IBM i
7.2

Programming
CL overview and concepts

Note

Before using this information and the product it supports, read the information in “Notices” on page 613.

This edition applies to IBM i 7.2 (product number 5770-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 document may contain references to Licensed Internal Code. Licensed Internal Code is Machine Code and is licensed to you under the terms of the IBM License Agreement for Machine Code.

© Copyright International Business Machines Corporation 1998, 2013.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Control language 1
Control language overview 1
What's new for IBM i 7.3 1
CL concepts 3
System operation control 3
Control language 3
Menus 3
Messages 3
Message descriptions 4
Message queues 4
CL commands 4
CL command names 5
Abbreviations used in CL commands and keywords 5
CL command parts 53
CL command syntax 54
CL command label 54
CL command parameters 55
CL command delimiter characters 55
CL command continuation 57
CL command comments 58
CL command definition 58
CL command coding rules 59
CL command information and documentation 61
CL command documentation format 61
CL command help 64
Printing CL command descriptions on the system 64
CL command prompters 65
CL commands that operate on IBM i objects 65
CL commands that operate on multiple objects 65
CL programs and procedures 66
CL procedure 68
CL module 68
CL program 68
Service program 68
CL parameters 68
Parameter values 68
Constant values 69
Variable name 74
Expressions 75
List of values 75
Parameters in keyword and positional form 76
Required, optional, and key parameters 78
Commonly used parameters 78
AUT parameter 78
CLS parameter 79
COUNTRY parameter 81
FILETYPE parameter 88
FRCRATIO parameter 89
IGCFEAT parameter 90
JOB parameter 91
LABEL parameter 92

LICOPT parameter....93
MAXACT parameter....97
OBJ parameter....98
OBJTYPE parameter....98
OUTPUT parameter....99
PRTTXT parameter....100
REPLACE parameter....101
JOBPTY, OUTPTY, and PTYLMT scheduling priority parameters....102
SEV parameter....103
SPLNBR parameter....105
TEXT parameter....105
VOL parameter....106
WAITFILE parameter....107
Parameter values used for testing and debugging....108
Program-variable description....108
Basing-pointer description....109
Subscript description....109
Qualified-name description....110
Control language elements....111
CL character sets and values....111
Character sets....111
Special character use....112
Symbolic operators....113
Predefined values....114
Expressions in CL commands....115
Arithmetic expressions....115
Character string expressions....116
Relational expressions....118
Logical expressions....119
Operators in expressions....119
Priority of operators when evaluating expressions....121
Built-in functions for CL....121
Naming within commands....123
Folder and document names....124
IBM i objects....126
Library objects....127
External object types....129
Simple and qualified object names....133
Generic object names....133
Object naming rules....135
Communication names (*CNAME)....137
Generic names (*GENERIC)....137
Names (*NAME)....137
Path names (*PNAME)....139
Simple names (*SNAME)....140
Additional rules for unique names....141
Database files and device files used by CL commands....141
CL programming....181
Process for creating a CL program or CL procedure....182
Interactive entry....182
Batch entry....183
Parts of a CL source program....183
Example: Simple CL program....185
Commands used in CL programs or procedures....186
Common commands used in CL programs and procedures....187
Operations performed by CL programs or procedures....190
Variables in CL commands....193
Declaring variables to a CL program or procedure....194

Uses for based variables ... 196
Uses for defined variables ... 196
Variables to use for specifying a list or qualified name ... 197
Cases of characters in variables ... 198
Variables that replace reserved or numeric parameter values ... 198
Changing the value of a variable ... 199
Trailing blanks on command parameters ... 204
Writing comments in CL programs or procedures ... 205
Controlling processing within a CL program or CL procedure ... 206
GOTO command and command labels in a CL program or procedure ... 206
IF command in a CL program or procedure ... 207
ELSE command in a CL program or procedure ... 209
Embedded IF commands in a CL program or procedure ... 211
DO command and DO groups in a CL program or procedure ... 212
Showing DO and SELECT nesting levels ... 214
DOUNTIL command in a CL program or procedure ... 215
DOWHILE command in a CL program or procedure ... 215
DOFOR command in a CL program or procedure ... 216
ITERATE command in a CL program or procedure ... 217
LEAVE command in a CL program or procedure ... 218
CALLSUBR command in a CL program or procedure ... 218
SELECT command and SELECT groups in a CL program or procedure ... 219
SUBR command and subroutines in a CL program or procedure ... 220
*AND, *OR, and *NOT operators ... 221
%ADDRESS built-in function ... 225
%BINARY built-in function ... 226
%CHAR built-in function ... 228
%CHECK built-in function ... 229
%CHECKR built-in function ... 230
%DEC built-in function ... 231
%INT built-in function ... 233
%LEN built-in function ... 234
%LOWER built-in function ... 235
%OFFSET built-in function ... 236
%SCAN built-in function ... 237
%SIZE built-in function ... 238
%SUBSTRING built-in function ... 239
%SWITCH built-in function ... 242
%SWITCH with the IF command ... 243
%SWITCH with the **Change Variable** command ... 243
%TRIM built-in function ... 243
%TRIML built-in function ... 244
%TRIMR built-in function ... 245
%UINT built-in function ... 246
%UPPER built-in function ... 248
Monitor Message command ... 249
Retrieving values that can be used as variables ... 251
Retrieving system values ... 251
Example: Retrieving QTIME system value ... 251
Retrieving the QDATE system value into a CL variable ... 252
Retrieving configuration source ... 254
Retrieving configuration status ... 254
Retrieving network attributes ... 254
Example: Using the Retrieve Network Attributes command ... 254
Retrieving job attributes ... 255
Example: Using the Retrieve Job Attributes command ... 255
Retrieving user profile attributes ... 256
Example: Using the Retrieve User Profile command ... 256

