on ILE source debugging , see "Debugging ILE Programs" on page 357.

Adding Programs to Debug Mode

Any program can be run in debug mode, but before you can debug it, you must put it in debug mode. You can place a program in debug mode by specifying it in the PGM parameter on the STRDBG command or by adding it to the debugging session with an Add Program (ADDPGM) command. You can specify as many as twenty (20) programs to be debugged simultaneously in a job. You must have *CHANGE authority to add a program to debug mode.

If you specified twenty (20) programs for debug mode (using either the STRDBG or ADDPGM command or both commands) and you want to add more programs to the debug job, you must remove some of the previously specified programs. Use the Remove Program (RMVPGM) command. When debug mode ends, all programs are automatically removed from debug mode.

When you start debug mode, you can specify that a program be a default program. By specifying a default program, you can use any debug command that has the PGM parameter without having to specify a program name each time a command is used. This is helpful if you are only debugging one program. For example, in the Add Breakpoint (ADDBKP) command, you would not specify a program name on the PGM parameter because the default program is assumed to be the program the breakpoint is being added to. The default program name must be specified in the list of programs to be debugged (PGM parameter). If more than one program is listed to be debugged, you can specify the default program on the DFTPGM parameter. If you do not, the first program in the list on the PGM parameter on the STRDBG command is assumed to be the default program.

The default program can be changed any time during testing by using either the Change Debug (CHGDBG) or the ADDPGM command.

Note: If a program that is in debug mode is deleted, re-created, or saved with storage freed, references made to that program (except a RMVPGM command) may result in a function check. You must either remove the program using a RMVPGM command or end debug mode using an ENDDBG command. If you want to change the program and then debug it, you must remove it from debug mode and after it is re-created, add it to debug mode (ADDPGM command).

Preventing Updates to Database Files in Production Libraries

You can use files in production libraries while you are in debug mode. To prevent database files in production libraries from being unintentionally changed, you can specify UPDPROD(*NO) or default to *NO on the STRDBG command. Then, only files in test libraries can be opened for updating or adding new records. If you want to open database files in production libraries for updating or adding new records or if you want to delete members from production physical files, you can specify UPDPROD(*YES).

You can use this function with the library list. In the library list for your debug job, you can place a test library before a production library. You should have copies of the production files that might be updated by the program being debugged in the test library. Then, when the program runs, it uses the files in the test library. Therefore, production files cannot be unintentionally updated.

The Call Stack

You can use the Display Debug (DSPDBG) command to display the call stack, which indicates:

- Which programs are currently being debugged
- The instruction number of the calling instruction or the instruction number of each breakpoint at which program processing is stopped
- The program recursion level
- The names of the programs that are in debug mode but have not been called

A call of a program is the allocation of *automatic* storage for the program and the transfer of machine processing to the program. A series of calls is placed in a call stack. When a program finishes processing or transfers control, it is removed from the call stack. For more information about the call stack, see Chapter 3.

A program may be called a number of times while the first call is still in the call stack. Each call of a program is a recursion level of the program.

When a call is ended (the program returns or transfers control), automatic storage is returned to the system.

Notes:

1. CL programs can be recursive; that is, a CL program can call itself either directly or indirectly through a program it has called.
2. Some high-level languages do not allow recursive program calls. Other languages allow not only programs to be recursive, but also procedures within a program to be recursive. (In this guide, the term *recursion level* refers to the number of times the program has been called in the call stack. A procedure's recursion level is referred to explicitly as the procedure recursion level.)
3. All CL commands and displays make use of only the program qualified name recursion level.

Program Activations

An activation of a program is the allocation of static storage for the program. An activation is always ended when one of the following happens:

- The current routing step ends.
- The request that activated the program is canceled.
- The Reclaim Resources (RCLRSC) command is run such that the last (or only) call of a program is ended.

In addition, an activation can be destroyed by actions taken during a program call. These actions are dependent on the language (HLL or CL) in which the program is written.

When a program is deactivated, static storage is returned to the system. The language (HLL or CL) in which the program is written determines when the program is normally deactivated. A CL program is always deactivated when the program ends.

An RPG/400® program is deactivated when the last record indicator (LR) is set on before the program ends. If there is a return operation and LR is off, the program is not deactivated.

Handling Unmonitored Messages

Normally, if a program receives an unmonitored escape message, the system sends the function check message (CPF9999) to the program's program message queue and the program stops processing. However, HLL program compilers may insert monitors for the function check message or for messages that may occur in the

program. (An inquiry message is sent to the program messages display.) This allows you to end the program the way you want. In an interactive debug job, when a function check occurs, the system provides default handling and gives you control instead of stopping the program. The system displays the following on the unmonitored message display:

- The message
- The MI instruction number and HLL statement identifier, if available, to which the message was sent
- The name and recursion level of the program to which the message was sent

The following is an example of an unmonitored message breakpoint display:

```
                Display Unmonitored Message Breakpoint

 Statement/Instruction . . . . . :   440 /0077
 Program . . . . . :   TETEST
 Recursion level . . . . . :   1

Errors occurred on command.







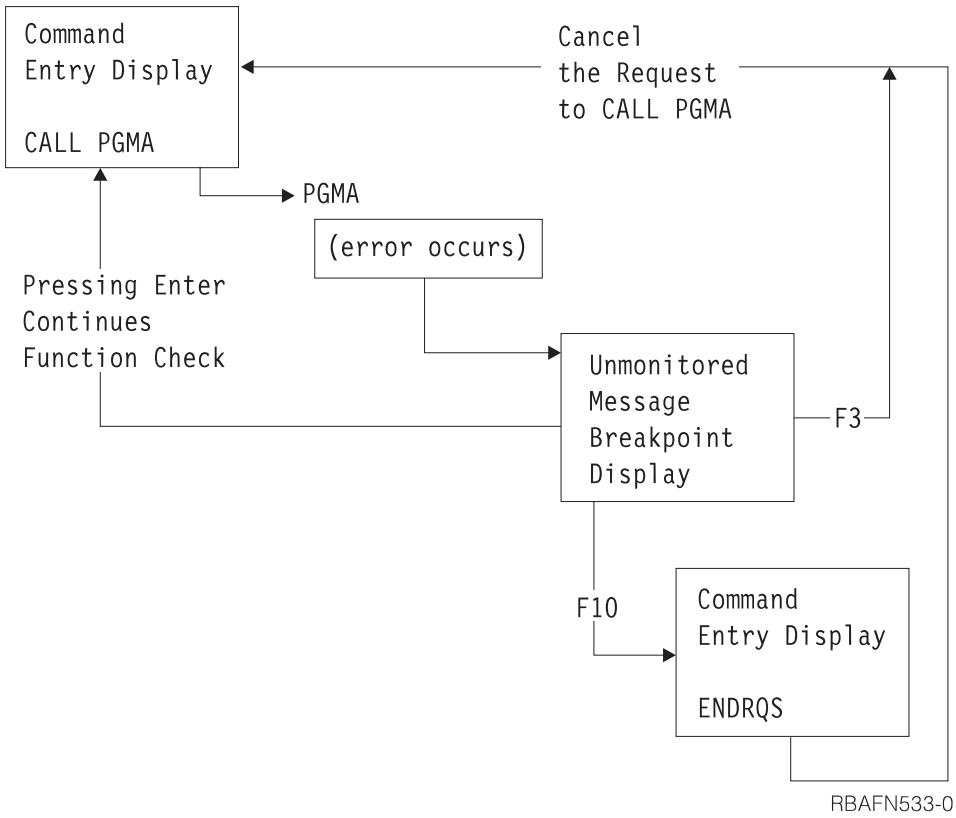


Press Enter to continue.

F3=Exit program   F10=Command Entry
```

You can try to isolate the source of the error by using the testing functions. However, the original request in error is still stopped at the point where the error occurred. To remove the request in error from the call stack, you must use the End Request (ENDRQS) command or press F3 when the unmonitored message breakpoint display is shown. You can let the usual function check processing continue by pressing the Enter key when the unmonitored message breakpoint display is shown. If you press F10 to call the command entry display, you must press F3 to return to the unmonitored message breakpoint display.

The following shows how a ENDRQS command works:



Program calls are destroyed when a ENDRQS command is entered. (In the previous diagram, the program call of PGMA is destroyed.)

Breakpoints

A breakpoint is a place in a program at which the system stops program processing and gives control to you at a display station (interactive mode) or to a program specified on the BKPPGM parameter in the Add Breakpoint (ADDBKP) command (batch mode).

Adding Breakpoints to Programs

Use the ADDBKP command to add breakpoints to the program you want debugged. You can specify up to 10 statement identifiers on the one ADDBKP command. The program variables specified on an ADDBKP command apply only to the breakpoints specified on that command. Up to 10 variables can be specified in one ADDBKP command.

You can also specify the name of the program to which the breakpoint is to be added. If you do not specify the name of the program that you want the breakpoint added to, the breakpoint is added to the default program specified on the STRDBG, CHGDBG, or ADDPGM command.

For more information about breakpoint commands, see the *CL* section of the **Programming** category of the iSeries Information Center.

To add a breakpoint to a program, specify a statement identifier, which can be:

- A statement label
- A statement number
- A machine interface (MI) instruction number

When you add a breakpoint to a program, you can also specify program variables whose values or partial values you want to display when the breakpoint is reached. These variables can be shown in character or hexadecimal format.

Program processing stops at a breakpoint *before* the instruction is processed. For an interactive job, the system displays what breakpoint the program has stopped at and, if requested, the values of the program variables.

In high-level language programs, different statements and labels may be mapped to the same internal instruction. This happens when there are several inoperable statements (such as DO and ENDDO) following one another in a program. You can use the IRP list to determine which statements or labels are mapped to the same instruction.

The result of different statements being mapped to the same instruction is that a breakpoint being added may redefine a previous breakpoint that was added for a different statement. When this occurs, a new breakpoint replaces the previously added breakpoint, that is, the previous breakpoint is removed and the new breakpoint is added. After this information is displayed, you can do any of the following:

- End the most recent request by pressing F3.
- Continue program processing by pressing Enter.
- Go to the command entry display at the next request level by pressing F10. From this display, you can:
  - Enter any CL command that can be used in an interactive debug environment. You may display or change the values of variables in your program, add or remove programs from debug mode, or perform other debug commands.
    - Continue processing the program by entering the Resume Breakpoint (RSMBKP) command.
    - Return to the breakpoint display by pressing F3.
    - Return to the command entry display at the previous request level by entering the End Request (ENDRQS) command.

For a batch job, a breakpoint program can be called when a breakpoint is reached. You must create this breakpoint program to handle the breakpoint information. The breakpoint information is passed to the breakpoint program. The breakpoint program is another program such as a CL program that can contain the same commands (requests for function) that you would have entered interactively for an interactive job. For example, the program can display and change variables or add and remove breakpoints. Any function valid in a batch job can be requested. When the breakpoint program completes processing, the program being debugged continues.

A message is recorded in the job log for every breakpoint for the debug job.

The following ADDBKP commands add breakpoints to the program CUS310. CUS310 is the default program, so it does not have to be specified. The value of the variable &ARBAL is shown when the second breakpoint is reached.

```
ADDBKP  STMT(900)
ADDBKP  STMT(2200)  PGMVAR('&ARBAL')
```

Note: A CL variable must be entered with surrounding apostrophes.

The source for CUS310 looks like this:

```
5728PW1 R01M00  880101                 SEU SOURCE LISTING

SOURCE FILE . . . . . .  QGPL/QCLSRC
MEMBER  . . . . . . . .  CUS310

SEQNBR*...+... 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8 ...
  100 PGM   PARM(&NBRITEMS &ITEMPRC &PARBAL &PTOTBAL)
  200   DCL VAR(&PARBAL)   TYPE(*DEC) LEN(15 5) /* INPUT AREA INV BALANCE */
  300   DCL VAR(&PTOTBAL)  TYPE(*DEC) LEN(15 5) /* INPUT TOTAL INV BALANCE*/
  400   DCL VAR(&NBRITEMS) TYPE(*DEC) LEN(15 5) /* NUMBER OF ITEMS        */
  500   DCL VAR(&ITEMPRC)  TYPE(*DEC) LEN(15 5) /* PRICE OF THE ITEM      */
  600   DCL VAR(&ARBAL)    TYPE(*DEC) LEN(5 2)  /* AREA INVENTORY BALANCE */
  700   DCL VAR(&TOTBAL)   TYPE(*DEC) LEN(5 2)  /* TOTAL INVENTORY BALANCE*/
  800   DCL VAR(&TOTITEM)  TYPE(*DEC) LEN(5 2)  /* TOTAL PRICE OF ITEMS   */
  900   CHGVAR     VAR(&ARBAL) VALUE(&PARBAL)
 1000   CHGVAR     VAR(&TOTBAL) VALUE(&PTOTBAL)
 1100   IF COND(&NBRITEMS *EQ 0) THEN(DO)
 1200      SNDPGMMSG  MSG('The number of items is zero.  This item +
 1300                   should be ordered.') TOMSGQ(INVLIB/INVQUEUE)
 1400      GOTO       CMDLBL(EXIT)
 1500   ENDDO
 1600   CHGVAR     VAR(&TOTITEM) VALUE(&NBRITEMS * &ITEMPRC)
 1700   IF COND(&NBRITEMS *GT 50) THEN(DO)
 1800      SNDPGMMSG  MSG('Too much inventory for this item.') +
 1900                 TOMSGQ(INVLIB/INVQUEUE)
 2000   ENDDO
 2100   CHGVAR     VAR(&ARBAL)   VALUE(&ARBAL + &TOTITEM)
 2200   IF COND(&ARBAL *GT 1000) THEN(DO)
 2300      SNDPGMMSG  MSG('The area has too much money in +
 2400                   inventory.') TOMSGQ(INVLIB/INVQUEUE)
 2500   ENDDO
 2600   CHGVAR     VAR(&TOTBAL)  VALUE(&TOTBAL + &TOTITEM)
 2700   EXIT:      ENDPGM
```

The following is displayed as a result of reaching the first breakpoint:

```
                        Display Breakpoint

Statement/Instruction . . . . . . . . . :   900 /0009
Program . . . . . . . . . . . . . . . . :   CUS310
Recursion level . . . . . . . . . . . . :   1

Press Enter to continue.

F3=Exit program   F10=Command entry
```

The following is displayed as a result of reaching the second breakpoint:

```
                        Display Breakpoint

Statement/Instruction . . . . . :   2200 /0022
Program . . . . . :   CUS310
Recursion level . . . . . :   1
Start position  . . . . . :   1
Format  . . . . . :   *CHAR
Length  . . . . . :   *DCL

Variable  . . . . . :   &ARBAL
  Type  . . . . . :     PACKED
  Length  . . . . . :     5 2
 '610.00'









Press Enter to continue.

F3=Exit program   F10=Command entry
```

The variable &ARBAL is shown. (Note that the value of &ARBAL will vary depending on the parameter values passed to the program.) You can press F10 to display the command entry display so that you could change the value of the variable &ARBAL to alter your program's processing. You use the Change Program Variable (CHGPGMVAR) command to change the value of a variable.

Conditional Breakpoints

You may add a conditional breakpoint to a program that is being debugged. Use the Add Breakpoint (ADDBKP) command to specify the statement and condition. If the condition is met, the system stops the program processing at the specified statement.

You may specify a skip value on the ADDBKP command. A **skip value** is a number that indicates how many times a statement should be processed before the system stops the program. For example, to stop a program at statement 1200 after the statement has been processed 100 times, enter the following command:

```
ADDBKP STMT(1200) SKIP(100)
```

If you specify multiple statements when the SKIP parameter is specified, each statement has a separate count. The following command causes your program to stop on statement 150 or 200, but only after the statement has processed 400® times:

```
ADDBKP STMT(150 200) SKIP(400)
```

If statement 150 has processed 400 times but statement 200 has processed only 300 times, then the program does not stop on statement 200.

If a statement has not processed as many times as was specified on the SKIP parameter, the Display Breakpoint (DSPBKP) command can be used to show how many times the statement was processed. To reset the SKIP count for a statement to zero, enter the breakpoint again for that statement.

You can specify a more general breakpoint condition on the ADDBKP command. This expression uses a program variable, an operator, and another variable or constant as the operands. For example, to stop a program at statement 1500 when variable &X is greater than 1000, enter the following command:

```
ADDBKP STMT(1500) PGMVAR('&X') BKPCOND(*PGMVAR1 *GT 1000)
```

The BKPCOND parameter requires three values:

- In the example, the first value specifies the first variable specified on the PGMVAR parameter. (To specify the third variable, you would use *PGMVAR3.)
- The second value must be an operator. For a list of all valid operators, see the *CL* section of the **Programming** category of the iSeries Information Center.
- The third value may be a constant or another variable. A constant may be a number, character string, or bit string, and must be the same type as the program variable specified in the first value.

The SKIP and BKPCOND parameters can be used together to specify a complex breakpoint condition. For example, to stop a program on statement 1000 after the statement has been processed 50 times **and** only when the character string &STR is TRUE, enter the following command:

```
ADDBKP STMT(1000) PGMVAR('&STR') SKIP(50)
       BKPCOND(*PGMVAR1 *EQ 'TRUE ')
```

Removing Breakpoints from Programs

To remove breakpoints from a program, use the Remove Breakpoint (RMVBKP) command. To remove a breakpoint you must specify the statement number of the statement for which the breakpoint has been defined.

Traces

A trace is the process of recording the sequence in which the statements in a program are processed. A trace differs from a breakpoint in that you are not given control during the trace. The system records the traced statements that were processed. However, the trace information is not automatically displayed when the program completes processing. You must request the display of trace information using the Display Trace Data (DSPTRCDTA) command. The display shows the sequence in which the statements were processed and, if requested, the values of the variables specified on the Add Trace (ADDTRC) command.

Adding Traces to Programs

Adding a trace consists of specifying what statements are to be traced and, if you want, the names of program variables. Before a traced statement processes, the value of the variable is recorded. Also, you can specify that the values of the variables are to be recorded only if they have changed from the last time a traced statement was processed. These variables can be displayed in character format or hexadecimal format.

To specify which statements are to be traced, you can specify:

- The statement identifier at which the trace is to start and the statement identifier at which the trace is to stop
- That all statements in the program are to be traced
- A single statement identifier of a statement to be traced

On the STRDBG or CHGDBG command, you can specify how many statement traces can be recorded for a job and what action the system should take when the maximum is reached. When the maximum is reached, the system performs one of the following actions (depending on what you specify):

- For an interactive job, either of the following can be done:
  - Stop the trace (*STOPTRC). Control is given to you (a breakpoint occurs), and you can remove some of the trace definitions (RMVTRC command), clear the

trace data (CLRTRCDTA command), or change the maximum (MAXTRC parameter on the CHGDBG command).
  - Continue the trace (*WRAP). Previously recorded trace data is overlaid with trace data recorded after this point.
- For a batch job, either of the following can be done:
  - Stop the trace (*STOPTRC). The trace definitions are removed and the program continues processing.
  - Continue the trace (*WRAP). Previously recorded trace data is overlaid with trace data recorded after this point.

You can change the maximum and the default action any time during the debug job using the Change Debug (CHGDBG) command. However, the change does not affect traces that have already been recorded.

You can only specify a total of five statement ranges for a single program at any one time, which is a total taken from all the Add Trace (ADDTRC) commands for the program. In addition, only 10 variables can be specified for each statement range.

In high-level language programs, different statements and labels may be mapped to the same internal instruction. This happens when there are several inoperable statements (such as DO, END) following one another in a program. You can use the IRP list to determine which statements or labels are mapped to the same instruction.

When you specify CL variables, you must enclose the & and the variable name in single apostrophes. For example:

```
ADDTRC PGMVAR('&IN01')
```

When you specify a statement range, the source statement number for the stop statement is ordinarily larger than the number for the start statement. Tracing, however, is performed with machine interface (MI) instructions, and some compilers (notably RPG/400) generate programs in which the order of MI instructions is not the same as the order of the source statements. Therefore, in some cases, the MI number of the stop statement may not be larger than the MI number of the start statement, and you will receive message CPF1982.

When you receive this message, you should do one of the following:

- Trace all statements in the program.
- Restrict a statement range to one specification.
- Use MI instruction numbers gotten from an intermediate representation of a program (IRP) list of the program. (See "Debugging at the Machine Interface Level" on page 396.)

The following Add Trace (ADDTRC) command adds a trace to the program CUS310. CUS310 is the default program, so it does not have to be specified. The value of the variable &TOTBAL is recorded only if its value changes between the times each traced statement is processed.

```
ADDTRC STMT((900 2700))  PGMVAR('&TOTBAL')  OUTVAR(*CHG)
```

The following displays result from this trace and are displayed using the Display Trace Data (DSPTRCDTA) command. Note that column headers are not supplied for all displays.

```
                          Display Trace Data

                Statement/
Program         Instruction          Recursion level      Sequence number
CUS310          900                        1                        1

 Start position  . . . . . . . . . . . . :   1
 Length  . . . . . . . . . . . . . . . . :   *DCL
 Format  . . . . . . . . . . . . . . . . :   *CHAR

 Variable  . . . . . . . . . . . . . . . :   &TOTBAL
   Type  . . . . . . . . . . . . . . . . :     PACKED
   Length  . . . . . . . . . . . . . . . :     5 2
  '    .00'

                Statement/
Program         Instruction          Recursion level      Sequence number
CUS310          1000                       1                        2
CUS310          1100                       1                        3  +

Press Enter to continue.

F3=Exit   F12=Cancel
```

```
                          Display Trace Data


 Start position  . . . . . . . . . . . . :   1
 Length  . . . . . . . . . . . . . . . . :   *DCL
 Format  . . . . . . . . . . . . . . . . :   *CHAR

*Variable  . . . . . . . . . . . . . . . :   &TOTBAL
   Type  . . . . . . . . . . . . . . . . :     PACKED
   Length  . . . . . . . . . . . . . . . :     5 2
  '   1.00'

                Statement/
Program         Instruction          Recursion level      Sequence number
CUS310          1600                       1                        4
CUS310          1700                       1                        5
CUS310          2100                       1                        6
CUS310          2200                       1                        7
CUS310          2600                       1                        8  +

Press Enter to continue.

F3=Exit   F12=Cancel
```

```
                          Display Trace Data

CUS310          2700                        1                          9

 Start position  . . . . . . . . . . . . . :   1
 Length  . . . . . . . . . . . . . . . . . :   *DCL
 Format  . . . . . . . . . . . . . . . . . :   *CHAR

*Variable  . . . . . . . . . . . . . . . . :   &TOTBAL
   Type  . . . . . . . . . . . . . . . . . :     PACKED
   Length  . . . . . . . . . . . . . . . . :     5 2
  '   2.00'









Press Enter to continue.

F3=Exit   F12=Cancel
```

Instruction Stepping

You can step through the instructions of a program by using the STRDBG or CHGDBG commands and setting the MAXTRC parameter to 1 and the TRCFULL parameter to *STOPTRC. When you specify a trace range (ADDTRC command) and the program processes an instruction within that range, a breakpoint display with an error message appears. If you press Enter, another breakpoint display with the same error message appears for the next instruction processed in the trace range. When tracing is completed, the trace data contains a list of the instructions traced. You can display this data by entering the Display Trace Data (DSPTRCDTA) command.

Using Breakpoints within Traces

Breakpoints can be used within a trace range. At a breakpoint within a trace, you can display the trace data (DSPTRCDTA command) to determine if you need to take some action. The trace data is recorded before the breakpoint occurs. The trace information contains the value of any variables *before* the statement was processed.

Removing Trace Information from the System

On the DSPTRCDTA command, you can specify whether the trace information is removed from the system or left on the system after the information is displayed. If you leave the trace information on the system, any other traces are added to it. The information remains on the system (unless removed) until the debug job ends or the ENDDBG command is submitted. You can also use the Clear Trace Data (CLRTRCDTA) command to remove trace information from the system.

Removing Traces from Programs

The Remove Trace (RMVTRC) command removes all or some of the ranges specified in one or more Add Trace (ADDTRC) commands. Removing a trace range consists of specifying the statement identifiers used on the RMVTRC command, or specifying that all ranges be removed.

You can use the STMT parameter on the RMVTRC command to specify:

- All HLL statements or machine instructions or both in the specified program are not to be traced regardless of how the trace was defined by the ADDTRC command.

- The start and stop trace location of the HLL statements or system instructions or both to be removed.

The RMVPGM and ENDDBG commands also remove traces, but they also remove the program from debug mode.

Display Functions

In debug mode, you can display testing information that lets you review how you have set up your debug job. You can display what programs are in debug mode and what breakpoints and traces have been defined for those programs. In addition, you can display the status of the programs in debug mode.

You can use the following commands to display testing information:

- Display Debug (DSPDBG), which displays the current call stack and the names of the programs that are in debug mode and indicates the following:
  - Which are stopped at a breakpoint
  - Which are currently called
  - The request level of those that are called
  - Debug options selected for the debug job
- Display Breakpoint (DSPBKP), which displays the locations of breakpoints that are currently defined in a program.
- Display Trace (DSPTRC), which displays the statements or statement ranges that are currently defined in a program.

Displaying the Values of Variables

When you are at a breakpoint, you can display the values of program variables. You can have this done automatically on the breakpoint display by specifying the variable names on the ADDBKP command, or you can enter the Display Program Variable (DSPPGMVAR) command at the breakpoint by pressing F10 to show the command entry display. Only 10 variables can be specified on one DSPPGMVAR command. For character and bit variables, you can tell the system to begin displaying the value of the variable starting at a certain position and for a specified length. Variables can be displayed in either character or hexadecimal format.

Notes:

1. If you specify an array variable, you can do one of the following:
   a. Specify the subscript values of the array element you want to display. The subscript values can either be integer values or the names of numeric variables in the program.
   b. Display the entire array by not entering any subscripts.
   c. Display a single-dimension cross-section of the array by specifying values for all subscripts except one, and an asterisk for that one subscript value.
2. Variable names can be specified as simple or qualified names, but must be placed between apostrophes. A qualified name can be specified in either of two ways:
   a. Variable names alternating with the special separator words OF or IN, ordered from lowest to highest qualification level. A blank must separate the variable name and the special separator word.
   b. Variable names separated by periods, ordered from highest to lowest qualification level.

The following DSPPGMVAR command displays the variable ARBAL used in the program CUS310. CUS310 is the default program, so it does not have to be specified. The entire value is to be displayed in character format.

```
DSPPGMVAR  PGMVAR('&ARBAL')
```

The resulting display looks like this:

```
                    Display Program Variables

Program . . . . . . . . . . . . . . . . :   CUS310
Recursion level . . . . . . . . . . . . :   1
Start position  . . . . . . . . . . . . :   1
Format  . . . . . . . . . . . . . . . . :   *CHAR
Length  . . . . . . . . . . . . . . . . :   *DCL

Variable  . . . . . . . . . . . . . . . :   &ARBAL
  Type  . . . . . . . . . . . . . . . . :   PACKED
  Length  . . . . . . . . . . . . . . . :   5 2
 '610.00'









Press Enter to continue.

F3=Exit   F12=Cancel
```

Some HLLs allow variables to be based on a user-specified pointer variable (HLL pointer). If you do not specify an explicit pointer for a based variable, the pointer specified in the HLL declaration (if any) is used. You must specify an explicit basing pointer if one was not specified in the HLL declaration for the based variable. The PGMVAR parameter allows you to specify up to five explicit basing pointers when referring to a based variable. When multiple basing pointers are specified, the first basing pointer specified is used to locate the second basing pointer, the second one is then used to locate the third, and so forth. The last pointer in the list of basing pointers is used to locate the primary variable.

Changing the Values of Variables

To change the value of a program variable, use the Change Program Variable (CHGPGMVAR), the Change HLL Pointer (CHGHLLPTR), or the Change Pointer (CHGPTR) command. Changing the value of a program variable consists of specifying the variable name and a value that is compatible with the data type of the variable. For example, if the variable is character type, you must enter a character value.

When changing the value of variables, you should be aware of whether the variable is an automatic variable or a static variable. The difference between the two is in the storage for the variables. For automatic variables, the storage is associated with the call of the program. Every time a program is called, a new copy of the variable is placed in automatic storage. A change to an automatic variable remains in effect only for the program call the change was made in.

Note: In some languages, the definition of a call is made at the procedure level and not just at the program level. For these languages, storage for automatic variables is associated with the call of the procedure. Every time a procedure is called, a new copy of the variable is gotten. A change to an

automatic variable remains in effect only while that procedure is called. Only the automatic variables in the most recent procedure call can be changed. The RCRLVL (recursion level) parameter on the commands applies only on a program basis and not on a procedure basis.

For static variables, the storage is associated with the activation. Only one copy of a static variable exists in storage no matter how many times a program is called. A change to a static variable remains in effect for the duration of the activation.

To determine if a program variable is a static or an automatic variable, request an intermediate representation of a program (IRP) list (*LIST and *XREF on the GENOPT parameter) when the program containing the variables is created.

When changing a variable that is an array, you must specify one element of the array. Consequently, you must specify the subscript values for the array element you want to change.

Using a Job to Debug Another Job

You may want to use a separate job to debug programs running in another job for one of the following reasons:

- Batch jobs can be debugged by an interactive job.
- An interactive job can be debugged from another interactive job. This allows one display to show debug information without interrupting the application program display.
- An interactive or batch job that is looping can be interrupted and put into debug mode.

Debugging Batch Jobs Submitted to a Job Queue

Using a separate job to debug another batch job submitted to the job queue allows you to put the batch job into debug mode and to set breakpoints and traces before the job starts to process. Use the following steps to debug batch jobs to be submitted to a job queue:

1. Submit the batch job using the Submit Job (SBMJOB) command or a program that automatically submits the job with HOLD(*YES).

   ```
   SBMJOB HOLD(*YES)
   ```

2. Determine the qualified job name (number/user/name) that is assigned to the job using the Work with Submitted Jobs (WRKSBMJOB) command or the Work with Job Queues (WRKJOBQ) command. The SBMJOB command also displays the name in a completion message when the command finishes processing.

   The WRKJOBQ (Work With Job Queue) command displays all the jobs waiting to start in a particular job queue. You can show the job name from this display by selecting option 5 for the job.

3. Enter the Start Service Job (STRSRVJOB) command from the display you plan to use to debug the batch job as follows:

   ```
   STRSRVJOB JOB(qualified-job-name)
   ```

4. Enter the STRDBG command and provide the names of all programs to be debugged. No other debug commands can be entered while the job is waiting on the job queue.
5. Use the Release Job Queue (RLSJOBQ) command to release the job queue. A display appears when the job is ready to start, indicating that you may begin debugging the job. Press F10 to show the Command Entry display.
6. Use the Command Entry display to enter any debug commands, such as the Add Breakpoint (ADDBKP) or Add Trace (ADDTRC) commands.

7. Press F3 to leave the Command Entry display, and then press Enter to start the batch job.
8. When the job stops at a breakpoint, you see the normal breakpoint display. When the job finishes, you cannot add breakpoints and traces, or display or change variables. However, you can display any trace data using the Display Trace Data (DSPTRCDTA) command.
9. If you wish to debug another batch job, first end debugging using the End Debug (ENDDBG) command and then end servicing the job using the End Servicing Job (ENDSRVJOB) command.

Debugging Batch Jobs Not Started from Job Queues

Some jobs started on the system are not submitted to a job queue. These jobs cannot be stopped before they start running but they can usually be debugged. To debug jobs not started from a job queue, do the following:

1. Rename the program that is called when the job starts. For example, if the job runs program CUST310, you can rename this program to CUST310DBG.
2. Create a small CL program with the same name as the original program (before the program was renamed). In the small CL program, use the Delay Job (DLYJOB) command to delay for one minute and then use the CALL command to call the renamed program.
3. Allow the batch job to start to force the CL program to be delayed for one minute.
4. Use the Work with Active Jobs (WRKACTJOB) command to find the batch job that is running. When the display appears, enter option 5 next to the job to obtain the qualified job name.
5. Enter the Start Service Job (STRSRVJOB) command as follows:

   ```
   STRSRVJOB JOB(qualified-job-name)
   ```

6. Enter STRDBG and any other debug commands, such as the Add Breakpoint (ADDBKP) or Add Trace (ADDTRC) command. Proceed with debugging as usual.

Debugging a Running Job

You can debug a job that is already running if you know what statements the job will run. For example, you may want to debug a running program if the job is looping or the job has not yet run a program that is to be debugged. The following steps allow you to debug a running job:

1. Use the Work with Active Jobs (WRKACTJOB) command to find the job that is running. When the display appears, enter option 5 next to the job to obtain the qualified job name.
2. Enter the Start Service Job (STRSRVJOB) command as follows:

   ```
   STRSRVJOB JOB(qualified-job-name)
   ```

3. Enter the Start Debug (STRDBG) command. (Entering the command does not stop the job from running.)

   Note: You can use the Display Debug (DSPDBG) command to show the call stack. However, unless the program is stopped for some reason, the stack is correct only for an instant, and the program continues to run.
4. If you know a statement to be run, enter the Add Breakpoint (ADDBKP) command to stop the job at the statement.

   If you do not know what statements are being run, do the following:

   a. Enter the Add Trace (ADDTRC) command.

b. After a short time, enter the Remove Trace (RMVTRC) command to stop tracing the program.
c. Enter the Display Trace Data (DSPTRCDTA) command to show what statements have processed. Use the trace data to determine which data statements to process next (for example, statements inside a program loop).
d. Enter the Add Breakpoint (ADDBKP) command to stop the job at the statement.

5. Enter the desired debug commands when the program is stopped at a breakpoint.

Debugging Another Interactive Job

You can debug a job from another display, whether the job is running or waiting at a menu or command entry display. To debug another interactive job, do the following:

1. Determine the qualified job name of the job to be debugged. To determine the name, either enter the Display Job (DSPJOB) command from the display of the job to be debugged, or use the Work with Active Jobs (WRKACTJOB) command.
2. Enter the Start Service Job (STRSRVJOB) command using the qualified job name.
3. Enter the Start Debug (STRDBG) command and any other debug commands desired. If the job is already running, you may need to enter the Display Debug (DSPDBG) command to determine what statement in the program is processing.

When the job being debugged is stopped at a breakpoint, the display station is locked.

Considerations When Debugging One Job from Another Job

Although most jobs can be debugged from another job, you must take the following into consideration:

- A job being debugged cannot be held or suspended (for example, when running another group job or a secondary job).
- When servicing another job with the Start Service Job (STRSRVJOB) command, you cannot also debug the job doing the servicing. All debug commands apply only to the job being serviced. To debug the job doing the servicing, you must either end the servicing of the other job, or have another job service and debug it.
- Debug commands operate on another job, even if that job is not stopped at a breakpoint. For example, if you are debugging a running job and you enter the Display Program Variable (DSPPGMVAR) command, the variable you specify is shown. Since the job continues to run, the value of the variable may change soon after the command is entered.
- A job being debugged must have enough priority to respond to debug commands. If you are debugging a batch job with a low priority and that job gets no processing time, then any debug command you issue waits for a response from the job. If the job does not respond, the command ends and an error message is displayed.
- You cannot service and debug a job that is debugging itself. However, you can service and debug a job that is servicing and debugging another job.

Debugging at the Machine Interface Level

To debug your programs at the machine interface (MI) level, you can specify an MI object definition vector (ODV) number for the PGMVAR parameter of a command and MI instruction numbers for the STMT parameter of a command. For a breakpoint, the system stops at the MI instruction number just as it would at an HLL statement number. You must always precede the ODV or MI instruction number with a slash (/) and enclose it in apostrophes (for example, `'/1A'`) to signal to the system that you are debugging at the MI level.

The ODV and MI instruction numbers can be obtained from the IRP listing produced by most high-level language compilers. Use the *LIST value of the GENOPT parameter to produce the IRP listing at program creation time.

Note: When you debug at the machine interface level, only the characteristics that are defined at the machine interface level are available; the HLL characteristics that are normally passed to the test environment are not available. These HLL characteristics may include: the variable type, number of fractional digits, length, and array information. For example, a numeric variable in your HLL program may be displayed without the correct decimal alignment or possibly as a character string.

Security Considerations

To debug a program, you must have *CHANGE authority to that program. The *CHANGE authority available by adopting another user's profile is not considered when determining whether a user has authority to debug a program. This prevents users from accessing program data in debug mode by adopting another user's profile.

Additionally, when you are at a user-defined breakpoint of a program that you are debugging with adopted user authority, you have only the authority of your user profile and not the adopted profile authority. You do not have authorities adopted by prior program calls for all breakpoints whether they are added by the Add Breakpoint (ADDBKP) command or are caused by an unmonitored escape message.

Using COPY, SAVE, RESTORE, CRTDUPOBJ, and CHKOBJITG while Debugging

Breakpoints or statement traces may be **temporarily removed** from a program while the debug function is running if you use certain control language (CL) commands to specify your library or program. Breakpoints and statement traces are *restored* when the CL command completes running. A CPD190A message is in the job log when the breakpoints or traces are removed; another CPD190A message is in the job log when the breakpoints and statement traces are restored.

Breakpoints or statement traces may be **temporarily removed** from a program when you use the following CL commands to specify your library:

CHKOBJITG	CPY	CPROBJ	RSTLIB	SAVLIB
	CPYLIB	CRTDUPOBJ	RSTOBJ	SAVOBJ
				SAVSYS
				SAVCHGOBJ

Note: When the CL commands are running on your program, you may not be able to add breakpoints or add traces to the program. If you enter the Add Breakpoint (ADDBKP) command or the Add Trace (ADDTRC) command

when any of the commands are running on your program, you will receive error message CPF7102.

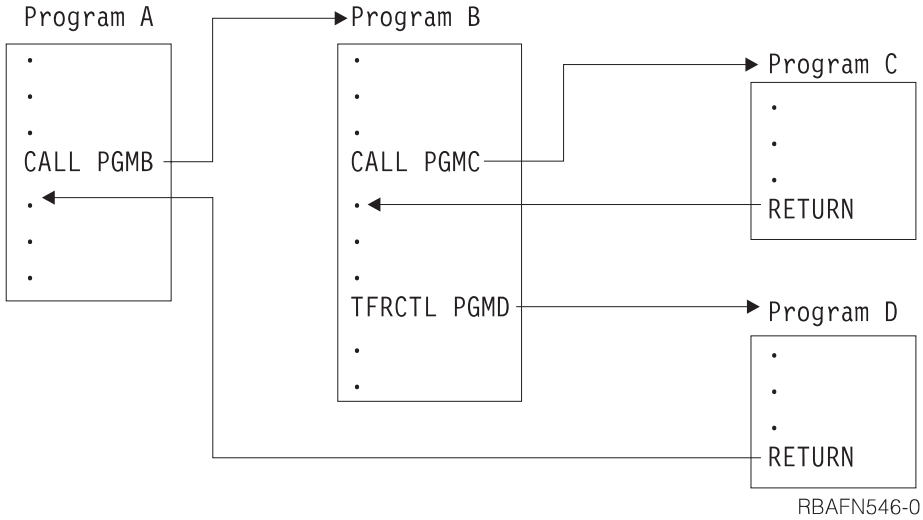
Appendix A. TFRCTL Command

The Transfer Control (TFRCTL) command calls the program specified on the command, passes control to it, and removes the transferring program from the call stack. Reducing the number of programs on the call stack can have a performance benefit. When a CALL command is used, the program called returns control to the program containing the CALL command. When a TFRCTL command is used, control returns to the first program in the call stack. The first program then initiates the next sequential instruction following the CALL command.

Note: The TRFCTL command is not valid in ILE CL procedures.

Using the TFRCTL Command

In the following illustration, if Program A is specified with USRPRF(*OWNER), the owner's authorities are in effect for all of the programs shown. If Program B is specified with USRPRF(*OWNER), the owner's authorities are in effect only while Programs B and C are active. When Program B transfers control to Program D, Program B is no longer in the call stack and the owner of Program B is no longer considered for authorization during the running of Program D. When the programs complete processing (by returning or transferring control), the owner's authorities are no longer in effect. Any overrides issued by Program B remain in effect while Program D is running and are lost when Program D does a return.



The TFRCTL command has the following format:

```
TFRCTL PGM(library-name/program-name) PARM(CL-variable)
```

The program (and library qualifier) may be a variable.

It is important to note that only variables may be used as parameter arguments on this command, and that those variables must have been received as a parameter in the argument list from the program that called the transferring program. That is, the TFRCTL command cannot pass a variable that was not passed to the program running the TFRCTL command.

© Copyright IBM Corp. 1997, 2004

In the following example, the first TRFCTL is valid. The second TFRCTL command is not valid because &B was not passed to this program. The third TFRCTL command is not valid because a constant cannot be specified as an argument.

```
PGM PARM(&A)
DCL &A *DEC 3
DCL &B *CHAR 10
IF (&A *GT 100) THEN (TFRCTL PGM(PGMA) PARM(&A)) /*   valid   */
IF (&A *GT 50) THEN (TFRCTL PGM(PGMB) PARM(&B))  /* not valid */
ELSE (TFRCTL PGM(PGMC) PARM('1'))                /* not valid */
ENDPGM
```

The PARM parameters are discussed under "Passing Parameters between Programs and Procedures" on page 72.

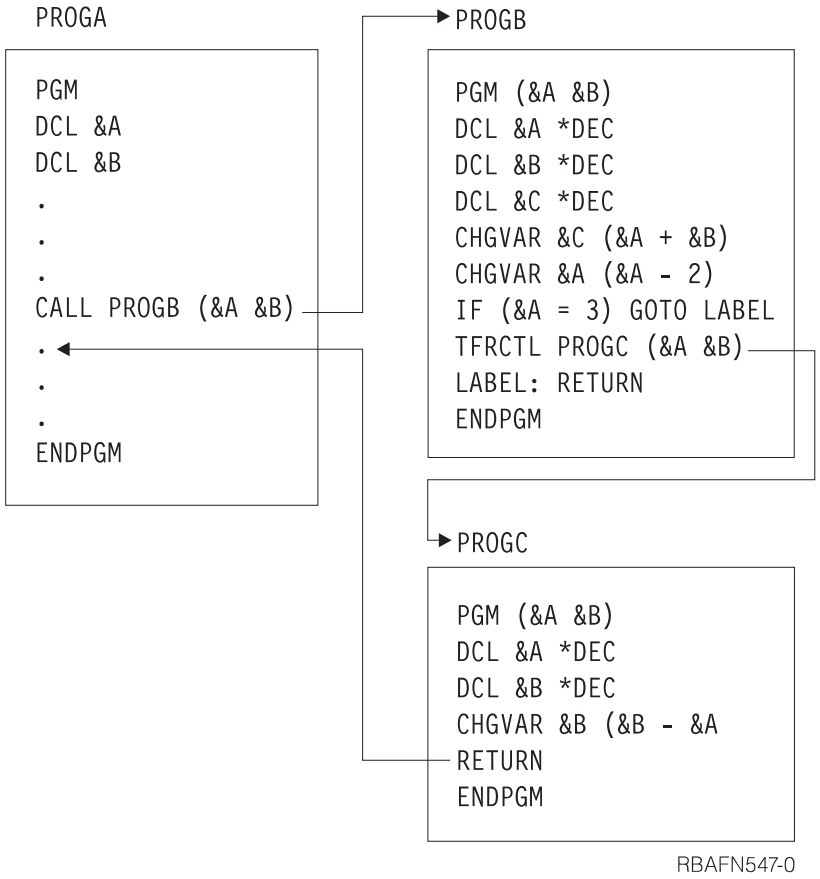
Passing Parameters

The TFRCTL command can be used to pass parameters to the program being called in the same way the CALL command passes parameters, but with these restrictions:

- The parameters passed must be CL variables.
- The CL variables passed by the transferring program must have been received as parameters by that program.
- This command is valid only within OPM CL programs.

In the following example, PROGA calls PROGB and passes two variables, &A and &B, to it. PROGB uses these two variables and another internally declared variable, &C. When control is transferred to PROGC, only &A and &B can be passed to PROGC. When PROGC finishes processing, control is returned to PROGA, where

these variables originated.



Appendix B. Job Log Output Files

Directing a Job Log

You can direct the job log for a job to one or two database files with the Control Job Log (QMHCTLJL) API or the Display Job Log (DSPJOBLOG) command. The first file is the primary job log file. This file contains the essential information for a message, such as, message ID, message type, and message severity. One record is produced in the primary job log file for each message selected for processing. The second file is the secondary job log file. The production of this file is only possible by using QMHCTLJL API; however, it is also optional.

The secondary job log file contains the first and second level text for a message. The text is in print format. Any message data is merged with the message description and the result is formatted into one or more print lines. For each message selected for processing there can be more than one record in the secondary job log file; one record for each first and second level print line.

Records in the primary file can be related to records in the secondary file through use of the Message Reference Key. Each record placed in the primary file contains a field that is the Message Reference Key (MRK) of the related message. Similarly, each secondary file record contains the MRK of the related message. The MRK for a message is unique within the context of a job. Once the MRK of a primary file record is known, the related secondary records can be readily identified since only these secondary records will contain the same MRK value.

Model for the Primary Job Log

The IBM supplied model for the primary job log file is QAMHJLPR in library QSYS. The primary record format is QMHPFT. A detailed description of this format follows:

Field Order	Field Name	Data Type	Length in Bytes	Field Description
1	QMHJDT	DATE	10	Date job log created
2	QMHJTM	TIME	8	Time job log created
3	QMHMRK	CHAR	4	Message reference key
4	QMHTYP	CHAR	10	Message type
5	QMHSEV	BIN	4	Message severity
6	QMHMID	CHAR	7	Message ID
7	QMHDAT	DATE	10	Message sent date
8	QMHTIM	TIME	8	Message sent time
9	QMHMF	CHAR	20	Message file name
10	QMHRPY	CHAR	4	Reply reference key
11	QMHRQS	CHAR	1	Request Message Status
12	QMHSTY	CHAR	1	Sending program type
13	QMHRTY	CHAR	1	Receiving program type
14	QMHSSN	BIN	4	Number of statements for sending program
15	QMHRSN	BIN	4	Number of statements for receiving program
16	QMHCID	BIN	4	CCSID of the message data or immediate message

© Copyright IBM Corp. 1997, 2004

Field Order	Field Name	Data Type	Length in Bytes	Field Description
17	QMHPRL	CHAR	1	Message percolated indicator
18	QMHSPR	VAR CHAR	256 MAX	Sending procedure name
19	QMHSMD	CHAR	10	Sending module name
20	QMHSPG	CHAR	12	Sending program name
21	QMHSLB	CHAR	10	Sending library name
22	QMHSTM	CHAR	30	Statement number(s) for sending program
23	QMHRPR	VAR CHAR	256 MAX	Receiving procedure name
24	QMHRMD	CHAR	10	Receiving module name
25	QMHRPG	CHAR	10	Receiving program name
26	QMHRLB	CHAR	10	Receiving program library name
27	QMHRTM	CHAR	30	Statement number(s) for receiving program
28	QMHSYS	CHAR	8	System name
29	QMHJOB	CHAR	26	Qualified Job name
30	QMHMDT	VAR CHAR	3000 MAX	Message data or immediate message
31	QMHCSP	VAR CHAR	4096 MAX	Complete sending procedure name
32	QMHCRP	VAR CHAR	4096 MAX	Complete receiving procedure name
33	QMHLSP	VAR CHAR	6144 MAX	Long sending program name
34	QMHTID	CHAR	8	Thread
35	QMHMSC	ZONED	6,0	Microseconds
36	QMHFUS	CHAR	10	From user

The definition of the fields in this record are as follows:

QMHJDT
Date job log created; DATE(10)

The date the production of the job log began. The field is a date field in the database record. The format of the date is *ISO. A value in this date field is in the format yyyy-mm-dd. Each record produced for the same job log will have the same value in this field.

QMHJTM
Time job log create; TIME(8)

The time the production of the job log began. This field is defined as a time field in the database record. The format of the time is defined to be *ISO. A value in this time field is in the format hh.mm.ss. Each record produced for the same job log will have the same value in this field.

QMHMRK
Message reference key; CHAR(4)

The message reference key the related message had in the job message queue. The records are placed in the primary database file in strictly ascending sequence by message reference key. Within the set of records produced for a single job log, this field is unique for each record and thus can be used as a unique key for the record. If the records for two or more job logs are placed into the same member, the key may no longer be unique.

QMHTYP
Message type; CHAR(10)

The message type of the related message. One of the following special values will appear in this field:

***CMD** Commands that are logged from the execution of a CL program.

***COMP**
Completion message type.

***COPY**
Sender's copy message type.

***DIAG**
Diagnostic message type.

***ESCAPE**
Escape message type.

***INFO**
Information message type.

***INQ** Inquiry message type.

***NOTIFY**
Notify message type.

***RQS** Request message type.

***RPY** Reply message type.

QMHSEV
Message severity; BIN(4)

The severity the message has. This is a value from 0 through 99.

QMHMID
Message ID; CHAR(7)

The message ID for the message. This field will contain the special value *IMMED if the message is an immediate message which has no message ID.

QMHDAT
Message sent date; DATE(10)

The date the message was sent. This field is defined as a date field in the database record. The format of the date is *ISO. A value in this field is in the format yyyy-mm-dd.

QMHTIM
Message sent time; TIME(8)

The time the message was sent. The field is defined as a time field in the database record. The format of the time is defined to be *ISO. A value in this field is in the format hh.mm.ss.

QMHMF
Message File; CHAR(20)

The name of the message file that is to be used to obtain the message description for the message. The first 10 characters of the field contain the message file name. The second 10 characters contain the library name. If the field QMHMID contains *IMMED to indicate an immediate message, this field will contain all blanks.

QMHRPY
Reply reference key; CHAR(4)

- If the message type of the message is inquiry, notify, or sender's copy, this is the message reference key of the related reply message.
- If there is no reply message available this field will contain a null value ('00000000'X).
- If the message type is not inquiry, notify, or sender's copy, this field will also contain a null value.

In order to maintain the strictly ascending sequence by message reference key, the record for the reply message may not immediately follow the record for the inquiry, notify, or sender's copy message.

QMHRQS

Request Message Status; CHAR(1)

- If the message type is *RQS, this is an indicator which shows whether the request message was run or not.
- If the indicator is set to zero ('F0'X) the request was not run.
- If the indicator is set to one ('F1'X) the request was run.

If the messages type is not *RQS, this indicator will always be zero.

QMHSTY

Sending program type; CHAR(1)

An indicator with the following values that shows whether the sending program was an OPM program or an ILE program.

- If this indicator is set to zero ('F0'X), the sending program is an OPM or System Licensed Internal Code (SLIC) program with a name less than or equal to 12 characters. The program name is placed in fields QMHSPG and QMHLSP.
- If the indicator is set to one ('F1'X), the sending program is an ILE program with a procedure name less than or equal to 256 characters. The procedure name is placed in fields QMHSPR and QMHCSP.
- If the indicator is set to two ('F2'X), the sending program is an ILE program with a procedure name greater than 256 characters and up to 4096 characters. The complete sending sending procedure name is in field QMHCSP; field QMHSPR is blank.
- If the indicator is set to three ('F3'X), the sending program is a SLIC program with a name greater than 12 characters and up to 256 characters. The complete sending program name is in field QMHLSP; field QMHSPG is blank.

QMHRTY

Receiving program type; CHAR(1)

An indicator with the following values that shows the type of the receiving program:

- If this indicator is set to zero ('F0'X), the receiving program was an OPM program. The program name is placed in field QMHRPG.
- If the indicator is set to one ('F1'X), the receiving program was an ILE program with a procedure name less than or equal to 256 characters. The procedure name is placed in fields QMHRPR and QMHCRP.
- If the indicator is set to two ('F2'X), the receiving program is an ILE program with a procedure name greater than 256 and up to 4096 characters. The entire receiving procedure name is placed field QMHCRP; the field QMHRPR is blank.

QMHSSN
Number of statements for sending program; BIN(4)

The number of statement numbers for sending program.

- If the sending program type field QMHSTY contains a zero ('F0'X) or a three ('F3'X), this field contains a value of 0 or 1.
- If the sending program type field contains a one ('F1'X) or a two ('F2'X), this field can contain the value 0, 1, 2, or 3.

The value provided in this field defines how many statement numbers are in the field QMHSTM.

QMHRSN
Number of statements for receiving program; BIN(4)

The number of statement numbers for receiving program.

- If the receiving program type field QMHRTY contains a zero ('F0'X), this field contains a value of 0 or 1.
- If the receiving program type field contains a one ('F1'X) or a two ('F2'X), this field contains the value 0, 1, 2, or 3. The value provided in this field defines how many statement numbers are in the field QMHRTM.

QMHCID
CCSID of message data; BIN(4)

The CCSID of the message data or immediate message that is contained in the field QMHMDT.

QMHPRL
Message percolate indicator; CHAR(1)

An indicator that shows whether the message was percolated to the receiving program or not.

- If the message was not percolated this field contains a zero ('F0'X).
- If the message was sent this field contains a one ('F1'X).

Message percolation can only occur within an ILE program. Therefore, this field contains a one only if the receiving program type field QMHRTY contains a one ('F1'X) or a two ('F2'X).

QMHSPR
Sending procedure name; VAR CHAR(*)

- If the sending program type field QMHSTY contains a zero ('F0'X) or three ('F3'X), this field contains the value *N.
- If the sending program type field QMHSTY contains a one ('F1'X), this field contains the sending ILE procedure name. The name can be a maximum of 256 characters in length.
- If the sending program type field QMHSTY contains a two ('F2'X), this filed contains blanks, while the entire name will be contained in the field QMHCSP.

This field can contain a nested procedure name for a sending program type of one ('F1'X) or two ('F2'X); each procedure name is separated by a colon. The outer-most procedure name is identified first and is followed by the procedures contained in it. The inner-most procedures are identified last in the string.

QMHSMD
Sending module name; CHAR(10)

- If the sending program type field QMHSTY contains a zero ('F0'X) or a three ('F3'X), this field contains the value *N.
- If the sending program type field QMHSTY contains a one ('F1'X) or a two ('F2'X), this field contains the sending ILE module name.

QMHSPG
Sending program name; CHAR(12)

- If the sending program type field QMHSTY contains a zero ('F0'X), a one ('F1'X), or a two ('F2'X), the field contains the program name from which the message was sent.
- If the sending program type is a three ('F3'X), this field contains blanks and field QMHLSP contains the sending program name.

QMHSLB
Sending library name; CHAR(10)

The name of the library that the sending program was contained in.

QMHSTM
Statement number(s) for sending program; CHAR(30)

The statement number(s) at which the sending program sent the message. Each statement number is 10 characters in length.

- If the sending program type field QMHSTY contains a zero ('F0'X) or a three ('F3'X), there is, at most, one statement number in the first 10 characters. That statement number represents an MI instruction number. The number is a hexadecimal number.
- If the sending program type field contains a one ('F1'X) or a two ('F2'X), this field can contain statement numbers of 0, 1, 2, or 3. The field QMHSSN specifies how many there are. In this case, a statement number is a higher level language statement number and not an MI instruction number. Each number is a decimal number.

QMHRPR
Receiving procedure name; VAR CHAR(*)

- If the receiving program type field contains a zero ('F0'X), this field contains the value *N.
- If the Receiving program type field QMHRTY contains a one ('F1'X), this field contains the receiving ILE procedure name. The name can be a maximum of 256 characters in length.
- If the Receiving program type field QMHRTY contains a two ('F2'X), this field contains blanks, while the entire name will be contained in the field QMHCRP.

This field can contain a nested procedure name for a sending program type of 1 or 2; each procedure name is separated by a colon. The outer-most procedure name is identified first and is followed by the procedures contained in it. The inner-most procedures are identified last in the string.

QMHRMD
Receiving module name; CHAR(10)

- If the receiving program type field contains a zero ('0F'X), this field contains the value *N.
- If the receiving program type field QMHRTY contains a one ('F1'X) or a two ('F2'X), this field contains the receiving ILE module name.

QMHRPG
Receiving program name; CHAR(10)

The program name of the OPM or ILE program to which the message was sent.

QMHRLB
Receiving library name; CHAR(10)

The name of the library that the receiving program was in.

QMHRTM
Statement number(s) for receiving program; CHAR(30)

The statement number(s) at which the receiving program was stopped when the message was sent. Each statement number is 10 characters in length.

- If the receiving program type field QMHRTY contains a zero ('F0'X), there is, at most, one statement number in the first 10 characters. That statement number represents an MI instruction number. The number is a hexadecimal number.

For any other value of the receiving program type, there can be 0, 1, 2, or 3 statement numbers in this field. The field QMHRSN specifies how many there are. In this case, a statement number is a higher level language statement number and not an MI instruction number. Each number is a decimal number.

QMHSYS
System name; CHAR(8)

The name of the system that the job log was produced on.

QMHJOB
Qualified job Name; CHAR(26)

The fully qualified name of the job for which the message is being logged for. The first 10 positions contain the job name, the next 10 positions the user name, and the last six positions the job number.

QMHMDT
Message data; VAR CHAR(*)

If the field QMHMID contains the special value *IMMED, this field contains an immediate message. Otherwise, this field contains the message data that was used when the message was sent. This field can contain a maximum of 3000 characters. If the immediate message or message data is longer, it is truncated to 3000 characters.

If the message data contains pointers, the pointers is invalidated before the message data is written to the database file.

QMHCSP
Complete sending procedure name; CHAR(VAR)

- If the sending program type is zero ('F0'X) or three ('F3'X), this field contains blanks.
- If the sending program type is one ('F1'X) or two ('F2'X), this field contains the entire ILE procedure name. The name can be a maximum of 4096 characters in length.

  This field can contain a nested procedure name where each procedure name is separated by a colon. The outer-most procedure name is identified first and is followed by the procedures contained in it. The inner-most procedures are identified last in the string.

QMHCRP
Complete receiving procedure name; CHAR(VAR)

- If the sending program type is zero ('F0'X), this field contains blanks.
- If the receiving program type is one ('F1'X) or two ('F2'X), this field contains the entire ILE procedure name. The name can be a maximum of 4096 characters in length.

  The field can contain a nested procedure name where each procedure name is separated by a colon. The outer-most procedure name is identified first and is followed by the procedures contained in it. The inner-most procedures are identified last in the string.

QMHLSP
Long sending program name; CHAR(VAR)

This field contains the entire sending program name from which the message was sent for all sending program types. The name can be a maximum of 6144 characters in length.

QMHTID
Thread; CHAR(8)

This field identifies the thread within the job that sent the message.

QMHMSC
Microseconds; ZONED(6,0)

This is the microseconds part of the time the message was sent. It can be used to determine with more precision the time the message was sent.

QMHFUS
From user; CHAR(10)

The name of the user profile that the thread was running under when the message was sent.

The IBM supplied model for the secondary job log file is QAMHJLSC in library QSYS. The secondary record format name is QMHSFT. A detailed description of the secondary record format follows:

Field Order	Field Name	Data Type	Length in Bytes	Field Description
1	QMHJDS	DATE	10	Date job log created
2	QMHJTS	TIME	8	Time job log created
3	QMHMKS	CHAR	4	Message reference key
7	QMHSYN	CHAR	8	System name
8	QMHJBN	CHAR	26	Qualified job name
4	QMHLNN	BIN	4	Message line number
5	QMHSID	BIN	4	CCSID of text line
6	QMHTTY	CHAR	1	Message text indicator
9	QMHLIN	CHAR	78	Message text line

The length of the field indicates the number of total bytes for the field.

The definition of the fields in this record are as follows:

QMHJDS
Date job log created; DATE(8)

The date the production of the job log began. The field is a date field in the database record. The format of the date is *ISO. A value in this field is in the format yyyy-mm-dd. Each record produced for the same job log will have the same value in this field.

QMHJTS
Time job log created; TIME(8);

The time the production of the job log began. This field is defined as a time field in the database record. The format of the time is defined to be *ISO. A value in this field is in the format hh.mm.ss. Each record produced for the same job log will have the same value in this field.

QMHMKS
Message reference key; CHAR(4)

The message reference key the related message had in the job message queue. The records are placed in the secondary database file in ascending sequence by message reference key. There can be more than one secondary record for a specific message reference key. This field also exists in the related primary record. Therefore, once the message reference key is obtained from a primary record, it can be used to read the related records from the secondary file.

QMHSYN
System name; CHAR(8)

The name of the system that the job log was produced on.

QMHJBN
Qualified job Name; CHAR(26)

The fully qualified name of the job for which the message is being logged for. The first 10 positions contain the job name, the next 10 positions the user name, and the last six positions the job number.

QMHLNN
Message line number; BIN(4)

The line number of the line within the text type. For both the first and second level text, the line number starts at one for the first line of the text and is incremented by one for each additional line within that level.

QMHSID
CCSID of message text line; BIN(4)

The CCSID of the message text line that is contained in field QMHLIN.

QMHTTY
Message text type; CHAR(1)

An indicator which specifies whether field QMHLIN contains a line of the first or second level text. This field will contain one of the following values:

1 Field QMHLIN contains first level text.

2 Field QMHLIN contains second level text.

QMHLIN
Message text line: CHAR(78)

This field contains one line of the first or second level text.

Appendix C. IBM-Supplied Libraries in Licensed Programs (LP)

The iSeries server contains the definitions of many libraries. They provide a method to organize most of the objects that are stored on the system.

In the following tables, the IBM-supplied libraries display in alphabetical order under the licensed program (LP) that provided them.

- "IBM-Supplied Libraries for the OS/400 Licensed Program" shows the libraries that are supplied for use with the base operating system, the Operating System/400 (OS/400) licensed program.
- "IBM-Supplied Libraries for Other iSeries Licensed Programs" on page 415 shows the libraries supplied for all the other iSeries licensed programs.
  - The licensed programs appear alphabetically by the full descriptive name of each program. They display on the Install Licensed Programs menu on the system.
  - Also, if there are features in a licensed program, the feature name and number precede the libraries that are contained in that feature.

IBM-Supplied Libraries for the OS/400 Licensed Program

Table 12. IBM-Supplied Libraries for the OS/400 Program

Program Name	Library Name	Purpose of Library
Operating System/400 (5722-SS1)		
	QCCA	CCA cryptographic service provider
	QCLUSTER	HA switchable resources
	QDB2MS	DB2 multisystem
	QDNS	Domain name system
	QDOC	Contains document library objects residing in the system auxiliary storage pool (ASP), ASP 1. Document library services (DLS) system objects reside here. Not shipped with system; created at IPL time.
	QDOC*nnnn*	Contains document library objects residing in user auxiliary storage pools (ASP). *nnnn* is the number, 0002 through 0016, of the ASP. Not shipped with system; created at IPL time.
	QDSNX	Not shipped with system; created at install time
	QFNTCPL	AFP* compatibility fonts
	QFNTWT	Additional fonts
	QFPNTWE	NetWare enhanced integration
	QGDDM	Support for graphical data display manager (GDDM*) and presentation graphics routines (PGR)
	QGPL	General purpose library of the user
	QGPLTEMP	Not shipped with system; created at install time
	QHLPSYS	Online documentation for some system functions

© Copyright IBM Corp. 1997, 2004

Table 12. IBM-Supplied Libraries for the OS/400 Program (continued)

Program Name	Library Name	Purpose of Library
	QICSS	Digital Certificate Manager
	QICU	International components for Unicode
	QIWS	Host servers
	QJRNL	HA journal performance
	QMSE	Media and Storage Extensions (MSE) library
	QPASE	Portable app solutions environment
	QPFRDATA	Library of system-collected performance data
	QQALIB	Question-and-answer utility library
	QRCL	Library for reclaiming objects via the RCLSTG command
	QRECOVERY	System recovery library
	QRPLOBJ	Library for replaced objects
	QSCxxxxxxx	Data library for collecting APAR data; xxxxxxx = last 7 digits of the problem identifier
	QSHELL	Qshell
	QSMP	DB2 symmetric multiprocessing
	QSOC	OptiConnect
	QSPL	Spooling library
	QSR	Object Connect
	QSRV	System service library
	QSYS	System library
	QSYSCGI	Extended base directory support
	QSYSDIR	Extended base directory support
	QSYSINC	System openness includes
	QSYSLOCALE	Contains locale source members for use in creating *LOCALE objects.
	QSYSNLS	Extended NLS support
	QSYSVxRxMx	iSeries CL compiler library, previous release support; x identifies the version, release, and modification level of the previous release.
	QSYS2	Extended base support; supplemental system library for objects whose names do not begin with the letter Q; includes, for example, objects for CPIs.
	QSYSxxxx	Online information.
	QTEMP	Temporary library of user
	QUSRSYS	Additional IBM-supplied objects
	QUSRTEMP	Not shipped with system; created at install time
	QUSRTOOL	Example tools
Libraries for System/36 Environment		
	QSSP	System/36 environment library
	QS36F	System/36 environment file library; created by system
	#CGULIB	System/36 Character Generator Utility (CGU)
	#DFULIB	System/36 Data File Utility (DFU)
	#DSULIB	System/36 Development Support Utility (DSU)
	#LIBRARY	System/36 environment, general purpose library of user

Table 12. IBM-Supplied Libraries for the OS/400 Program (continued)

Program Name	Library Name	Purpose of Library
	#SDALIB	System/36 Screen Design Aid (SDA)
	#SEULIB	System/36 Source Entry Utility (SEU)
Library for System/38 Environment		
	QSYS38	System/38 environment library

IBM-Supplied Libraries for Other iSeries Licensed Programs

Table 13. Libraries for Other LPs on the iSeries server

Program Name	Library Name	Purpose of Library
IBM Advanced DBCS Printer Support for iSeries (5722-AP1)		
	QAPS	Advanced double-byte character set (DBCS) printer support
	QAPS2	Advanced DBCS Printer Support for AS/400-IPDS
IBM Advanced Function Printing DBCS Fonts for AS/400 (5769-FN1)		
	QFNT60	AFP DBCS Fonts/400–Base Support
	Library for: Japanese Fonts	
	QFNT61	AFP DBCS Fonts/400–Japanese
	Library for: Korean Fonts	
	QFNT62	AFP DBCS Fonts/400–Korean
	Library for: Traditional Chinese Fonts	
	QFNT63	AFP DBCS Fonts/400–Traditional Chinese
	Library for: Simplified Chinese Fonts	
	QFNT64	AFP DBCS Fonts/400–Simplified Chinese
	Library for: Thai Fonts	
	QFNT65	AFP DBCS Fonts/400–Thai
IBM Advanced Function Printing Fonts for AS/400 (5769-FNT)		
	QFNT00	OS/400 Base support for AFP™ fonts
	QFNT01	OS/400 Sonoran Serif** font support (Sonoran Serif is a functional equivalent of Monotype Times New Roman**)
	QFNT02	OS/400 Sonoran Serif Headliner font
	QFNT03	OS/400 Sonoran Sans Serif** font support (Sonoran Sans Serif is a functional equivalent of Monotype Arial**)
	QFNT04	OS/400 Sonoran Sans Serif Headliner font
	QFNT05	OS/400 Sonoran Sans Serif Condensed font
	QFNT06	OS/400 Sonoran San Serif Expanded font
	QFNT07	OS/400 Monotype Garamond** font
	QFNT08	OS/400 Century Schoolbook** font
	QFNT09	OS/400 Pi and special characters font
	QFNT10	OS/400 ITC Souvenir** font
	QFNT11	OS/400 ITC Avant Garde Gothic** font
	QFNT12	OS/400 Math and Science font
	QFNT13	OS/400 DATA1 font
	QFNT14	OS/400 APL2® font
	QFNT15	OS/400 OCR A and OCR B fonts

Table 13. Libraries for Other LPs on the iSeries server (continued)

Program Name	Library Name	Purpose of Library
Advanced Job Scheduler for iSeries (5722-JS1)		
	QIJS	Job Scheduler
IBM Advanced Function Printing Utilities for iSeries (5722-AF1)		
	QAFP	IBM Advanced Function Printing Utilities for iSeries
IBM Application Program Driver for AS/400 (5722-PD1)		
	QAPD	Application Program Driver/400
IBM Backup Recovery and Media Services for iSeries (5722-BR1)		
	QBRM	BRM Services/400 support
	QUSRBRM	BRM Services/400 user data
	Q1ABRMSFnn	Save file libraries per ASP(nn)
IBM Business Graphics Utility for AS/400 (5722-DS1)		
	QBGU	AS/400 Business Graphics Utility (BGU)
IBM CICS Transaction Server for iSeries (5722-DFH)		
	QCICS	CICS/400
	QCICSSAMP	Sample CICS® application programs
IBM Communications Utilities for iSeries (5722-CM1)		
	QRJE	Remote job entry (RJE) and VM/MVS bridge
Content Manager for iSeries (5722-VI1)		
	QVI	Content Manager for iSeries
Content Manager OnDemand for iSeries (5722-RD1)		
	QRDARS	OnDemand
	QUSRRDARS	OnDemand user data
	QUSROND	OnDemand user data
Cryptographic Access Provider 128-bit (5722-AC3)		
	QCAP3	Cryptographic Access Provider 128-bit
IBM Cryptographic Support for AS/400 (5722-CR1)		
	QCRP	IBM Cryptographic Support for AS/400
DataPropagator Relational V8 for iSeries (5722-DP4)		
	QDPR	DB2 DataPropagator
IBM DB2 Query Manager and SQL Development Kit for iSeries (5722-ST1)		
	QSQL	Structured Query Language/400 (SQL/400®)
DB2 Universal Database Extenders for iSeries V7.2 (5722-DE1)		
	QDBEX	DB2 UDB Extenders
	QDBXM (option 2)	XML Extender
	QDB2TX (option 1)	Text Extender
	QIMO (option3)	Text Search Engine
DCE Base Services for AS/400 (5769-DC1)		
	QDCE2	DCE Base Services for AS/400
DCE DES Library Routines for AS/400 (5769-DC3)		
	QDCEE	DCE DES Library Routines
	QDCE2	DCE Base Services for AS/400
Developer Kit for Java (5722-JV1)		
	QJAVA	Developer Kit for Java
HTTP Server for iSeries (5722-DG1)		
	QHTTPSVR	HTTP Server
	QTCM	Triggered Cache Manager
ILE C (5722-WDS)		
	QCLE	ILE C
ILE C++ (5722-WDS)		

Table 13. Libraries for Other LPs on the iSeries server (continued)

Program Name	Library Name	Purpose of Library
	QCPPLE	ILE C++
	QCXXN	ILE C++, previous release support
ILE COBOL (5722-WDS)		
	QCBLLE	ILE COBOL library
	QCBLLEP	ILE COBOL, previous release support
	QLBL	OPM COBOL
	#COBLIB	System/36-compatible COBOL
	QCBL	System/38-compatible COBOL
ILE RPG (5722-WDS)		
	QRPG	RPG/400
	QRPGLE	ILE RPG/400
	QRPGLEP	ILE RPG/400, previous release support
	#RPGLIB	System/36-compatible RPG II
	QRPG38	System/38-compatible RPG III
Infoprint Server for iSeries (5722-IP1)		
	QIPS	Infoprint Server
IBM e(logo)server iSeries Access Family (5722-XW1)		
	QCA400W	iSeries Access Family Base
iSeries Access for Windows® (5722-XE1)		
	QCAEXP	iSeries Access for Windows
iSeries Access for Web (5722-XH1)		
	QIWA	iSeries Access for Web
iSeries Access for Wireless (5722-XP1)		
	QIWR	iSeries Access for Wireless
iSeries Client Encryption (128-bit) (5722-CE3)		
	QCE3	Client Encryption 128-bit
iSeries Integration for Windows Server (5722-WSV)		
	QNTAP	Integration for Windows Server
IBM Managed System Services for iSeries (5722-MG1)		
	QSVMSS	SystemView Managed System Services/400
	QSVDSTRPS	SystemView distribution repository for user objects; created when LP is installed
IBM Performance Tools for iSeries (5722-PT1)		
	QPFR	IBM Performance Tools for iSeries
IBM Query for iSeries (5722-QU1)		
	QQRYLIB	IBM Query for iSeries
IBM System/38 Utilities for AS/400 (5722-DB1)		
	QIDU	System/38 Query, Text Management, and Data File Utility (DFU)
IBM System Manager for iSeries (5722-SM1)		
	QSMU	SystemView System Manager/400
IBM TCP/IP Connectivity Utilities for iSeries (5722-TC1)		
	QTCP	TCP/IP Connectivity Utilities/400
IBM Toolbox for Java (5722-JC1)		
	QJT400	IBM Toolbox for Java
VisualAge Generator Server for AS/400 (5769-VG1)		
	OVGEN	VisualAge Generator Server
	QGPL	General purpose library of the user
IBM WebSphere Studio Development Suite for iSeries (5722-WDS)		

Table 13. Libraries for Other LPs on the iSeries server (continued)

Program Name	Library Name	Purpose of Library
	QPDA	Includes the following tools and utilities: • APF (advanced printer function) • CGU (character generator utility), for DBCS systems only • DFU (data file utility) • ISDB (interactive source debugger) • PDM (programming development manager) • RLU (report layout utility) • SDA (screen design aid) • SEU (source entry utility)

Appendix D. Abbreviations of CL Commands and Keywords

This section contains alphabetic lists of abbreviations that are used in CL commands that are part of IBM OS/400 and other IBM iSeries licensed programs.

This information can assist you in naming commands and keywords in a consistent manner when using command definition. (See Chapter 9, "Defining Commands," on page 289 and the article on command definition statements in the *CL* section of the **Programming** category in the iSeries Information Center.

CL Command Verb Abbreviations

Most CL command names follow a consistent naming style. The first three letters of the command name represent the action that is being performed. The remaining letters of the command name describe the object that is having the action performed on it.

Another name for the three-letter command prefix is the command 'verb'. The majority of all CL commands use one of the following common command verbs:

Verb Abbreviation	Meaning
ADD	add
CHG	change
CRT	create
DLT	delete
DSP	display
END	end
RMV	remove
STR	start
WRK	work with

The following is a list of all the abbreviations that are used as command verbs:

Verb Abbreviation	Meaning
ADD	add
ALC	allocate
ALM	alarm
ANS	answer
ANZ	analyze
APY	apply
ASK	ask
BLD	build
CFG	configuration
CHG	change
CHK	check
CLO	close
CLR	clear
CMP	compare
CNL	cancel
CPH	cipher
CPR	compress

© Copyright IBM Corp. 1997, 2004

Verb Abbreviation	Meaning
CPY	copy
CRT	create
CVT	convert
DCP	decompress
DLC	deallocate
DLT	delete
DLY	delay
DMP	dump
DSC	disconnect
DSP	display
DUP	duplicate
EDT	edit
EJT	eject
EML	emulate
ENC	encipher
END	end
EXP	export
EXT	extract
FIL	file
FMT	format
FND	find
GEN	generate
GRT	grant
HLD	hold
IMP	import
INS	install
INZ	initialize
LNK	link
LOD	load
MGR	migrate
MON	monitor
MOV	move
MRG	merge
OPN	open
ORD	order
OVR	override
PAG	paginate
PKG	package
POS	position
PRM	promote
PRT	print
PWR	power
QRY	query
RCL	reclaim
RCV	receive
RGZ	reorganize
RLS	release
RMV	remove
RNM	rename
RPL	replace
RQS	request
RRT	reroute
RSM	resume
RST	restore

Verb Abbreviation	Meaning
RTV	retrieve
RUN	run
RVK	revoke
SAV	save
SBM	submit
SET	set
SLT	select
SND	send
STR	start
TFR	transfer
TRC	trace
UPD	update
VFY	verify
VRY	vary
WRK	work with

CL Command Abbreviations

The following is a list of all abbreviations that are used in CL command names, including command verb abbreviations:

Command Abbreviation	Meaning
A (suffix)	attributes
ABN	abnormal
ACC	access code
ACCGRP	access group
ACG	accounting
ACN	action
ACNE	action entry
ACT	active, activity, activation
ADD	add
ADM	application development manager, administration, administrative
ADP	adopt, adopting
ADPI	adapter information
ADPP	adapter profile
ADPT	adapter
ADR	address
ADSM	ADSTAR distributed storage manager
AFP	advanced function printing
AGR	agreement
AGT	agent
AJE	autostart job entry
ALC	allocate
ALR	alert
ALRD	alert description
ALRTBL	alert table
ALS	alias
ANS	answer
ANZ	analyze
AP	access path
APAR	authorized program analysis report
APF	advanced printer function

Abbreviations of CL Commands and Keywords

Command Abbreviation	Meaning
APP	application
APPC	advanced program-to-program communications
APPN	advanced peer-to-peer networking
APY	apply
ARA	area
ARC	archive
ASC	asynchronous
ASK	ask
ASN	association
ASP	auxiliary storage pool
AST	assistance
ATM	asynchronous transfer mode
ATN	attention
ATR	attribute
AUD	audit, auditing
AUT	authority
AUTE	authentication entry
AUTL	authorization list
BACK	back
BAL	balance, balancing
BAS	BASIC language
BCD	barcode
BCH	batch
BCK	backup
BCKUP	backup
BGU	business graphics utility
BKP	breakpoint
BKU	backup
BND	binding, bound
BP	boot protocol
BRM	BRMS (backup recovery and media services)
BSC	binary synchronous
BSCF	bsc file
BUF	buffer
C	C language
CAL	calendar
CALL	call
CAP	capture
CBL	COBOL language
CCS	change control server
CCT	IPX circuit
CCTRTE	circuit route
CCTSRV	circuit service
CDE	code, coded
CDS	coded data store
CFG	configuration
CFGL	configuration list
CFGLE	configuration list entry
CGY	category
CHG	change
CHK	check
CHT	chart
CICS	customer information control system

Command Abbreviation	Meaning
CL	control language
CLD	C locale description
CLG	catalog
CLNUP	cleanup
CLO	close
CLR	clear
CLS	class
CLT	client
CLU	cluster
CMD	command
CMN	communications
CMNE	communications entry
CMNF	communications file
CMP	compare
CMT	commit
CNL	cancel
CNN	connection
CNNL	connection list
CNNLE	connection list entry
CNR	container
CNT	contact
CNV	conversation
CODE	cooperative development environment
COL	collection
COM	community
COSD	class-of-service description
CP	check pending
CPH	cipher
CPIC	common programming interface for communications
CPP	C++ language
CPR	compress
CPT	component
CPY	copy
CPYSCN	copy screen
CRG	cluster resource group
CRL	crawler
CRP	cryptographic
CRQ	change request
CRSDMN	cross-domain
CRT	create
CSI	communications side information
CSL	console
CST	constraint, customization
CTG	cartridge
CTL	control
CTLD	controller description
CUR	current
CVG	coverage
CVN	conversion
CVT	convert
D (suffix)	description
DAT	date
DB	database

Command Abbreviation	Meaning
DBF	database file
DBG	debug
DCL	declare
DCP	decompress
DCT	dictionary
DDI	distributed data interface
DDM	distributed data management
DDMF	distributed data management file
DEP	dependent
DEV	device
DEVD	device description
DFN	definition
DFT	default
DFU	data file utility
DHCP	dynamic host configuration protocol
DIR	directory
DIRE	directory entry
DIRSHD	directory shadow
DKT	diskette
DKTF	diskette file
DL	DataLink
DLC	deallocate
DLF	DataLink file
DLFM	DataLink file manager
DLO	document library object
DLT	delete
DLY	delay
DMN	domain
DMP	dump
DNS	domain name service
DO	do
DOC	document
DOM	Domino
DPCQ	DSNX/PC queue
DPR	DataPropagator Relational
DSC	disconnect
DSK	disk
DSP	display
DSPF	display file
DST	distribution
DSTL	distribution list
DSTLE	distribution list entry
DSTQ	distribution queue
DSTSRV	distribution services
DTA	data
DTAARA	data area
DTAQ	data queue
DUP	duplicate
DWN	down
E (suffix)	entry
EDT	edit
EDTD	edit description
EDU	education
EJT	eject

Command Abbreviation	Meaning
EML	emulate, emulation
ENC	encipher
END	end
ENR	enrollment
ENV	environment
ENVVAR	environment variable
EPM	extended program model
ERR	error
ETH	ethernet
EWC	extended wireless controller
EWL	extended wireless line
EXIT	exit
EXP	expiration, export
EXT	extract
F (suffix)	file
FAX	facsimile
FCN	function
FCT	forms control table
FCTE	forms control table entry
FD	file description
FFD	file field description
FIL	file
FILL	fill
FLR	folder
FMT	format
FNC	finance
FND	find
FNT	font
FNTRSC	font resource
FNTTBL	font table
FORM	form
FORMD	form description
FORMDF	form definition
FR	frame relay
FRM	from
FRW	firewall
FTN	FORTRAN language
FTP	file transfer protocol
FTR	filter
GDF	graphics data format
GEN	generate
GO	go to
GPH	graphics
GPHPKG	graph package
GRP	group
GRT	grant
GSS	graphic symbol set
HDB	host database
HDW	hardware
HDWRSC	hardware resources
HLD	hold, held
HLL	high level language
HLP	help
HLR	holder

Command Abbreviation	Meaning
HOST	host
HST	history, historical
HTE	host table entry
HTTP	hypertext transfer protocol
I (suffix)	information, item, ILE
ICF	intersystem communications function
ICFF	icf file
IDD	interactive data definition utility
IDLC	ISDN data link control
IDX	index
IDXE	index entry
IFC	interface
IMG	image
IMP	import
INF	information
INP	input
INS	install
INT	internal machine
INTR	intrasystem
INZ	initialize
IPI	internet protocol over IPX
IPL	initial program load
IPS	internet protocol over SNA
IPX	internetwork packet exchange
IPXD	IPX description
ISDB	interactive source debugger
ISDN	integrated services digital network
ITF	interactive terminal facility
ITG	integrity
ITM	item
IWS	intelligent work station
JOB	job
JOBD	job description
JOBE	job entry
JOBQ	job queue
JOBQE	job queue entry
JRN	journal
JRNRCV	journal receiver
JS	job scheduler
JVA	Java
JVM	Java virtual machine
KBD	keyboard
KEY	key
L (suffix)	list
LAN	local area network
LANG	language
LBL	label
LCK	lock
LCL	local
LCLA	local attributes
LF	logical file
LFM	logical file member
LIB	library
LIBL	library list

Command Abbreviation	Meaning
LIBM	library member
LIC	license, licensed
LIN	line
LIND	line description
LNK	link
LOC	location
LOCALE	locale
LOD	load
LOF	logical optical file
LOG	log
LOGE	log entries
LPD	line printer daemon
LPDA®	link problem determination aid
LPR	line printer requester
LWS	local workstation
M (suffix)	member
MAC	message authentication code
MAIL	mail
MAP	map
MAX	maximum
MBR	member
MDL	model
MED	media
MEDDFN	media definition
MEDI	media information
MFS	mounted file system
MGD	managed
MGR	migrate, migration, manager
MGTCOL	management collection
MLB	media library
MLM	media library media
MNT	minutes, maintenance
MNU	menu
MOD	mode, module
MODD	mode description
MON	monitor
MOV	move
MRG	merge
MSF	mail server framework
MSG	message
MSGD	message description
MSGF	message file
MSGQ	message queue
MST	master
M36	AS/400 Advanced 36® machine
M36CFG	AS/400 Advanced 36 machine configuration
NAM	name
NCK	nickname
NDSCTX	NetWare directory services context
NET	network
NETF	network file
NFS	network file system
NODGRP	node group
NODL	node list

Command Abbreviation	Meaning
NTB	netbios
NTF	NetFinity
NTS	Notes
NTW	NetWare
NWI	network interface
NWS	network server
NWSAPP	network server application
NWSD	network server description
OBJ	object
OCL	operation control language
OF	optical file
OFC	office
OFF	off
OMC	object management cycle
OPC	opticonnect
OPN	open
OPT	optical
ORD	order
OUT	out, outgoing, output
OUTQ	output queue
OUTQD	output queue description
OVL	overlay
OVLU	overlay utility
OVR	override
OWN	owner
PAG	page, paginate
PAGDFN	page definition
PAGS	page segment
PAGSEG	page segment
PARM	parameter
PART	part
PASTHR	pass through
PC	personal computer
PCD	pc document
PCL	protocol
PCO	pc organizer
PCY	policy
PDF	portable document format
PDG	print descriptor group
PDM	programming development manager
PEX	performance explorer
PF	physical file
PFD	printout format definition
PFM	physical file member
PFR	performance
PFRG	performance graphics
PFRT	performance tools
PFU	print format utility
PFX	prefix
PGM	program
PGP	primary group
PGR	pager
PHS	phase
PIN	personal identification number

Command Abbreviation	Meaning
PJ	prestart job
PJE	prestart job entry
PKG	package
PLI	PL/I (programming language one)
PMN	permission
PMT	prompt
PNLGRP	panel group
POF	physical optical file
POL	pool
POP	post office protocol
PORT	port
POS	position
PPP	point-to-point protocol
PRB	problem
PRC	procedure
PRD	product
PRF	profile
PRFL	profile list
PRJ	project
PRM	promote
PRP	preparation
PRS	personal
PRT	print
PRTF	printer file
PRTQ	print queue
PSFCFG	print services facility configuration
PTC	portable transaction computer
PTF	program temporary fix
PTP	point-to-point
PTR	pointer
PVD	provider
PWD	password
PWR	power
PYM	payment
QM	query management
QRY	query
QRYF	query file
QSH	Qshell interpreter
QST	question
RBD	rebuild
RCD	record
RCL	reclaim
RCV	receive
RCY	recovery
RDAR	report/data archive and retrieval
RDB	relational database
RDR	reader
REF	reference
REG	registration
REX	REXX (restructured extended executor language)
RGP	ranking group
RGPE	ranking group entry
RGZ	reorganize

Command Abbreviation	Meaning
RJE	rje
RLS	release
RLU	report layout utility
RMC	report management cycle
RMT	remote
RMV	remove
RNM	rename
ROLL	roll
RPC	remote procedure call
RPDS	VM/MVS bridge (formerly Remote Spooling Communications Subsystem (RSCS)/PROFS distribution services)
RPG	report program generator
RPL	replace
RPT	report
RPY	reply
RPYL	reply list
RQS	request
RRT	reroute
RSC	resource
RSI	remote system information
RSM	resume
RST	restore
RTD	RouteD (TCP/IP)
RTE	route entry
RTGE	routing entry
RTL	retail
RTLF	retail file
RTN	return
RTV	retrieve
RUN	run
RVK	revoke
RWS	remote workstation
RXC	REXEC (remote execution)
SAV	save
SAVF	save file
SAVRST	save and restore
SBM	submit, submitted
SBS	subsystem
SBSD	subsystem description
SCD	schedule
SCDE	schedule entry
SCHIDX	search index
SCHIDXE	search index entry
SCN	screen
SDA	screen design aid
SDLC	synchronous data link control
SEC	security
SET	set
SEU	source entry utility
SFW	software
SHD	shadow, shadowing
SHRPOOL	shared pool
SIGN	sign

Command Abbreviation	Meaning
SIT	situation
SLT	select, selection
SLTE	selection entry
SMG	system manager
SMW	system manager workstation
SMTP	simple mail transfer protocol
SNA	systems network architecture
SND	send
SNI	SNA over IPX
SNMP	simple network management protocol
SNPT	SNA pass through
SNUF	SNA upline facility
SOC	sphere of control
SPADCT	spelling aid dictionary
SPL	spooling
SPLF	spooled file
SPT	support
SPTN	support network
SQL	structured query language
SRC	source
SRCF	source file
SRV	service
SRVPGM	service program
SSN	session
SSND	session description
SST	system service tools
STC	statistics
STG	storage
STGL	storage link
STM	stream, statement
STR	start
STS	status
SVR	server
SWA	save while active
SWL	stop word list
SYS	system
SYSCTL	system control
SYSDIR	system directory
SYSVAL	system value
S34	System/34
S36	System/36
S38	System/38
TAP	tape
TAPF	tape file
TBL	table
TBLE	table entry
TCP	TCP/IP (transmission control protocol/internet protocol)
TDLC	twinaxial data link control
TELN	telnet
TFR	transfer
TFTP	trivial file transfer protocol
THD	thread
THLD	threshold

Command Abbreviation	Meaning
TIE	technical information exchange
TIEF	tie file
TIMZON	time zone
TNS	transaction
TO	to
TOS	type of service
TPL	template
TRC	trace
TRG	trigger
TRN	token-ring network
TRP	trap
TXT	text
TYPE	type
T1	transport class 1
UBC	Ultimedia business conference
UDFS	user-defined file system
UPD	update
UPG	upgrade
USF	Ultimedia system facilities
USG	usage
USR	user
USRIDX	user index
USRPRF	user profile
USRPRTI	user print information
USRQ	user queue
USRSPC	user space
VAL	value
VAR	variable
VFY	verify
VLDL	validation list
VOL	volume
VRY	vary
VT	VT100 or VT220
VWS	virtual work station
WAIT	wait
WLS	wireless
WNT	Windows NT
WP	word processing
WRK	work with
WSE	work station entry
WSG	workstation gateway
WSO	workstation object
WTR	writer
X25	X.25

CL Command Keyword Abbreviations

Each command parameter has a keyword name associated with it. The keyword name can be up to ten characters. Keyword names do not follow the same style as command names in that they do not necessarily begin with a command verb. Where possible, use a single word or abbreviation. For example, common keyword names in CL commands include OBJ (Object), LIB (Library), TYPE, OPTION, TEXT, and AUT (Authority).

Construct a keyword name whenever using more than a single word or abbreviation to describe the parameter. Construct the keyword name by using a combination of standard command abbreviations and short unabbreviated words. For example, OBJTYPE is a common keyword name which combines the abbreviation 'OBJ' with the short word 'TYPE'.

The two primary goals for keyword names are to be recognizable and to be consistent between commands that provide the same function. Use of simple words and standard abbreviations helps to make the keyword names recognizable.

The following is a list of abbreviations that are used in CL command parameter keyword names:

Keyword Abbreviation	Meaning
A (suffix)	attributes, activity, address
ABS	abstract, absolute
ABN	abnormal
AC	autocall
ACC	access, access code
ACCMTH	access method
ACG	accounting
ACK	acknowledge, acknowledgement
ACN	action
ACP	accept
ACQ	acquire
ACSE	association control service element
ACT	active, activate, activation, activity, action
ACTSNBU	activate switched network backup
ADDR (or ADR)	address
ADJ	adjacent, adjust
ADL	additional
ADM	administration, Application Development Manager
ADMD	administration domain
ADP	adopt, adaptive
ADPT	adapter
ADR (or ADDR)	address
ADV	advance
AFN	affinity
AIX	AIX operating system
AJE	autostart job entry
AFP	advanced function printing
ALC	allocate
ALM	alarm
ALR	alert
ALRD	alert description
ALS	alias
ALW	allow
ANET	adjacent network entity title
ANS	answer
ANZ	analyze
AP	access path
APAR	authorized program analysis report
APF	advanced printer function
APPP	application process
APW	advanced print writer

Keyword Abbreviation	Meaning
APP	application
APPC	advanced program-to-program communications
APPN	advanced peer-to-peer networking
APY	apply
ARA	area
ARP	address resolution protocol
ASC	asynchronous communications
ASCII	American National Standard Code for Information Interchange
ASMT	assignment
ASN	assigned, association
ASP	auxiliary storage pool
AST	assistance
ASYNC	asynchronous
ATD	attended
ATN (or ATTN)	attention
ATR (or ATTR)	attribute
ATTACH	attached
ATTN (or ATN)	attention key
ATTR (or ATR)	attribute
AUD	audit, auditing
AUT	authority, authorized, authorization
AUTL	authorization list
AUTO	automatic
AUX	auxiliary
AVG	average
AVL	available
BAL	balance
BAS	BASIC language, base
BCD	barcode, broadcast data
BCH	batch
BCKLT	backlight
BCKUP (or BKU)	backup
BDY	boundary
BEX	branch extender
BGU	business graphics utility
BIN	binary
BIO	block input/output
BITS	data bits
BKP	breakpoint
BKT	bracket, backout
BKU (or BCKUP)	backup
BLDG	building
BLK	block
BLN	blinking cursor
BND	binding, bind, bound
BNR	banner
BOT	bottom
BRK	break
BSC	binary synchronous communications
BSCEL	binary synchronous communications equivalence link
BSP	backspace

Keyword Abbreviation	Meaning
BUF	buffer
C	C language
CAB	cabinet
CAL	calendar
CAP	capacity, capture
CB	callback
CBL	COBOL language
CCD	call control data
CCSID	coded character set identifier
CCT	circuit
CDE	code
CDR	call detail records
CFG	configuration
CFGL	configuration list
CFGLE	configuration list entry
CFM	confirmation, confirm
CGU	character generator utility
CHAR (or CHR)	character
CHG	change
CHK	check
CHKSUM	checksum
CHKVOL	check volume identifier
CHL	channel
CHR (or CHAR)	character
CHRSTR	character string
CHT	chart
CGY	category
CKR	checker
CKS	checksum
CL	control language
CLG	catalog
CLN	clean, cleaning, cleanup
CLNS	connectionless-mode network service
CLNUP (or CLN)	cleanup
CLO	close
CLR	clear
CLS	class
CLSF	class file
CLT	client
CLU	cluster
CMD	command
CMN	communications
CMNE	communications entry
CMP	compare
CMT	commitment, comment
CNG	congestion
CNL	cancel
CNLMT (or CNNLMT)	connection limit
CNN	connection
CNNL	connection list
CNNLMT	connection limit
CNR	container
CNS	constant
CNT	contact

Keyword Abbreviation	Meaning
CNTL (or CTL)	control
CNTRY	country
CNV	conversation
COD	code
CODPAG	code page
CODE	code, cooperative development environment
COL	column, collect, collection
COM	common, community
COMPACT	compact, compaction
CON	confidential
CONCAT	concatenation
COND	condition
CONS	connection-mode network service
CONTIG	contiguous
CONT	continue
COS	class-of-service
COSD	class-of-service description
COSTBYTE	cost per byte
COSTCNN	cost per connection
COVER	cover letter
CP	control point
CPB	change profile branch, compatibility
CPI	characters per inch
CPL	complete
CPP	C++ language
CPR	compressed, compress
CPS	call progress signal
CPT	component
CPU	central processing unit
CPY	copy
CPYRGT	copyright
CRC	cyclic redundancy check
CRDN	credentials
CRG	charges, charging, cluster resource group
CRL	correlation
CRQ	change request
CRQD	change request description
CRSDMNK	cross-domain key
CRT	create
CSI	communications side information
CSL	console
CSR	cursor
CST	constraint, cost
CTG	cartridge
CTL	controller, control
CTLD	controller description
CTN	contention
CTS	clear to send
CTX	context
CUR	current
CVN	conversion
CVR	cover
CVT	convert, converting
CXT (or CTX)	context

Keyword Abbreviation	Meaning
CYC	cycle
D (suffix)	description
DAP	directory access protocol
DAT	date
DB	database
DBCS	double-byte character set
DBF	database file
DBG	debug
DBR	database relations
DCE	data communications equipment, distributed computing environment
DCL	declare
DCP	decompress
DCT	dictionary
DDI	distributed data interface
DDM	distributed data management
DDMF	distributed data management file
DDS	data description specification
DEC	decimal
DEGREE	parallel processing degree
DEL	delivery
DEP	dependent
DEPT	department
DES	data encryption standard
DEST	destination
DEV	device
DEVD	device description
DFN	definition, defined
DFR	defer
DFT	default
DFU	data file utility
DIAG	dialogue
DIF	differences
DIFF	differentiated
DIR	directory
DKT	diskette
DLC	deallocate
DLO	document library object
DLT	delete
DLVRY	delivery
DLY	delay
DMN	domain
DMP	dump
DN	distinguished name
DOC	document
DPR	DataPropagator Relational
DRAWER	drawers
DRT	direct
DRV	drive
DSA	directory systems agent
DSAP	destination service access point
DSB	disable, disabled
DSC	disconnect
DSK	disk

Keyword Abbreviation	Meaning
DSP	display
DST	daylight savings time, dedicated service tools, distribution
DTA	data
DTE	data terminal equipment
DTL	detail
DUP	duplicate
DVL	development
DWN	down
E (suffix)	entry
EBCDIC	extended binary-coded decimal interchange code
ECN	explicit congestion notification
EDT	edit
EDU	education
EIM	enterprise identity mapping
EJT	eject
ELAN	ethernet LAN
ELEM	element
ELY	early
EML	emulate, emulation
ENB	enable
ENC	encode
ENR	enrollment
ENT	enter
ENV	environment
EOF	end of file
EOR	end of record
EOV	end of volume
ERR	error
EST	establish, established
ETH	ethernet
EVT	event
EXC	exclude
EXCH	exchange
EXD	extend, extended
EXEC	executive
EXIST	existence
EXN	extension
EXP	expiration, expire
EXPR	expression
EXT	extract, extend, extended
F (suffix)	file
FAIL	failure
Fnn	function key 'nn'
FA	file attributes
FAX	facsimile
FCL	facilities
FCN	function, functional
FCT	forms control table
FCTE	forms control table entry
FEA	front end application
FEA	front end application
FEAT	feature

Keyword Abbreviation	Meaning
FID	file identifier
FIL	file
FLD	field
FLG	flag
FLIB	files library
FLR	folder
FLW	flow
FLV	failover
FMA	font management aid
FMT	format
FNC	finance
FNT	font
FORMDF	form definition
FP	focal point
FRAC	fraction
FRC	force
FRI	Friday
FRM	from, frame
FRQ	frequency
FSC	fiscal
FSN	file sequence number
FST	first
FTAM	file transfer, access, and management
FTP	file transfer protocol (TCP/IP)
FTR	filter
GC	garbage collection
GCH	garbage collection heap
GDF	graphics data file
GDL	guideline, guidelines
GEN	generate, generation
GID	group identifier number
GIV	give
GLB	global
GNL	general
GPH	graph
GRP	group
GRT	grant
GSS	graphics symbol set
GVUP	give up
HCP	host command processor
HDL (or HNDL)	handle
HDR	header
HDW	hardware
HEX	hexadecimal
HFS	hierarchical file systems
HLD	hold, held
HLL	high-level language
HLP	help
HLR	holder
HNDL (or HDL)	handle
HPCP	host to printer code page
HPFCS	host to printer font character set
HPR	high performance routing
HRZ	horizontal

Keyword Abbreviation	Meaning
HST	history, historical
HTML	hypertext markup language
HTTP	hypertext transfer protocol (TCP/IP)
I (suffix)	information, ILE
ICF	intersystem communication function
ICV	initial chaining value
ID	identifier
IDD	interactive data definition
IDL	idle
IDLC	integrated data link control
IDP	interchange document profile
IDX	index
IE	information element
IFC	interface
IGC	ideographic (double-byte character set)
IGN	ignore
IFC	interface
ILE	integrated language environment
IMG	image
IN	input
INAC (or INACT)	inactivity
INACT (or INAC)	inactivity
INC	include
IND	indirect
INF	information
INFSKR	Infoseeker
INH	inhibit
INIT	initiate
INL	initial
INM	intermediate
INP	input
INPACING	inbound pacing
INQ	inquiry
INS	install
INST	instance
INT	interactive, integer, internal
INTNET	internet
INTNETA	internet address
INTR	intrasystem
INV	invitee, inventory, invoke
INZ	initialize, initialization
IP	internet protocol
IPDS	intelligent printer data stream
IPI	IP over IPX
IPL	initial program load
IPX	internet packet exchange
ISDN	integrated services digital network
ISP	internet service provider
IT	intermediate text, internal text
ITF	interactive terminal facility
ITM	item
ITV	interval
IW2	IPX WAN version 2 protocol
J (suffix)	job

Keyword Abbreviation	Meaning
JDFT	join default
JE (suffix)	job entry
JFLD	join field
JORDER	join file order
JRN	journal
JRNRCV	journal receiver
JVA	Java
KBD	keyboard
KNW	know, knowledge
KPF	Kanji printer function
KWD	keyword
L (suffix)	list
LADN	library assigned document name
LAN	local area network
LANG (or LNG)	language
LBL	label
LCL	local
LCLE	local location entry
LCK	lock
LDTIME	lead time
LE (suffix)	list entry
LEC	LAN emulation client
LECS	LAN emulation configuration server
LEN	length
LES	LAN emulation server
LF	logical file
LFM	logical file member
LFT	left
LGL	logical
LIB	library
LIBL	library list
LIC	license, licensed, licensed internal code
LIFTM	lifetime
LIN	line, line description
LMI	local management interface
LMT	limit
LNG (or LANG)	language
LNK	link
LNR	listener
LOC	location
LOD	load
LPI	lines per inch
LRC	longitudinal redundancy check
LRG	large
LRSP	local response
LST	list, last
LTR	letter
LVL	level
LWS	local workstation
LZYWRT	lazy write
M (suffix)	member, messages
MAC	macro, medium access control
MAINT	maintenance
MAJ	major

Keyword Abbreviation	Meaning
MAP	map, manufacturing automation protocol
MAX	maximum
MBR	member
MBRS	members
MCA	message channel agent
MCH	machine
MDL	model
MDM	modem
MDTA	message data
MED	media, medium
MEDI	media information
METAFILE	metatable file
MFR	manufacturer
MFS	mounted file system
MGR	manager
MGRR	manager registration
MGT	management
MID	middle
MIN	minimal, minimize
MLB	media library device
MLT	multiplier, multiple
MM	multimedia
MNG	manage
MNT	maintenance, mount, mounted
MNU	menu
MOD	mode, module
MODD	mode description
MON	monitor, Monday
MOV	move
MQM	Message Queue Manager
MRG	merge
MRK	mark
MRT	multiple requester terminal
MSF	mail server framework
MSG	message
MSGS	messages
MSGQ	message queue
MSR	measurement
MSS	managed system services
MST	master
MTD	mounted
MTG	meeting
MTU	maximum transmission unit
MTH	method
MULT (or MLT)	multiple
M36	AS/400 Advanced 36 machine
M36CFG	AS/400 Advanced 36 machine configuration
N (suffix)	name, network
NAM	name
NBR	number
NCK	nickname
NDE	node
NDM	normal disconnect mode
NDS	NetWare directory services

Keyword Abbreviation	Meaning
NEG	negative, negotiation
NEP	never-ending program
NET	network
NFY	notify
NL	network-layer
NLSP	NetWare link services protocol
NML	namelist
NNAM (or NCK)	nickname
NOD	node
NODL	node list
NORM	normal
NOTVLD	not valid
NPRD	nonproductive
NRM	normal, normal response mode
NRZI	non-return-to-zero-inverted
NT	network termination
NTB	NetBIOS
NTBD	NetBIOS description
NTC	notice
NTF	NetFinity
NTP	network time protocol
NTS	Notes
NTW	NetWare
NTW3	NetWare 3.12
NUM	numeric, number
NWI	network interface
NWID	network interface description
NWS	network server
NWSD	network server description
NXT	next
OBJ	object
OBS	observable information
OFC	office
OFSET	offset
OMT	omit
OPN	open
OPR	operator, operating
OPT	option, optical, optimum
ORD	order
ORG	organization, organizational
ORGUNIT	organizational unit
OS	operating system
OSDB	object store database
OUT	output
OVF	overflow
OVL	overlay
OVR	override
OVRFLW	overflow
OWN	owner, owned
PAD	packet assembly/disassembly
PAG	page, paginate
PAL	product activity log
PARM	parameter
PASTHR	pass-through

Keyword Abbreviation	Meaning
PBL	probable
PBX	private branch exchange
PC	personal computer
PCD	PC document
PCL	protocol
PCO	PC organizer
PCS	personal computer support
PCT	percent
PCTA	personal computer text assist
PCY	policy
PDF	portable document format
PDG	print descriptor group
PDM	programming development manager
PDU	protocol data unit
PENWTH	pen width
PERS	personal
PF	physical file
PFnn	program function key 'nn'
PFD	printout format definition
PFM	physical file member
PFVLM	physical file variable-length member
PFR	performance
PFX	prefix
PGM	program
PGP	primary group
PGR	pager
PHCP	printer to host code page
PHFCS	printer to host font character set
PHS	phase
PHY	physical
PIN	personal identification number
PJE	prestart job entry
PKA	public key algorithm
PKG	package
PKT	packet
PL	presentation-layer
PLC	place
PLL	poll, polling
PLT	plotter
PMN	permission
PMP	point-to-multipoint
PMT	prompt
PND	pending
PNL	panel
PNT	point
POL	pool
POLL	polled, polling
POP	post office protocol (TCP/IP)
PORT	port number
POS	positive, position
PPP	point-to-point protocol
PP	preprocessor
PPR	paper
PPW	page printer writer

Keyword Abbreviation	Meaning
PRB	problem
PRC	procedure, procedural, process
PRD	product, productive
PRJ	project
PREBLT	prebuilt
PRED	predecessor
PREEST	pre-established
PREF	preference, preferred
PREOPR	pre-operation
PREREQ	prerequisite
PRF	profile, profiling
PRI	primary
PRJ	project
PRM	promote, parameters
PRMD	private management domain
PRN	parent
PRO	proposed
PROC (or PRC)	procedure, processing
PROD	production
PROP	property/properties
PRP	prepare, propagate, propagation
PRS	personal
PRT	print, printer
PRTQ	print queue
PRV	previous
PRX	proxy
PSAP	presentation-layer service access point
PSF	print services facility
PSN	presentation
PSTOPR	post-operation
PTC	protected, protection, portable transaction computer
PTF	program temporary fix
PTH	path
PTL	partial
PTN	partition, partitioning
PTP	point-to-point
PTR	pointer
PTY	priority
PUB	public
PUNS	punches
PVC	permanent virtual circuit
PVD	provider
PVT	private
PWD	password
PWR	power
Q (suffix)	queue
QE (suffix)	queue entry
QLTY	quality
QOS	quality of service
QRY	query
QST	question
QSTDB	question-and-answer database
QSTLOD	question-and-answer load

Keyword Abbreviation	Meaning
QUAL	qualifier
RAR	route addition resistance
RBD	rebuild
RCD	record
RCDS	records
RCL	reclaim
RCMS	remote change management server
RCP	recipient
RCR	recursion, recurs
RCV	receive
RCY	recovery
RDB	relational database
RDN	relative distinguished name
RDR	reader
RDRE	reader entry
REACT	reactivation
REASSM	reassembly
REC	record
RECNN	reconnect
REF	reference
REINZ	reinitialize
REL	relations, release
REP	representation, representative
REQ (or RQS)	required, request, requester
RES	resident, resolution
RESYNC	resynchronize
RET	retention
REX	REXX language
RFS	refuse, refused
RGS	registration
RGT	right
RGZ	reorganize
RINZ	reinitialize
RIP	routing information protocol
RJE	remote job entry
RJT	reject
RLS	release
RMD	remind, reminder
RMT	remote
RMV	remove
RNG	range
RNM	rename
RPG	RPG language
RPL	replace, replacement
RPT	report
RPY	reply
RQS (or REQ)	request, requester
RQT	requisite
RRSP	remote response
RRT	reroute
RSB	reassembly
RSC	resource, resources
RSL	result, resolution
RSM	resume

Keyword Abbreviation	Meaning
RSP	response
RSRC	resource
RST	restore
RSTD	restricted
RTE	route
RTG	routing
RTL	retail
RTM	retransmit
RTN	return, returned, retransmission
RTR	router
RTT	rotate
RTV	retrieve
RTY	retry
RU	request unit
RVK	revoke
RVS	reverse
RWS	remote workstation
SAA®	systems application architecture
SADL	saddle
SAP	service access point
SAT	Saturday
SAV	save
SAVF	save file
SBM	submit, submitted
SBS	subsystem
SCD	schedule, scheduled
SCH	search
SCN	screen
SCT	sector
SDLC	synchronous data link control
SDU	service data unit
SEC	second, secure, security
SEG	segment
SEGMENT	segmentation
SEL (or SLT)	select, selection
SENSITIV	sensitivity
SEP	separator
SEQ	sequence, sequential
SEV	severity
SFW	software
SGN	sign-on
SHD	shadow, shadowing
SHF	shift
SHM	short hold mode
SHR	shared
SHUTD	shutdown
SI	shift-in
SIG	signature, signed
SIGN	sign-on
SIZ	size
SL	session-layer
SLR	selector
SLT (or SEL)	select, selection
SMAE	systems management application entity

Keyword Abbreviation	Meaning
SMG	systems manager
SMTP	simple mail transfer protocol
SMY	summary
SNA	systems network architecture
SNBU	switched network back-up
SND	send
SNG	single
SNI	SNA over IPX
SNP	snap
SNPT	SNA pass-through
SNUF	SNA upline facility
SO	shift-out
SOC	sphere of control
SPA	spelling aid
SPC	space, special
SPD	supplied
SPF	specific
SPID	service provider identifier
SPL	spooled, spooling
SPR	superseded
SPT	support, supported
SPTN	support network
SPX	sequenced packet exchange
SQL	structured query language
SRC	source
SRCH (or SCH)	search
SRM	system resource management
SRQ	system request
SRT	sort
SRV	service
SSAP	source service access point, session-layer service access point
SSCP	system services control point
SSL	secure sockets layer
SSN	session
SSND	session description
SSP	suspend
SST	system service tools
STAT	statistical data records
STATION	convenience station
STC	statistics
STD	standard
STG	storage
STK	stack
STM	stream
STMF	stream file
STMT	statement
STN	station
STP	step
STPL	staple
STR	start, starting
STS	status
STT	state
STX	start-of-text character

Keyword Abbreviation	Meaning
SUB	substitution, subject
SUBADR	subaddress
SUBALC	suballocation
SUBDIR	subdirectory
SUBFLR	subfolder
SUBNET	subnetwork
SUBPGM	subprogram
SUBST	substitution
SUCC	successor
SUN	Sunday
SURNAM	surname
SVC	switched virtual circuit
SVR	server
SWL	stop word list
SWS	switches
SWT	switch, switched
SWTSET	switch setting
SYM	symbol, symbolic
SYN	syntax
SYNC	synchronous
SYS	system
SYSLIBL	system library list
S36	System/36
TAP	tape
TAPDEV	tape devices
TBL	table
TCID	transport connection identifier
TCP	TCP/IP (transmission control protocol/internet protocol)
TCS	telephony connection services
TDC	telephony data collector
TDLC	twinaxial data link control
TEID	terminal endpoint identifier
TEL	telephone
TELN	TELNET (TCP/IP)
TERM	terminal
TFR	transfer
TGT	target
THD	thread
THLD	threshold
THR	through, throughput
THRPUT	throughput
THS	thesaurus
THU	Thursday
TIE	technical information exchange
TIM	time
TIMMRK	timemark
TIMO	timeout
TIMOUT (or TIMO)	timeout
TIMZON	time zone
TKN	token
TL	transport-layer
TM	time
TMN	transmission

Keyword Abbreviation	Meaning
TMP	temporary
TMPL (or TPL)	template
TMR	timer
TMS	transmission
TMT	transmit
TNS	transaction
TOKN (or TKN)	token
TOT	total
TPDU	transport-layer protocol data unit
TPL	template, topology
TPT	transport
TRANS	transit, transaction
TRC	trace
TRG	trigger
TRM	term
TRN	token-ring network, translate
TRNSPY	transparency
TRP	trap
TRS	transit
TRUNC	truncate
TSE	timeslice end
TSP	timestamp
TSAP	transport-layer service access point
TSK	task
TST	test
TUE	Tuesday
TWR	tower
TXP	transport
TXT	text
TYP	type
T1	transport class 1
T2	transport class 2
T4	transport class 4
UDFS	user-defined file system
UDP	user datagram protocol
UI	user interface, unnumbered information
UID	user identifier number
UNC	unclassified
UNL	unlink
UNPRT	unprintable
UNQ	unique
UOM	unit of measure
UPCE	universal product code type E barcode
UPD	update
UPG	upgrade
URL	uniform resource locator
USG	usage
USR	user
VAL	value
VAR	variable
VCT	virtual circuit
VDSK	virtual disk
VER (or VSN)	version
VFY	verify

Keyword Abbreviation	Meaning
VLD	valid, validity, validation
VND	vendor
VOL	volume
VRF	verification
VRT	virtual
VRY	vary
VSN (or VER)	version
VWS	virtual work station
WAN	wide area network
WCH	watch
WDW	window
WED	Wednesday
WIN	winner
WK	week
WNT	Windows NT
WP	word processing
WRD	word
WRK	work, working
WRT	write
WS	workstation
WSC	workstation controller
WSCST	workstation customization object
WSE	workstation entry
WSG	workstation gateway (TCP/IP)
WSO	workstation object
WTR	writer
WTRE	writer entry
WTRS	writers
X25	X.25
X31	X.31
X400	X.400
3270	3270 display

Appendix E. 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 give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
500 Columbus Avenue
Thornwood, NY 10594-1785
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:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

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.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created

© Copyright IBM Corp. 1997, 2004

programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Software Interoperability Coordinator, Department 49XA
3605 Highway 52 N
Rochester, MN 55901
U.S.A.

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 information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, 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. 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 IBM's application programming interfaces.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. _enter the year or years_. All rights reserved.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Programming Interface Information

CL Programming documents intended Programming Interfaces that allow the customer to write CL programs.

Trademarks

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

Advanced 36
AFP
Application System/400
CICS
CICS/400
Client Access
COBOL/400
C/400
DataPropagator
DB2
DB2 Universal Database
e (logo)
IBM
Integrated Language Environment
IPDS
iSeries
Operating System/400
OS/400
RPG/400
SAA
System/36
System/38

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Other company, product, and service names may be trademarks or service marks of others.

Bibliography

For more information about OS/400 programming, see the following books and iSeries Information Center topics.

- Application Display Programming

  This guide provides information about using DDS to create and maintain displays, creating and working with display files, creating online help information, using UIM to define displays, and using panel groups, records, and documents.

- Backup and Recovery

  This manual provides general information about recovery and availability options for the iSeries server. It describes the options available on the server, compares and contrasts them, and tells where to find more information about them.
- The **Backup and Recovery** topic in the **Systems Management** category of information in the iSeries Information Center includes information about how to plan a backup and recovery strategy, how to back up your server, how to manage tape libraries, and how to set up disk protection for your data. It also includes information about the Backup, Recovery and Media Services plug-in to iSeries(TM) Navigator, information about recovering your server, and answers to some frequently asked questions about backup and recovery.
- The **CL** topic in the**Programming** category of information in the iSeries Information Center provides information for system programmers and system administrators who write programs using OS/400 commands and other IBM-supplied commands.

- ILE Concepts

  This book explains concepts and terminology pertaining to the Integrated Language Environment (ILE) architecture of the OS/400 licensed program. Topics covered include creating modules, binding, running programs, debugging programs, and handling exceptions.
- The **Database** category of information in the iSeries Information Center provides about DB2 Universal Database™ (UDB) for iSeries, which enables you to access and manage server data through an application or a user interface, as well as providing advanced functions such as referential integrity and parallel database processing.
- The **Files and File Systems** category of information in the iSeries Information Center gives you information about both the database file management capabilities and the integrated file system capabilities of the IBM iSeries server.
- The **Globalization** topic in the**Programming** category of information in the iSeries Information Center provides the data processing manager, system operator and manager, application programmer, end user, IBM marketing representative, and system engineer with information required to understand and use the globalization function on the iSeries server. This book prepares the user for planning, installing, configuring, and using globalization and multilingual support of the iSeries server. It also provides an explanation of the database management of multilingual data and application considerations for a multilingual system.

- Printer Device Programming

  This guide provides specific information on printing elements and concepts of the iSeries server, printer file and print spooling support, and printer connectivity.
- The **Printing** category of information in the iSeries Information Center provides information about how to plan for and configure printing functions, as well as basic printing information.

- Security - Reference

  This book discusses general security concepts and planning for security on the system. It also includes information for all users about resource security.
- The **Security** category of information in the iSeries Information Center provides information about how to set up and plan for your iSeries security, how to secure network and communications applications, and how to add highly secure cryptographic processing capability to your iSeries server. It also includes information about object signing and signature validation, identity mapping, and solutions to Internet security risks.

© Copyright IBM Corp. 1997, 2004

- The **Systems Management** category of information in the iSeries Information Center provides information about creating and changing the work management environment, working with system values, collecting and using performance data to improve system performance.
- The **APIs** topic in the**Programming** category of information in the iSeries Information Center provides information and examples for experienced application and system programmers who want to use the OS/400 application programming interfaces (APIs).

For more information about OS/400 programming utilities mentioned in this book, see the following books:

- ADTS/400: Character Generator Utility
- ADTS/400: Programming Development Manager
- ADTS for AS/400: Screen Design Aid
- ADTS for AS/400: Source Entry Utility

Index

Special characters

/* (comment) delimiter 29
* (asterisk)
  comments in programs 29
  OUTPUT (output) parameter 117
*ALL authority 112
*AND operator 41
*CHANGE authority 112
*EXCL (exclusive) lock state 140
*EXCLRD (exclusive allow read) lock state 140
*EXCLUDE authority 112
*INT2 value 344
*INT4 value 344
*LDA value
  local 93
*NOT operator 41
*OR operator 41
*SHRNUP (shared-no-update) lock state 140
*SHRRD (shared-for-read) lock state 140
*SHRUPD (shared-for-update) lock state 140
*UINT2 344
*UINT4 344
*USE authority 112
%BIN (binary) function 45
%BINARY (binary) built-in function
  description 45
%SST (substring) function
  description 47
  processing qualified name 318
%SUBSTRING (substring) built-in function
  description 47
  processing qualified name 318
%SWITCH (switch) function 49

A

abnormal job end 178
activation program 381
add authority 112
Add Breakpoint (ADDBKP) command
  description 383
  example 384
Add Library List Entry (ADDLIBLE) command 107
Add Message Description (ADDMSGD) command
  example 196
  file name 187
  FMT (format) parameter 190
  specifying information 183
Add Program (ADDPGM) command 380
Add Trace (ADDTRC) command 388
  example 388
ADDBKP (Add Breakpoint) command
  description 383
ADDBKP (Add Breakpoint) command *(continued)*
  example 384
adding
  adding to program 387
  breakpoint to program 383
  library list entry 107
  message description
    ADDMSGD (Add Message Description) command 196
    example 196
    file 187
    FMT (format) parameter 190
    value 183
  program objects to debug session 361
  trace to program 387
ADDLIBLE (Add Library List Entry) command 107
ADDMSGD (Add Message Description) command
  example 196
  file name 187
  FMT (format) parameter 190
  specifying information 183
ADDPGM (Add Program) command 380
ADDTRC (Add Trace) command 388
  example 388
alert identifier
  specifying 196
allocating
  resource 139
Allow alerts
  alerts
    using allow alerts 203
ALROPT code
  entry size 186
  specifying 196
API (application programming interface)
  days used count 130
application programming interface (API)
  days used count 130
asterisk (*)
  comments in programs 29
  OUTPUT (output) parameter 117
attribute
  basic 119
  command 332
  default for newly created object 115
  displaying module 65
  displaying program 65
  full 119
  message queue 202
  object description 119
  retrieving 181
  service 119
authority
  *ALL 112
  *CHANGE 112
  *EXCLUDE 112
authority *(continued)*
  *USE 112
  add 112
  combined 112
  data 112
  default for newly created object 113
  defined command 292
  delete 112
  execute 112
  library 108, 111
  object 111
  object existence 111
  object management 111
  object operational 111
  read 112
  update 112
automatic decompression 138
automatic variable
  program 392

B

batch entry 13
batch job
  breakpoint program 384
  debugging job not started from job queue 394
  submitted to a job queue, debugging 393
batch job log
  consideration 282
binary function 45
branching
  unconditional 29
break delivery of message 202
break message
  sending 183, 211
break-handling program 205, 246, 247
breakpoint
  characteristics 366
  conditional
    add 386
    description 365
    example 370
    removing 367
    setting 367
  function 366
  removing
    conditional 367
    description 371
    from program 365
    unconditional 366
  resuming program processing 384
  setting
    conditional 367
    description 365
    unconditional 366
  unconditional
    description 365
    removing 366
    setting 366

© Copyright IBM Corp. 1997, 2004

breakpoint program
batch job 384
built-in function 13

C

calculation 16
See also expression
CHGVAR (Change Variable) command 13
call
description 380
level
description 380
nesting 380
stack 380
CALL (Call Program) command
description 69
function 16
using 75
Call Procedure (CALLPRC) command
description 70
Call Program (CALL) command
description 69
function 16
using 75
call stack
description 380
displaying testing information 391
entry identification
on SNDPGMMSG 218
relationship to CALL command 70
relationship to CALLPRC command 71
removing call 70, 71
removing request in error 382
TFRCTL (Transfer Control) command 399
call stack entry message queue 207
calling procedure
CALLPRC command description 70
CALLPRC command example 82
calling program
CALL command description 69
using the CALL command 75
Callprc (CALL PROCEDURE) command 16
CALLPRC (Call Procedure) command 16
description 70
example 82
canceling
request while testing 382
CCSID 213
messages
using CCSID 203
CEELOCT program 55
century digit
parameter value to CPP (command processing program)
date 295
change authority 112
Change Command (CHGCMD) command 337
Change Current Library (CHGCURLIB) command 107
Change Data Area (CHGDTAARA) command 16, 96
Change Debug (CHGDBG) command 380
Change Job (CHGJOB) command 272
Change Library List (CHGLIBL) command 23, 107
Change Message Description (CHGMSGD) command 187, 200
Change Message Queue (CHGMSGQ) command 202, 205
Change Program Variable (CHGPGMVAR) command 392
Change System Library List (CHGSYSLIBL) command 107
Change Variable (CHGVAR) command
definition 16
example 26, 217
changing
CL program at run time 168
command 337
command definition effect on program 337
current library 107
data area 16, 96
debug 380
job 272
library list 23, 107
message description 187, 200
message queue 202, 205
module object 364, 365
program variable 392
system library list 107
variable
CL procedure 13, 16
example 26, 217
variable value
in program 392
character
lowercase
variable 24
character length error 81
Check Object (CHKOBJ) command 16, 147
checking
object 16, 147
program validity 290
CHGCMD (Change Command) command 337
CHGCURLIB (Change Current Library) command 107
CHGDBG (Change Debug) command 380
CHGDTAARA (Change Data Area) command 16, 96
CHGJOB (Change Job) command 272
CHGLIBL (Change Library List) command 23, 107
CHGMSGD (Change Message Description) command 187, 200
CHGMSGQ (Change Message Queue) command 202, 205
CHGPGMVAR (Change Program Variable) command 392
CHGSYSLIBL (Change System Library List) command 107
CHGVAR (Change Variable) command
%SWITCH setting 50
CL procedure 13
definition 16
example 26, 217
CHKOBJ (Check Object) command 16, 147
choice for parameter 320
CL command
See command, CL
CL procedure
advantages for using 11
batch entry 13
command
logging 60
used 16
compiler listing 61
controlling processing 20, 29
creating
CRTCLMOD command 16
using CRTCLMOD command 12, 59
using source statements 12
description 3
example 15
interactive entry 12
introduction 11
obtaining dump 63
overriding database file 160
overriding display file 156
parts 13
procedure creation 12
purpose 11
referring to file 151
source creation 12
using 18
variable, command to work with 16
working with 59, 145
working with file 148
writing comment 29
CL program
creating 12
display formatting 148
files supported 148
receiving data 153
sending data 153
substring built-in function (%SUBSTRING)
used to process qualified name 318
CL variable
declaring 13, 16
Clear Library (CLRLIB) command 115
Clear Trace Data (CLRTRCDTA) command 387, 388
clearing
library 115
trace data 387
CLRLIB (Clear Library) command 115
CLRTRCDTA (Clear Trace Data) command 387
CMD (command) parameter 16
CMD (Command) statement
defining 292
example 346
combined authority 112

command
See also command definition
debug 358
description 1
equating a name 377
STEP debug 371
command (CMD) parameter 16
Command (CMD) statement
defining 292
example 346
command analyzer exit points 175
command default
changing 339
command definition
See command processing program (CPP)
See also object
See parameter
data type parameter restriction 299
defining
simple list 305
displaying 336
effect of changing 337
example 318
creating a command to call an application program 346
creating a command to display an output queue 347
creating a command to substitute default value 347
creating abbreviated commands 349
defining a parameter 298
introduction 4
mixed list with 311
object 290
parameter combination, valid 305
processing
qualified name in a CL program 318
prompt text for parameter 294
required parameter for 294
return value for parameter 294
simple list with 305
source list 333
statement
DEP 320
description 290
ELEM 311
error during processing 335
QUAL 317
usage 4
use of qualified name 317
valid parameter by parameter type 305
command documentation, CL
creating
steps 356
defining
description 356
Command Entry display 274
command help, CL
creating
steps 354
defining
description 354
command processing procedure
writing REXX 344
command processing program (CPP)
See also command definition
definition 4
description 291
example 347
writing 342
command prompters
introduction 5
command usage
printing 16
command, CL 13, 16, 30, 32, 34, 35, 40, 72, 158, 311, 333
Add Breakpoint (ADDBKP) 383
Add Library List Entry (ADDLIBLE) 107
Add Message Description (ADDMSGD) 187
defining substitution variables 190
example 196
Add Program (ADDPGM) 380
Add Trace (ADDTRC) 388
ADDBKP (Add Breakpoint) 383
ADDLIBLE (Add Library List Entry) 107
ADDMSGD (Add Message Description)
defining substitution variables 190
example 196
ADDPGM (Add Program) 380
ADDTRC (Add Trace) 388
attribute 332
CALL (Call Program) 69, 75
Call Procedure (CALLPRC) 70
Call Program (CALL) 69, 75
calling
description 82
CALLPRC (Call Procedure) 70, 82
Change Command (CHGCMD) 337
Change Current Library (CHGCURLIB) 107
Change Data Area (CHGDTAARA) 16, 96
Change Debug (CHGDBG) 380
Change Job (CHGJOB) 272
Change Library List (CHGLIBL) 23, 107
Change Message Description (CHGMSGD) 187, 200
Change Message Queue (CHGMSGQ) 202, 205
Change Program Variable (CHGPGMVAR) 392
Change System Library List (CHGSYSLIBL) 107
Change Variable (CHGVAR) 13, 26
changing 337
changing program control command 16
Check Object (CHKOBJ) 16, 147
CHGCMD (Change Command) 337
CHGCURLIB (Change Current Library) 107
CHGDBG (Change Debug) 380
command, CL *(continued)*
CHGDTAARA (Change Data Area) 16, 96
CHGJOB (Change Job) 272
CHGLIBL (Change Library List) 23, 107
CHGMSGD (Change Message Description) 187, 200
CHGMSGQ (Change Message Queue) 202, 205
CHGPGMVAR (Change Program Variable) 392
CHGSYSLIBL (Change System Library List) 107
CHGVAR (Change Variable) 13, 26
CHKOBJ (Check Object) 16, 147
Clear Library (CLRLIB) 115
Clear Trace Data (CLRTRCDTA) 387, 388
CLRLIB (Clear Library) 115
CLRTRCDTA (Clear Trace Data) 387, 388
CMD (Command) statement 293
command processing program (CPP) 291
Convert Date (CVTDAT) 16, 53
CREATE BOUND CONTROL LANGUAGE (Create Bound CL) 16
Create Bound Control Language (CRTBNDCL) 16
Create Command (CRTCMD) 290, 332
Create Control Language Module (CRTCLMOD) 16, 59
Create Data Area (CRTDTAARA) 16, 96
Create Duplicate Object (CRTDUPOBJ) 133
Create Library (CRTLIB) 111
Create Message File (CRTMSGF) 185, 186
Create Message Queue (CRTMSGQ) 202
CREATE PROGRAM 16
Create Program (CRTPGM) 16
CREATE SERVICE PROGRAM 16
Create Service Program (CRTSRVPGM) 16
creating
definition 290
process 332
steps 289
CRTCLMOD (Create Control Language Module) 16, 59
CRTCMD (Create Command) 290, 332
CRTDTAARA (Create Data Area) 16, 96
CRTDUPOBJ (Create Duplicate Object) 133
CRTLIB (Create Library) 111
CRTMSGF (Create Message File) 185, 186
CRTMSGQ (Create Message Queue) 202
CVTDAT (Convert Date) 16, 53
DCL (Declare CL Variable) 13, 16

command, CL *(continued)*
DCLF (Declare File)
description 13
using 152
variables 21
Deallocate Object (DLCOBJ) 141
Declare CL Variable (DCL) 13, 16
Declare File (DCLF)
description 13
using 152
variables 21
defined, authority needed 292
defining
dependent relationship 320
description 289
error encountered 335
example 346
instructions 292
list within list 313
mixed list 311
qualified name 317
source list 333
validity checking 345
Delete Command (DLTMCD) 333
Delete Data Area (DLTDTAARA) 16
Delete File (DLTF) 13
Delete Library (DLTLIB) 115
Delete Program (DLTPGM) 16
Display Breakpoints (DSPBKP) 391
Display Command (DSPCMD) 336
Display Data Area (DSPDTAARA) 16, 96
Display Debug (DSPDBG) 391
Display Job (DSPJOB) 142
Display Job Log (DSPJOBLOG) 279
Display Library (DSPLIB) 116
Display Library Description (DSPLIBD) 117
Display Log (DSPLOG) 283
Display Message Descriptions (DSPMSGD) 187, 196
Display Messages (DSPMSG) 202
Display Object Description (DSPOBJD)
common attributes 99
log-version selection 283
use 119
Display Program Variable (DSPPGMVAR) 391
Display Spooled File (DSPSPLF) 279
Display Trace (DSPTRC) 391
Display Trace Data (DSPTRCDTA) 388, 390
displaying 336
DLCOBJ (Deallocate Object) 141
DLTCMD (Delete Command) 333
DLTDTAARA (Delete Data Area) 16
DLTF (Delete File) 13
DLTLIB (Delete Library) 115
DLTPGM (Delete Program) 16
DO (Do) 35
Do For (DOFOR) 16, 38
Do Until (DOUNTIL) 16, 36
Do While (DOWHILE) 16, 37
DOFOR (Do For) 16, 38
DOUNTIL (Do Until) 16, 36
DOWHILE (Do While) 16, 37
DSPBKP (Display Breakpoints) 391

command, CL *(continued)*
DSPCMD (Display Command) 336
DSPDBG (Display Debug) 391
DSPDTAARA (Display Data Area) 16, 96
DSPJOB (Display Job) 142
DSPJOBLOG (Display Job Log) 279
DSPLIB (Display Library) 116
DSPLIBD (Display Library Description) 117
DSPLOG (Display Log) 283
DSPMSG (Display Messages) 202
DSPMSGD (Display Message Descriptions) 187, 196
DSPOBJD (Display Object Description)
common attributes 99
log-version selection 283
use 119
DSPPGMVAR (Display Program Variable) 391
DSPSPLF (Display Spooled File) 279
DSPTRC (Display Trace) 391
DSPTRCDTA (Display Trace Data) 388, 390
effect of changing definition 337
End Do (ENDDO) 16, 35
End Program (ENDPGM) 13, 16
End Receive (ENDRCV) 157, 159, 160
End Request (ENDRQS) 382
End Select (ENDSELECT) 16, 40
ENDDO (End Do) 16, 35
ENDPGM (End Program) 13, 16
ENDRCV (End Receive) 157, 159, 160
ENDRQS (End Request) 382
ENDSELECT (End Select) 16, 40
example of creating 346
frequently used in CL procedure 16
functions 16
GOTO (Go To) 16, 30
Iterate (ITERATE) 16
ITERATE (Iterate) 16, 39
Leave (LEAVE) 16
LEAVE (Leave) 16, 40
Load and Run Media Program (LODRUN) 181
LODRUN (Load and Run Media Program) 181
logging CL procedure 60
Merge Message File (MRGMSGF) 185, 187
Monitor Message (MONMSG) 50, 239
MONMSG (Monitor Message) 50, 239
Move Object (MOVOBJ) 131
MOVOBJ (Move Object) 131
MRGMSGF (Merge Message File) 185, 187
online help information, providing 354
Otherwise (OTHERWISE) 16
OTHERWISE (Otherwise) 16, 40
Override with Database File (OVRDBF) 13

command, CL *(continued)*
Override with Message File (OVRMSGF) 197
OVRDBF (Override with Database File) 13
OVRMSGF (Override with Message File) 197
Print Command Usage (PRTCMDUSG) 16
processing program (CPP)
definition 4
writing 342
PRTCMDUSG (Print Command Usage) 16
RCLRSC (Reclaim Resources) 381
RCVF (Receive File) 150, 159
RCVMSG (Receive Message) 232, 233
Receive File (RCVF) 150, 159
Receive Message (RCVMSG) 232, 233
Reclaim Resources (RCLRSC) 381
Remove Breakpoint (RMVBKP) 387
Remove Library List Entry (RMVLIBLE) 107
Remove Message (RMVMSG) 16, 238
Remove Message Description (RMVMSGD) 187
Remove Program (RMVPGM) 380
Rename Object (RNMOBJ) 135
Resume Breakpoint (RSMBKP) 384
Retrieve Configuration Source (RTVCFGSRC) 16, 55
Retrieve Configuration Status (RTVCFGSTS) 16, 56
Retrieve Data Area (RTVDTAARA) 16, 96
Retrieve Job Attributes (RTVJOBA) 16, 57
Retrieve Library Description (RTVLIBD) 117
Retrieve Member Description (RTVMBRD) 16, 58
Retrieve Message (RTVMSG) 16, 237
Retrieve Network Attributes (RTVNETA) 56
Retrieve Object Description (RTVOBJD) 57, 123
Retrieve System Value (RTVSYSVAL) 16, 53
Retrieve User Profile (RTVUSRPRF) 16, 58
RMVBKP (Remove Breakpoint) 387
RMVLIBLE (Remove Library List Entry) 107
RMVMSG (Remove Message) 16, 238
RMVMSGD (Remove Message Description) 187
RMVPGM (Remove Program) 380
RNMOBJ (Rename Object) 135
RSMBKP (Resume Breakpoint) 384
RTVCFGSRC (Retrieve Configuration Source) 16, 55
RTVCFGSTS (Retrieve Configuration Status) 16, 56
RTVDTAARA (Retrieve Data Area) 16, 96
RTVJOBA (Retrieve Job Attributes) 16, 57

command, CL *(continued)*
RTVLIBD (Retrieve Library Description) 117
RTVMBRD (Retrieve Member Description) 16, 58
RTVMSG (Retrieve Message) 16, 237
RTVNETA (Retrieve Network Attributes) 56
RTVOBJD (Retrieve Object Description) 57, 123
RTVSYSVAL (Retrieve System Value) 16, 53
RTVUSRPRF (Retrieve User Profile) 16, 58
Select (SELECT) 16
SELECT (Select) 16, 40
selective prompting 170
Send Break Message (SNDBRKMSG) 211
Send File (SNDF) 150, 159
Send Message (SNDMSG) 211
Send Program Message (SNDPGMMSG) 13, 212
Send Reply (SNDRPY) 16, 238
Send User Message (SNDUSRMSG) 16, 213
Send/Receive File (SNDRCVF) 150, 154
setting CL procedure limits command 16
SNDBRKMSG (Send Break Message) 211
SNDF (Send File) 150, 159
SNDMSG (Send Message) 211
SNDPGMMSG (Send Program Message) 13, 212
call stack entry 218
SNDRCVF (Send/Receive File) 150, 154
SNDRPY (Send Reply) 16, 238
SNDUSRMSG (Send User Message) 16, 213
specifying prompt override program
when changing 329
when creating 329
Start Debug (STRDBG) 380, 387
Start Programmer Menu (STRPGMMNU) 174
STRDBG (Start Debug) 380, 387
STRPGMMNU (Start Programmer Menu) 174
TFRCTL (Transfer Control) 399, 400
Transfer Control (TFRCTL) 399, 400
used frequently in CL procedure 16
used in CL procedure 16
using the prompter 170
When (WHEN) 16
WHEN (When) 16, 40
Work with Object Locks (WRKOBJLCK) 142
WRKOBJLCK (Work with Object Locks) 142
command, Display Audit Journal Entries DSPAUDJRNE 113
comment delimiter (/*) 29
communicate
between procedure 92
communicate *(continued)*
using data area 92
compiler error 63
compiler listing
CL procedure 61
sample program 61
compiler, CL
installing support for 67
compiling
source programs 66
completion message 185, 214
compressing
object 136
object table 137
conditional breakpoint
adding 386
example 370
removing 367
setting 367
conditional processing of command 29
conditional prompting 322
configuration source
retrieving 16, 55
configuration status
retrieving 16, 56
constant value
defining for parameter 294
control
transferring
description 399
use 400
control language (CL)
See command, CL
command
definition 1
entering 1
syntax 2
menu
using CL program to control 155
procedure
creating 16, 59
description 3
monitoring for message 239
parts 13
referring to object 145
used within CL 3
program
allowing user changes at run time 168
controlling flow between programs 69
controlling menu 155
controlling processing 20
DBCS data 176
description 11
display file, using 149
display formatting 148
example 15
example program 177
files supported 148
introduction 11
message handling 205
message subfile 168
receiving data 153
receiving message 232
sending data 153
sending message 212
control language (CL) *(continued)*
program *(continued)*
substring built-in function (%SUBSTRING) 318
control processing with CL command 20
controlling
logic flow in CL procedure 29
processing in CL procedure 29
Convert Date (CVTDAT) command 16, 53
converting
date 16, 53
date format 53
CPP (command processing program)
See also command definition
definition 4
description 291
example 346
writing 342
create
creating 16
Create Bound Control Language (CRTBNDCL) command 16
Create Command (CRTCMD) command
CL program 289
example 347
parameters 332
processing 290
relationship 343
Create Control Language Module (CRTCLMOD) command 16, 59
Create Data Area (CRTDTAARA) command 16, 96
Create Duplicate Object (CRTDUPOBJ) command 133
Create Library (CRTLIB) command 111
Create Message File (CRTMSGF) command 185, 186
Create Message Queue (CRTMSGQ) command 202
Create Program (CRTPGM) command 16
Create Service Program (CRTSRVPGM) command 16
creating
CL procedure 16, 59
command
attribute 332
description 290, 332
example 292, 346
create 16
data area 16, 96
duplicate object 133
information for object 125
library 111
message file 185, 186
message queue 202
online help information 354
valid type 95
CRTBNDCL (Create Bound Control Language) command 16
CRTCLMOD (Create Control Language Module) command 16, 59
CRTCMD (Create Command) command
CL program 289
example 347
parameters 332

CRTCMD (Create Command) command *(continued)*
processing 290
relationship 343
CRTDTAARA (Create Data Area) command 16, 96
CRTDUPOBJ (Create Duplicate Object) command 133
CRTLIB (Create Library) command 111
CRTMSGF (Create Message File) command 185, 186
CRTMSGQ (Create Message Queue) command 202
CRTPGM (Create Program) command 16
CRTSRVPGM (Create Service Program) command 16
current library
changing 107
CVTDAT (Convert Date) command 16, 53

D

data area 95
See also object
changing 16, 96
communicate 92
creating 16, 96
deleting 16
description 92
displaying 16, 96
example of retrieving 96
group 94
initial value 92
retrieving 16, 96
data area, command to work with 16
data authority 112
data queue
allocating 87
communicating between programs 82
creating 87
example 88
managing storage 87
sending data 87
using 88
data type error 78
database file 160
overriding 13
receiving data area 160
referring to output file 161
using as data queue 86
date
conversion 16
converting format 53
DBCS (double-byte character set)
defining message 196
designing application program 175
sending message 194
using QCMDEXC with 165
writing CL program with DBCS data 176
DCL (Declare CL Variable) command 13, 16
DCLF (Declare File) command
CL procedure 13, 16
DCLF (Declare File) command *(continued)*
declaring
variable 152
description 21
Deallocate Object (DLCOBJ) command 141
deallocating
object 141
debug
changing 380
command 358
displaying 391
session
adding program object 361
prepare program object 359
removing program object 362
starting 387
debug command
BREAK 368
CLEAR 368
debugger
ILE source 357
debugging 378
batch job not started from job queue 394
batch job submitted to a job queue 393
considerations for one job from another job 395
from another job 393
ILE program 357
ILE source debugger commands 360
interactive job 395
machine interface level 396
running job 394
starting 380
starting ILE source debugger 360
testing applications 379
decimal length error 80
Declare CL Variable (DCL) command 13, 16
Declare File (DCLF) command
CL procedure 13, 16
declaring
variable 23, 152
description 21
declaring
CL variable 13
decompressing
object 136
default delivery of message 203
default handling 244
unmonitored message while testing 381
unmonitored, default handling 381
default program
used in testing 380
default value
changing command 339
defining for parameter 298
message 194
reply 194
default value table 298
defining
CL command table 290
command
authority 292
defining *(continued)*
command *(continued)*
definition 289
parameter 320
statements 292
command documentation
definition 356
command help
definition 354
element in list
simple list 307
list for parameter 305
list within list 313
optional parameter 294
parameter 294
prompt text for a parameter 294
qualified name 317
required parameter 294
restricted value for parameter 294
return value for parameter 294
simple list 307
substitution variable 190
valid parameter 294
definition object, command 290
definition statement, command 290
delete authority 112
Delete Command (DLTCMD) command 333
Delete Data Area (DLTDTAARA) command 16
Delete File (DLTF) command 13
Delete Library (DLTLIB) command 115
Delete Program (DLTPGM) command 16
deleting
command 333
data area 16
file 13
file member 350
HLL programs 350
library 115
object 139
program 16
program object 350
QHST file 288
source member 350
DEP (Dependent) statement
command definition 292
example 320
use 320
detailed message
description 274
detecting unused object on system 125
diagnostic message 185, 214
display 385
breakpoint 385
Command Entry 12
menu, using for command entry 12
programmer menu 15, 173
trace data 388
unmonitored message breakpoint 381
Display Audit Journal Entries (DSPAUDJRNE) command 113
Display Breakpoints (DSPBKP) command 391
Display Call Stack display 236

Display Command (DSPCMD)
command 336
Display Data Area (DSPDTAARA)
command 16, 96
Display Debug (DSPDBG)
command 391
display file
creating 152
receiving 150, 153
referring to 151
sending 153
using in CL program 149
using multiple device displays 157
Display History Log Contents
display 283
Display Job (DSPJOB) command 142
Display Job Log (DSPJOBLOG)
command 279
Display Library (DSPLIB) command 116
Display Library Description (DSPLIBD)
command 117
Display Log (DSPLOG) command 283
Display Message Descriptions
(DSPMSGD) command 187, 196
Display Messages (DSPMSG)
command 202
Display Object Description (DSPOBJD)
command
description 99
log-version selection 283
use 119
Display Program Variable (DSPPGMVAR)
command 391
Display Spooled File (DSPSPLF)
command 279
Display Trace (DSPTRC) command 391
Display Trace Data (DSPTRCDTA)
command 388, 390
displaying 282
batch job log 283
breakpoint 391
command 336
command definition 336
data area 16, 96
debug information 391
history log (QHST) 283
job 142
job log 279
library 116
library description 117
log 283
message 202, 348
message description 187, 196
module attribute 65
object description
common attributes 99
log-version selection 283
use 119
object in library 116
object lock 142
program attribute 65
program variable 391
QHST log 283
spooled file 279
testing information 391
trace 391
trace data 388, 390
displaying *(continued)*
value of variable in a program 391
DLCOBJ (Deallocate Object)
command 141
DLTCMD (Delete Command)
command 333
DLTDTAARA (Delete Data Area)
command 16
DLTF (Delete File) command 13
DLTLIB (Delete Library) command 115
DLTPGM (Delete Program)
command 16
DO (Do) command 16, 35
Do For (DOFOR) command 16, 38
DO group 36
Do Until (DOUNTIL) command 16, 36
Do While (DOWHILE) command 16, 37
documentation aid
listing CL command 61
DOFOR (Do For) command 16, 38
double-byte character set (DBCS)
defining message 196
designing application program 175
sending message 194
using QCMDEXC with 165
writing CL program with DBCS
data 176
double-byte data
defining double-byte message 196
designing application program 175
how to send immediate 194
prompting for in CL program 165
prompting for using QCMDEXC
program 165
sending message that contains
double-byte characters 194
using in CL program 176
double-byte message 196
DOUNTIL (Do Until) command 16, 36
DOWHILE (Do While) command 16, 37
DSPAUDJRNE 113
DSPBKP (Display Breakpoints)
command 391
DSPCMD (Display Command)
command 336
DSPDBG (Display Debug)
command 391
DSPDTAARA (Display Data Area)
command 16, 96
DSPJOB (Display Job) command 142
DSPJOBLOG (Display Job Log)
command 279
DSPLIB (Display Library) command 116
DSPLIBD (Display Library Description)
command 117
DSPLOG (Display Log) command 283
DSPMSG (Display Messages)
command 202
DSPMSGD (Display Message
Descriptions) command 187, 196
DSPOBJD (Display Object Description)
command
description 99
log-version selection 283
use 119
DSPPGMVAR (Display Program Variable)
command 391
DSPSPLF (Display Spooled File)
command 279
DSPTRC (Display Trace) command 391
DSPTRCDTA (Display Trace Data)
command 388, 390
duplicate object
creating 133

E

element
defining in a list 311
Element (ELEM) statement
command definition 292
example 311, 314
use 311
ELSE (Else) command 16, 32
embedded IF (If) command 34
End Do (ENDDO) command 16, 35
End Program (ENDPGM) command
CL procedure 13, 16
example 168
End Receive (ENDRCV) command
multiple device display files 157,
159, 160
End Request (ENDRQS) command 382
End Select (ENDSELECT) command 16,
40
end, abnormal 178
ENDDO (End Do) command 16, 35
ending
program 13, 16
receive 157, 159, 160
request 382
ENDPGM (End Program) command
CL procedure 13, 16
example 168
ENDRCV (End Receive) command
multiple device display files 157,
159, 160
ENDRQS (End Request) command 382
ENDSELECT (End Select) command 16,
40
entry
batch 13
interactive 12
error
calling program 78
character length 81
command definition statement 335
compiler 63
data type 78
decimal length 80
precision 80
procedure 78
escape message
CPF2469 198
definition 185
monitoring 239
sending 215
example
*BCAT value 216
adding
breakpoint to program 384
trace to program 387
ADDMSGD (Add Message
Description) command 196

example *(continued)*
attribute of variable 377
BIN function 45
binary function 45
break-handling program 247
CALL command 69
CALLPRC command 70
change variable
character 376
decimal 376
logical 376
changing
lock state 142
message 215
variable value 26
CL procedure
control processing 20
simple 15
typical 12, 18
CL program
processing qualified name 318
command processing program 347
compiler listing 61
conditional breakpoint 370
controlling menu 155
converting system value 54
creating
CL procedure 15
command 292, 346, 347
command to call application program 346
command to display output queue 347
command to substitute default value 347
creating abbreviated commands 349
CRTMSGF (Create Message File) command 186
data queue 88
DBCS data in CL programs 176
DDS
display file 152
declaring display file 152
defining
parameter 299, 346
prompt text for command name 346
deleting QHST file 288
describing
message 196
display character variable 374
display decimal variable 374
display file 152
display logical variable 374
displaying variables in hexadecimal format 374
DO command 35
DOFOR command 38
DOUNTIL command 36
DOWHILE command 37
ENDDO command 35
ENDSELECT command 40
GOTO command 30
IF (If) command 30
initial program 108
ITERATE command 39
example *(continued)*
LEAVE command 40
logging message in job log 274
logical expression 41
message 196
message handling program 205
monitoring
message for specific command 241
message within procedures 243
moving object 132
nested Do group 36
object
qualified name 7
OTHERWISE (Otherwise) 40
overriding message file 199
passing
control to procedure 70
control to program 69
parameter 78
processing
qualified name in CL program 318
prompt override program 329
QINSTAPP program 181
qualified name of object 7
receiving message from QSYSMSG 267
replacing library list 108
retrieving
data area 96
job attribute 57
network attribute 56
object description 124
system value 53
user profile 58
runtime call stack 220, 230
sample CL program 177
sample default message program 195
saving library list 108
SELECT command 40
sending
message 215
program message 214
SST function 47
substring function 47
switch function 50
TOPGMQ(*PRV*) 221
Transfer Control (TFRTCL) command 400
using *CTLBDY 231
using *PGMBDY 226, 227, 228
using complex name 224
using simple name 223
WHEN (When) 40
exception message
using the RMV keyword 233
exclusive (*EXCL) lock state 140
exclusive allow read (*EXCLRD) lock state 140
execute authority 112
expression
equating a name 377
logical 41
relational 41

F

field definition
QMHCID 407
QMHCRP 410
QMHCSP 409
QMHDAT 405
QMHJBN 411
QMHJDT 404, 410
QMHJOB 409
QMHJTM 404
QMHJTS 411
QMHLIN 411
QMHLNN 411
QMHLSP 410
QMHMDT 409
QMHMF 405
QMHMID 405
QMHMKS 411
QMHMRK 404
QMHMSC 410
QMHPRL 407
QMHRLB 409
QMHRMD 408
QMHRPG 409
QMHRPR 408
QMHRPY 405
QMHRQS 406
QMHRSN 407
QMHRTM 409
QMHRTY 406
QMHSEV 405
QMHSID 411
QMHSLB 408
QMHSMD 407
QMHSPG 408
QMHSPR 407
QMHSSN 407
QMHSTM 408
QMHSTY 406
QMHSYN 411
QMHSYS 409
QMHTID 410
QMHTIM 405
QMHTTY 411
QMHTYP 404
file
See also display file
See also member
See also object
CL procedure
overriding database file 160
overriding display file 156
referring to 151
working with 148
database
closing 151
declaring 151
opening 151
declaring
in CL program 152
name 21
to program 16
variable 13
deleting 13, 349
display
closing 151
declaring 151

file *(continued)*
display *(continued)*
opening 151
name
using as parameter value 294
receiving
data 150, 159
record 16, 154
sending
CL procedure 16
data 150, 159
subfile records 154
file member
deleting 350
filtering
description 274
messages
using severity code filter (SEV) parameter 203
format of date
converting 53
frequently-used objects
description 139
function
CL commands 16
testing
description 9

G

GENCMDDOC (Generate Command Documentation) command
CL program 354, 356
general purpose library (QGPL) 116
Generate Command Documentation (GENCMDDOC) command
CL program 354, 356
generic name
description 109
Get Current Local Time (CEELOCT) 55
GOTO (Go To) command 16, 30

H

handling
default 244
help information
See online help information
help panel group
online help information 354
high-level language (HLL) program 166
mixed list 312
QCMDEXC program 163
history log (QHST)
description 282
format 285
format table 285
version 282
HLL (high-level language) program
mixed list 312
QCMDEXC program 163
hold delivery of message 203
HTML (Hyper Text Markup Language) 356
Hyper Text Markup Language (HTML) 356

I

If (IF) command
CL procedure 16
IF (If) command 16
description 16
embedded 34
example 30
using %SWITCH with 50
ILE (Integrated Language Environment) model
CL program
debugging 357
message queue
call stack entry 207
notify message 215
procedure
receiving 278
sending 278
source debugger 357
starting source debugger 360
immediate message 183
informational message 183, 213
initializing
library list 107
input field length 298
inquiry message 183, 213
installing
CL compiler support 67
instruction, stepping 390
Integrated Language Environment (ILE) model
message queue
call stack entry 207
notify message 215
procedure
receiving 278
sending 278
Integrated Language Environment (ILE) procedure
call stack entry message queue 207
receiving 278
sending 278
interactive
entry 12
job
debugging another 395
job log
consideration 281
Iterate (ITERATE) command 16
ITERATE (Iterate) command 16, 39

J

job
See also batch job
batch
testing functions 378
changing 272
displaying 142
interactive
testing functions 378
submitting 179
job attribute
retrieving 16, 57
job log
consideration for interactive 281
job log *(continued)*
description 272
directing 403
displaying 279
model for primary 403
output file 403
preventing production of 280
suggestions when using 280
job message queue 202, 206
job queue
debugging batch job not started from 394
debugging batch job submitted to 393

K

key parameter
defining 294
identifying 326
using 325

L

label
in CL procedure 30
language
feature code 117
using different 117
LDA (local data area) 93
Leave (LEAVE) command 16
LEAVE (Leave) command 16, 40
length of parameter value table 297
library
allocating resource 139
authority 111
clearing 115
creating 111
definition 6
deleting 115
description 100
displaying
library list 109
names and contents 116
object 116
object description 119
grouping 6
grouping object 110
placing object in 115
previous-release 66
production 111
renaming consideration 135
retrieving object description 57, 123
security 111
test 111
library description
displaying 117
retrieving 117
library list
*CURLIB value 101
accessing object 102
changing 23, 107
comparison with qualified name 104
current library 101, 106
displaying 109

library list *(continued)*
entry
adding 107
removing 107
initializing
QSYSLIBL system value 107
QUSRLIBL system value 107
job 106
part of
current library 101
product library 101
system part description 101
user part 101
product library 106
saving 108
search order 102
setting up 109
system part 106
user part 106
library name
specifying 100
list
CL or HLL for list within 314
CL or HLL for mixed 312
CL or HLL for simple 308
command definition 333
defining 305
REXX
mixed 313
simple 310
within 316
variable to specify 23
list of parameter value
defining 305
elements
using Element (ELEM) statement 311
simple 305
list within list 313
using CL or HLL for 314
using REXX for 316
listing view
using 360
Load and Run Media Program (LODRUN) command 181
local data area 93
lock state
*EXCL (exclusive) 140
*EXCLRD (exclusive allow read) 140
*SHRNUP (shared-no-update) 140
*SHRRD (shared-for-read) 140
*SHRUPD (shared-for-update) 140
combination table 140
exclusive (*EXCL) 140
exclusive allow read (*EXCLRD) 140
object type table 140
shared-for-read (*SHRRD) 140
shared-for-update (*SHRUPD) 140
shared-no-update (*SHRNUP) 140
log
consideration for batch job 282
displaying 283
displaying system 282
history 282
job 272
QHST (history) 282
logging CL procedure command 60
logic control command 13
logical expression 41

M

member
source
deleting 349
member description
retrieving 16, 58
menu
introduction 5
programmer 173
Merge Message File (MRGMSGF) command 185, 187
merging
message file 185, 187
message 201, 282
adding to file 187
assigning message identifier 187
assigning severity code 189
break delivery 202
break-handling program 205
changing delivery mode 205
completion 185
default handling while testing 382
default value 194
defining
description 188
help 188
substitution variable 190
definition 8
delivery 202
describing predefined 187
description
definition 8
diagnostic 185
displaying
break delivery 202
command options 183
double-byte
defining 196
escape
definition 185
description 239
purpose 214
example
changing 215
sending 215
file
IBM-supplied 183
filtering
description 274
handling 183
IBM-supplied message file 183
immediate 8, 183
informational 183, 213
inquiry 183, 213
job message queue 206
logging in history log 272
logging on job log 272
monitoring
description 239
example 16
numeric subtype code 188
use 50
notify 185, 245
message *(continued)*
online help information 188
overriding message file 197
parameters 50
predefined
description 8
IBM-supplied file 183
message queue 183
QHST (history log) file 286
queue 9
receiving
CL procedure 16, 232
CL program 232
removing
CL procedure 16
from message queue 238
reply 185
request 185, 234
retrieving
CL procedure 16
from CL procedure 237
in CL procedure 237
sample program to receive from QSYSMSG 267
sending 183, 211
sending from CL program 212
sending to system user 211
sent to QSYSMSG message queue
CPD4070 250
CPF0907 249, 250
CPF1269 250
CPF1393 250
CPF1397 250
CPF510E 251
CPF5167 251
CPF5244 251
CPF5248 251
CPF5250 251
CPF5251 251
CPF5257 252
CPF5260 252
CPF5274 252
CPF5341 252
CPF5342 253
CPF5344 253
CPF5346 253
CPF5355 253
CPF8AC4 253
CPF9E7C 253
CPI091F 254
CPI0948 254
CPI0949 254
CPI0950 254
CPI0953 254
CPI0954 254
CPI0955 254
CPI095A 254
CPI0964 255
CPI0965 255
CPI0966 255
CPI096B 255
CPI096C 255
CPI096D 255
CPI096E 255
CPI0970 256
CPI0988 256
CPI0989 256

message *(continued)*
sent to QSYSMSG message queue *(continued)*
CPI0998 256
CPI0999 256
CPI099C 256
CPI099D 257
CPI099E 257
CPI099F 258
CPI1117 259
CPI1136 259
CPI1138 259
CPI1139 259
CPI1153 260
CPI1154 260
CPI1159 260
CPI1160 260
CPI1161 260
CPI1162 260
CPI1165 260
CPI1166 260
CPI1167 260
CPI1168 261
CPI1169 261
CPI116A 258
CPI116B 258
CPI116C 259
CPI1171 261
CPI1468 261
CPI2209 261
CPI2239 261
CPI2283 261
CPI2284 262
CPI22AA 261
CPI8A13 262
CPI8A14 262
CPI9014 262
CPI9490 262
CPI94A0 262
CPI94CE 262
CPI94CF 262
CPI94FC 262
CPI96C0 262
CPI96C1 263
CPI96C2 263
CPI96C3 263
CPI96C4 263
CPI96C5 263
CPI96C6 263
CPI96C7 263
CPP0DD9 263
CPP0DDA 263
CPP0DDB 263
CPP0DDC 264
CPP0DDD 264
CPP0DDE 264
CPP0DDF 264
CPP1604 264
CPP29B0 264
CPP29B8 264
CPP29B9 264
CPP29BA 264
CPP951B 264
CPP9522 264
CPP955E 265
CPP9575 265
CPP9576 265
message *(continued)*
sent to QSYSMSG message queue *(continued)*
CPP9589 265
CPP9616 265
CPP9617 265
CPP9618 265
CPP961F 265
CPP9620 265
CPP9621 265
CPP9622 266
CPP9623 266
CPP962B 266
CPPEA02 266
CPPEA04 266
CPPEA05 266
CPPEA12 266
CPPEA13 266
CPPEA26 266
CPPEA32 266
CPPEA38 267
CPPEA39 267
sent to QSYSOPR message queue
CPP0DDD 264
CPP1604 264
CPPEA02 266
CPPEA04 266
CPPEA05 266
CPPEA12 266
CPPEA13 266
CPPEA26 266
CPPEA32 266
CPPEA38 267
CPPEA39 267
size of message file 186
status
definition 185
description 245
using 215
subfile
using 168
text 188
type 183
using system reply list 269
validity checking 192
working with 183, 211
message description
adding 183
example 196
substitution variable 190
to a file 187
value 183
changing 183, 187, 200
definition 8
displaying 187, 196
removing 183, 187
working with 183
message file
changing 183
creating 183, 185, 186
merging 185, 187
overriding with 197
specifying entry size 186
specifying maximum size 185
message files in independent ASPs 186
message help 188
message identifier
specifying 188
message logging levels
detailed 274
high-level 274
message queue
amount of storage 202
call stack entry 207
changing 202, 205
creating 183, 202
QSYSMSG 249
QSYSOPR 204
sending message from program to 212
sending message to 211
working with 183
workstation 204
message queue type table 213
message reference key 232
message subfile 168
message type table 213
message, immediate 8
mixed list
defining 311
description 311
element in list
mixed list 311
passing to CPP 311
using CL or HLL for 312
using REXX for 313
model for primary job log 403
module
description 1
module attribute
displaying 65
module object
changing view 364, 365
Monitor Message (MONMSG) command
in CL procedure 239
use 50
monitoring
message
in CL procedure 239
program level 241
specific command level 241
use 50
MONMSG (Monitor Message) command
in CL procedure 239
use 50
Move Object (MOVOBJ) command 131
moving
object from one library to another 131
MOVOBJ (Move Object) command 131
MRGMSGF (Merge Message File) command 185, 187

N

National Language Sort Sequence (NLSS) 369
national language support 117
National Language Support 377
national language version
definition 117
nested Do group
example 36

nesting
description 380
network attribute
retrieving 56
notify delivery of message 202
notify message
defining 185
monitoring 240, 245
sending 215
number of
number of statement ranges for 388
programs that can be debugged simultaneously 380
statement ranges for trace 388
values in list 294
numeric parameter value
replacing 25
variable replacing 16

O

object
accessing
with library list 101
with qualified name 100
allocating 139
authority verification 99
checking 16, 147
CL procedure
working with 145
command definition 290
common attribute 99
common function table 100
compressing
restriction 137
table 137
use 136
creating 133
information 125
providing description 119
using variable 21
damage detection and notification 99
deallocating 141
decompressing
after operating system installation 138
restrictions 136
temporarily 137
default auditing attribute 115
default public authority 113
definition 5
deleting 139
describing 119
description 99
detecting unused 125
displaying in library 116
duplicate 133
function performed on 99, 100
generic name 109
grouping 6
library 100
lock enforcement 99
lock state 140
module
changing 364
changing view 365
object *(continued)*
moving
restriction 131
moving between libraries 131
moving from test library to production 178
naming 6
placing in library 115
program
adding to debug session 361
prepare for debug session 359
removing from debug session 362
qualified name
description 7
example 7
referring to
in CL procedure 145
object 145
renaming 135
renaming object
restriction 135
restriction
duplicating 134
saving specific 178
searching for multiple 110
searching for single 110
security 111, 113
specific functions 100
TEXT (text) parameter 119
type 99
updating 126
type verification 99
types 5
usage information 126
object authority 111
object description
displaying
log-versions 283
online help 99
use 119
retrieving 57, 123
object existence (*OBJEXIST) authority 111
object lock
working with 142
object management (*OBJMGT) authority 111
object operational (*OBJOPR) authority 111
obtaining
program dump 63
online help information
command 354
help panel group for 354
providing for command 354
operator
arithmetic 41
character 41
logical 41
relational 41
OPM (original program model)
sending or receiving 278
OPM (original program model) program message queue
call stack entry 207
optional parameter
defining 294
original program model (OPM)
sending or receiving 278
original program model (OPM) program message queue
call stack entry 207
OS/400 language support 117
Otherwise (OTHERWISE) command 16
OTHERWISE (Otherwise) command 16, 40
Override with Database File (OVRDBF) command 13
Override with Message File (OVRMSGF) command 197
overriding
database file 13
message file 197
OVRDBF (Override with Database File) command 13
OVRMSGF (Override with Message File) command 197

P

parameter
See command definition
CMD (command) 16
defining 294
consideration 294
constant value 294
default value 298
description 294
determining valid value 294
example 299
keyword, naming 295
optional 294
passing attribute information 294
required 294
restricted value 294
return value 294
type 295
using qualified name 317
valid by parameter type 305
valid combination 305
valid value 294
value length 297
with list within list 313
with mixed list 311
with simple list 305
EXITPGM (exit program) 174
identifying key 326
key 325
order of 72
passing 76, 400
passing attribute information 294
passing between programs 72
possible choice and value 320
receiving 76
restricted value for parameter 294
RQSDTA (request data) 16
RTNCDE (return code) 41
specifying
length returned with value 294
prompt text 294
value length 294
TEXT (text) 119
trailing blanks 27

parameter *(continued)*
type
character (*CHAR) 295
decimal (*DEC) 295
generic name (*GENERIC) 295
integer (*INTn) 295
logical (*LGL) 295
name (*NAME) 295
null (*NULL) 295
path name (*PNAME) 295
statement label 295
valid parameter combination 305
variable name (*VARNAME) 295
valid parameter 294
value
length 297
valid 294
Parameter (PARM) command definition statement
See also parameter
description 292
example 346
use 294
Parameter (PARM) statement
example 299
use 294
parameter combination table 299
parameter value
list of
defining 305
mixed 311
simple 307
replacing 25
PARM (Parameter) command definition statement
See also parameter
description 292
example 346
use 294
PARM (Parameter) statement
example 299
use 294
passing 295
attribute information for a parameter 294
parameter value to CPP 295
character value 295
decimal value 295
generic name 295
list 305
logical value 295
name 295
path name value 295
qualified name 317
variable 295
type
date (*DATE) 295
time (*TIME) 295
percolate 215
performance
benefit
using TFRCTL command 399
consideration 86
data queue advantage 86
message queue 86
performing
calculation
arithmetic 41
character 41
relational 41
PGM (Program) command 13, 16
placing object in library 115
PMTCTL (Prompt Control) command
definition statement 292
precision error 80
predefined message 8, 183
prepare
program object for debug session 359
preventing
display of status message 246
job log 280
production of job log 280
update to files while testing 380
previous release
compiling source programs for 66
install compiler support 67
Print Command Usage (PRTCMDUSG) command 16
printing
command usage 16
procedure
calling
description 70
CL 3
control language (CL) introduction 3
description 1
parts of CL
description 13
working with object 145
receiving message 232
procedure command
logging 60
procedure control command 13
processing
using CL command 20
within CL procedure 29
production library 111, 178
program
See CL program
See also object
See program message
See program variable
activation 381
adding 380
adding breakpoint to 383
adding trace to 387
break-handling 247
breakpoint 383
call 380
calling
CL procedure 16
description 69
use 75
controlling program logic command 16
creating CL 59
default, in testing 380
deleting 16
description 2
dump 63
ending 13, 16
program *(continued)*
number that can be debugged simultaneously 380
placing in debug mode 380
program logic command 16
prompt override program 291
QCMDCHK 166
QCMDEXC 169
removing 380
removing breakpoint from 387
removing trace from 390
service 2
variable
displaying 391
writing command processing procedure 342
writing command processing program 342
writing prompt control 322
writing prompt override 326
writing validity checking 342, 345
Program (PGM) command 13, 16
program attribute
displaying 65
program control command 13
program dump
obtaining 63
program flow 69, 70
program initialization parameter (PIP) data area
program initialization parameter (PIP) 94
program message
changing 183
sending
CL procedure 13
message queue 16, 212
program object
adding to debug session 361
deleting 350
prepare for debug session 359
removing from debug session 362
step into 372
step over 372
stepping into 372
stepping through 371
program source
viewing 364
program variable
changing 392
displaying 391
programmer menu
starting 174
using 173
prompt
text
defining for parameter 294
prompt control 322
prompt override program
allowing for errors 328
CL sample, using 329
description 291
information passed to 326
information returned 327
procedure for using 325
specifying when creating or changing command 329

prompt override program *(continued)*
using key parameter 325
writing 326
prompt parameter 293
prompter
help 13
using 168
prompters 5
prompting
character description table 172
character table 171
conditional 322
for command 170
for double-byte data in a CL program 165
for using QCMDEXC 165
in CL procedure
using 168
with QCMDEXC 173
selective 170
protecting
file from unintentional modification, testing 380
PRTCMDUSG (Print Command Usage) command 16
PRV 66

Q

QBATCH subsystem 282
QCMDCHK program 166
QCMDEXC program
call prompter 169, 173
process command string 108
prompting for double-byte data 165
run command from program 163
QGPL library 116
QHST (history log)
format table 285
message queue 282, 285
processing 286
QHST (history log) file
deleting 288
job completion message 286
job start message 286
QHST (history log) message queue 283, 285
QPJOBLOG (job log) file 279
QRECOVERY library 116
QSYS library 106
QSYSMSG
message queue
CPF0907 249, 250
CPF1269 250
CPF1393 250
CPF1397 250
CPF4070 250
CPF510E 251
CPF5167 251
CPF5244 251
CPF5248 251
CPF5250 251
CPF5251 251
CPF5257 252
CPF5260 252
CPF5274 252
CPF5341 252
QSYSMSG *(continued)*
message queue *(continued)*
CPF5342 253
CPF5344 253
CPF5346 253
CPF5355 253
CPF8AC4 253
CPF9E7C 253
CPI091F 254
CPI0948 254
CPI0949 254
CPI0950 254
CPI0953 254
CPI0954 254
CPI0955 254
CPI095A 254
CPI0964 255
CPI0965 255
CPI0966 255
CPI096B 255
CPI096C 255
CPI096D 255
CPI096E 255
CPI0970 256
CPI0988 256
CPI0989 256
CPI0998 256
CPI0999 256
CPI099C 256
CPI099D 257
CPI099E 257
CPI099F 258
CPI1117 259
CPI1136 259
CPI1138 259
CPI1139 259
CPI1153 260
CPI1154 260
CPI1159 260
CPI1160 260
CPI1161 260
CPI1162 260
CPI1165 260
CPI1166 260
CPI1167 260
CPI1168 261
CPI1169 261
CPI116A 258
CPI116B 258
CPI116C 259
CPI1171 261
CPI1468 261
CPI2209 261
CPI2239 261
CPI2283 261
CPI2284 262
CPI22AA 261
CPI8A13 262
CPI8A14 262
CPI9014 262
CPI9490 262
CPI94A0 262
CPI94CE 262
CPI94CF 262
CPI94FC 262
CPI96C0 262
CPI96C1 263
QSYSMSG *(continued)*
message queue *(continued)*
CPI96C2 263
CPI96C3 263
CPI96C4 263
CPI96C5 263
CPI96C6 263
CPI96C7 263
CPP0DD9 263
CPP0DDA 263
CPP0DDB 263
CPP0DDC 264
CPP0DDD 264
CPP0DDE 264
CPP0DDF 264
CPP1604 264
CPP29B0 264
CPP29B8 264
CPP29B9 264
CPP29BA 264
CPP951B 264
CPP9522 264
CPP955E 265
CPP9575 265
CPP9576 265
CPP9589 265
CPP9616 265
CPP9617 265
CPP9618 265
CPP961F 265
CPP9620 265
CPP9621 265
CPP9622 266
CPP9623 266
CPP962B 266
CPPEA02 266
CPPEA04 266
CPPEA05 266
CPPEA12 266
CPPEA13 266
CPPEA26 266
CPPEA32 266
CPPEA38 267
CPPEA39 267
definition 249
sample program 249
QSYSOPR
message queue
CPP0DDD 264
CPP1604 264
CPPEA02 266
CPPEA04 266
CPPEA05 266
CPPEA12 266
CPPEA13 266
CPPEA26 266
CPPEA32 266
CPPEA38 267
CPPEA39 267
QSYSOPR message queue 202
QUAL (Qualifier) statement
definition 292
example 317, 348
use 317
qualified name
accessing object 100
defining 317

qualified name *(continued)*
example of defining for command 348
passing to CPP 318, 320
processing in CL program 318
specifying 23
specifying with prompting 101
syntax for 100
using CL or HLL 318
using REXX 319
qualifier (QUAL) statement
example 317
Qualifier (QUAL) statement
definition 292
example 348
use 317
queue
changing message queue delivery type 205
external message (*EXT) 206
job message queue 206
message 9, 201
QSYSMSG 249
receiving message from 232
removing message from 238

R

RCLRSC (Reclaim Resources) command 381
RCVF (Receive File) command 150, 159
RCVMSG (Receive Message) command 232, 233
read authority 112
receive
ending 157, 159, 160
Receive File (RCVF) command 150, 159
Receive Message (RCVMSG) command 232, 233
receiving
database file 16, 150
display data 150
file
example 154, 159
message
function 16
in CL procedure 232
in CL program 232
information placement 233
user reply 16
Reclaim Resources (RCLRSC) command 381
reclaiming
resources 381
recovery
after abnormal system end 178
reference key
message 232
relational expression 41
relationship
PARM statement and DCL command 343
part of command definition 343
remote data areas
remote data areas 95
remote data queues
remote data queues 85
Remove Breakpoint (RMVBKP) command 387
Remove Library List Entry (RMVLIBLE) command 107
Remove Message (RMVMSG) command 16, 238
Remove Message Description (RMVMSGD) command 187
Remove Program (RMVPGM) command
breakpoint program 390
traced program 390
using 380
Remove Trace (RMVTRC) command 390
removing
breakpoint 365, 371, 387
breakpoint from program 387
library list entry 107
message 16, 238
message description 187
message from message queue 238
program 380
program object from debug session 362
trace data from system 390
trace from program 390
Rename Object (RNMOBJ) command 135
renaming
object 135
reply
sending 16, 238
reply message 185
reply to message 192
request
ending 382
request data (RQSDTA) parameter 16
request message 185, 234
request processor program
determining existence 236
request-processing procedure
writing 235
required parameter 294
reserved parameter value
replacing 25
variable replacing 16
resource
allocating 139
reclaiming 381
restriction
CL procedure 11
compressing object 137
duplicating objects 134
moving object 131
Resume Breakpoint (RSMBKP) command 384
resuming
breakpoint 384
Retrieve Configuration Source (RTVCFGSRC) command 16, 55
Retrieve Configuration Status (RTVCFGSTS) command 16, 56
Retrieve Data Area (RTVDTAARA) command 16, 96
Retrieve Job Attributes (RTVJOBA) command 16, 56
Retrieve Library Description (RTVLIBD) command 117
Retrieve Member Description (RTVMBRD) command 16, 58
Retrieve Message (RTVMSG) command 16, 237
Retrieve Network Attributes (RTVNETA) command 56
Retrieve Object Description (RTVOBJD) command 57, 123
Retrieve System Value (RTVSYSVAL) command 16, 53
Retrieve User Profile (RTVUSRPRF) command 16, 58
retrieving
configuration source 16, 55
configuration status 16, 56
data area 16, 96
job attribute 16, 57
library description 117
member description 16, 58
message 16, 237
message in CL procedure 237
network attribute 56
object description 57, 123
program attribute 181
program creation commands 16
system value 16, 53
user profile 16, 58
user profile attribute 58
Return (RETURN) command 16, 72
RETURN (Return) command 16, 72
return code
BASIC program 41
CL procedure 41
parameter 41
Pascal program 41
PL/I program 41
RPG IV program 41
summary 41, 65
return code (RTNCDE) parameter 41
REXX procedure
list within list 316
using
for mixed list 313
for qualified name 319
for simple list 310
writing command processing procedure 344
RMVBKP (Remove Breakpoint) command 387
RMVLIBLE (Remove Library List Entry) command 107
RMVMSG (Remove Message) command 16, 238
RMVMSGD (Remove Message Description) command 187
RMVPGM (Remove Program) command
breakpoint program 390
traced program 390
using 380
RMVTRC (Remove Trace) command 390
RNMOBJ (Rename Object) command 135
root source view
using 359
RQSDTA (request data) parameter 16
RSMBKP (Resume Breakpoint) command 384

RTNCDE (return code) parameter 41
RTVCFGSRC (Retrieve Configuration Source) command 16, 55
RTVCFGSTS (Retrieve Configuration Status) command 16, 56
RTVDTAARA (Retrieve Data Area) command 16, 96
RTVJOBA (Retrieve Job Attributes) command 16, 56
RTVLIBD (Retrieve Library Description) command 117
RTVMBRD (Retrieve Member Description) command 16, 58
RTVMSG (Retrieve Message) command 16, 237
RTVNETA (Retrieve Network Attributes) command 56
RTVOBJD (Retrieve Object Description) command 57, 123
RTVSYSVAL (Retrieve System Value) command 16, 53
RTVUSRPRF (Retrieve User Profile) command 16, 58
run time
  allowing user changes to CL commands 168

S

sample program to receive message from QSYSMSG 267
searching
  for object 109
securing
  object 111
security
  for object 113
see='breakpoint program'.breakpoint 384, 385, 387
  adding to program 383
  using within trace 390
see='breakpoint'.debug mode 380, 391, 396
see='breakpoint'.trace 387, 388
  description 387
  removing from a program 390
  removing information from system 390
  see='breakpoint program'.breakpointdisplaying 391
  using breakpoint within trace 390
see='member'.database file
  preventing, update in production library 380
see='message queue'.message 381
see='testing'.debug mode 391, 396
  adding program 380
  placing program 380
see='trace'.breakpoint 383, 384, 385, 390
  displaying location 391
  removing from program 387
see='trace'.debug mode 380
  security consideration 396
see='user profile'.security
  debugging consideration 396
Select (SELECT) command 16
SELECT (Select) command 16, 40
selective prompting
  character description table 172
  character table 171
  description 170
Send Break Message (SNDBRKMSG) command 211
Send File (SNDF) command
  canceling request for input 159
  CL procedure 16
  function 150
Send Message (SNDMSG) command 211
Send Message (SNDMSG) display 166
Send Program Message (SNDPGMMSG) command
  CL procedure 13, 16
  use 212
Send Reply (SNDRPY) command 16, 238
Send User Message (SNDUSRMSG) command 16, 213
Send/Receive File (SNDRCVF) command
  CL procedure 16
  function 150
  use 154
sending
  break message 211
  data to display 150
  display file 16, 150
  file
    data 154
    example 159
  message 211, 215
  message to system user 211
  program message 13, 212
  reply 16, 238
  user message 16, 213
service program 2
session
  debug
    adding program object 361
    removing program object 362
setting
  breakpoint 365
severity code 189
shared-for-read (*SHRRD) lock state 140
shared-for-update (*SHRUPD) lock state 140
shared-no-update (*SHRNUP) lock state 140
simple list
  parameter value
    defining 307
    description 307
    passing to CPP 307
  using CL or HLL for 308
  using REXX for 310
skip value
  definition 386
SNDBRKMSG (Send Break Message) command 211
SNDF (Send File) command
  canceling request for input 159
  CL procedure 16
  function 150
SNDMSG (Send Message) command 211
SNDPGMMSG (Send Program Message) command
  CL procedure 13, 16
  use 212
SNDRCVF (Send/Receive File) command
  CL procedure 16
  function 150
  use 154
SNDRPY (Send Reply) command 16, 238
SNDUSRMSG (Send User Message) command 16, 213
source debugger
  ILE
    starting 360
source list
  command definition 333
source member
  deleting 349
source view
  working with 378
spooled file
  displaying 279
stack, call
  description 380
  displaying testing information 391
  relationship to CALL command 70
  relationship to CALLPRC command 71
  removing call 70, 71
  removing request in error 382
Start Debug (STRDBG) command
  adding program 380
  example 379
  preventing update to file 380
start position for compare date 270
Start Programmer Menu (STRPGMMNU) command 174
starting
  debug 380, 387
  ILE source debugger 360
  programmer menu 174
statement
  command definition 290
statement combination table 301
statement view
  using 360
static variable
  description 392
status message
  definition 185
  monitoring 245
  preventing display 246
  receiving 240
  sending 215
step into debug command 372
step over debug command 372
STRDBG (Start Debug) command
  adding program 380
  example 379
  preventing update to file 380
STRPGMMNU (Start Programmer Menu) command
  using 174
subfile
  message 168

submitting
job 179
substitution variable 190
substring function
description 47
processing qualified name 318
switch function 49
syntax
command 2
syntax checking 166
system library (QSYS) 106, 116
system library list
changing 107
system log
See also system value
naming version 283
system operator (QSYSOPR) message queue 202, 205
system reply list 269
system user
sending messages to 211
system value
retrieving 16, 53

T

test library 111, 178
testing
canceling request during 382
debug mode 379
default program 380
testing function
description 9
TFRCTL (Transfer Control) command 399, 400
timing out 179, 180
trace
displaying 391
trace data
clearing 387
displaying 388
trailing blank
command parameter 27
example 28
Transfer Control (TFRCTL) command 399, 400
transferring
control 399, 400

U

UIM (user interface manager) 354
unconditional branching 29
unconditional breakpoint
removing 366
setting 366
unmonitored message
breakpoint display 382
handling 381
update authority
update 112
updating
usage information 126
usage information
no updating 130
table 126
usage information *(continued)*
updating 126
User interface manager (UIM) 354
user message
sending
CL procedure 16
function 183
informational 213
inquiry 213
user profile attribute
retrieving 16, 58
using
listing view 360
QCMDCHK program 166
root source view 359
statement view 360

V

validity checking
program 290
reply 192
writing 345
value
parameter 320
variable
changing
CL procedure 13, 16
example 26, 217
value in program 392
value of 26, 375
creating object 21
declaring
description 23
for field 152
for file 152
definition 21
displaying 373
displaying value in program 391
equating a name 377
indicator declared as variable 150
lowercase character in 24
replacing parameter value 25
retrieving system value 53
specifying list 23
specifying qualified name 23
substitution 190
value used as 53
working with 21
view
program source 364

W

Wait (WAIT) command 16, 158
WAIT (Wait) command 16, 158
When (WHEN) command 16
WHEN (When) command 16, 40
Work with Object Locks (WRKOBJLCK) command 142
working with
messages 211
object locks 142
working with message 16
workstation message queue 202
writing
comment in CL procedure 29
request-processing procedure 235
REXX command processing procedure 344
WRKOBJLCK (Work with Object Locks) command 142

Readers' Comments — We'd Like to Hear from You

iSeries
CL Programming
Version 5

Publication No. SC41-5721-06

Overall, how satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Overall satisfaction	☐	☐	☐	☐	☐

How satisfied are you that the information in this book is:

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Accurate	☐	☐	☐	☐	☐
Complete	☐	☐	☐	☐	☐
Easy to find	☐	☐	☐	☐	☐
Easy to understand	☐	☐	☐	☐	☐
Well organized	☐	☐	☐	☐	☐
Applicable to your tasks	☐	☐	☐	☐	☐

Please tell us how we can improve this book:

Thank you for your responses. May we contact you? ☐ Yes ☐ No

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

Name

Address

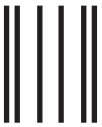
Company or Organization

Phone No.

Readers' Comments — We'd Like to Hear from You
SC41-5721-06

Cut or Fold
Along Line

Fold and Tape | **Please do not staple** | Fold and Tape



NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

IBM CORPORATION
ATTN DEPT 542 IDCLERK
3605 HWY 52 N
ROCHESTER MN 55901-7829

Fold and Tape | **Please do not staple** | Fold and Tape

Cut or Fold
Along Line

Printed in USA

SC41-5721-06