Retrieving member description information....256
Example: Using the Retrieve Member Description command....257
Compiling CL source program....257
Setting create options in the CL source program....258
Embedding CL commands from another source member....259
Logging CL program or procedure commands....260
Retrieving CL source code....261
CL module compiler listings....262
Common compilation errors....264
Obtaining a CL dump....265
Displaying module attributes....266
Displaying program attributes....266
Return code summary....267
Compiling source programs for a previous release....268
Previous-release (*PRV) libraries....268
Installing CL compiler support for a previous release....269
Controlling flow and communicating between programs and procedures....269
Passing control to another program or procedure....269
Using the Call Program command to pass control....270
Using the Call Bound Procedure command to pass control....271
Using the Return command to pass control....272
Passing parameters....273
Using the Call Program command to pass control to a called program....276
Common errors when calling programs and procedures....278
Communicating between programs and procedures....282
Using data queues....282
Remote data queues....284
Comparisons with using database files as queues....285
Similarities to message queues....286
Prerequisites for using data queues....286
Managing the storage used by a data queue....286
Allocating data queues....287
Examples: Using a data queue....287
Creating data queues associated with an output queue....291
Creating data queues associated with jobs....291
Using data areas....292
Local data area....292
Group data area....293
Program Initialization Parameter data area....294
Remote data areas....294
Creating a data area....295
Data area locking and allocation....295
Displaying a data area....296
Changing a data area....296
Retrieving a data area....296
Examples: Retrieving a data area....296
Example: Changing and retrieving a data area....298
Defining and documenting CL commands....298
Defining CL commands....298
CL command definition statements....299
CL command definition process....301
Defining a CL command....305
Data type and parameter restrictions....312
Defining a CL command list parameter....319
Specifying prompt control for a CL command parameter....334
Key parameters and prompt override programs for a CL command....337
Creating a CL command....344
Displaying a CL command definition....348

Effect of changing the command definition of a CL command in a procedure or program...349
Command processing program for a CL command...354
Validity checking program for a CL command...357
Examples: Defining and creating CL commands...358
Documenting a CL command... 364
CL commands and command online help... 364
Writing CL command help...365
Generating HTML source for CL command documentation...367
Proxy CL commands... 368
Command-related APIs... 368
QCAPCMD program...368
QCMDEXC program...369
QCMDEXC program with DBCS data...371
QCMDCHK program... 372
Prompting for user input at run time... 373
Using the IBM i prompter within a CL procedure or program...374
Using selective prompting for CL commands... 375
Using QCMDEXC with prompting in CL procedures and programs... 378
Entering program source... 379
Using the Start Programmer Menu command... 379
Using the EXITPGM parameter of the Start Programmer Menu command...380
Command analyzer exit points... 380
Designing application programs for DBCS data... 380
Designing DBCS application programs... 380
Converting alphanumeric programs to process DBCS data... 381
Using DBCS data in a CL program... 381
Unicode support in control language...382
Unicode overview ...382
Design of Unicode in control language...383
Example: Passing the EBCDIC and Unicode value... 383
Calling Unicode-enabled commands...384
Loading and running an application from tape or optical media... 385
Example: QINSTAPP program... 385
Transferring control to improve performance... 386
Example: Using the Transfer Control command... 386
Passing parameters using the Transfer Control command... 387
Examples: CL programming... 388
Example: Initial program for setup (programmer)... 389
Example: Saving specific objects in an application (system operator)...389
Example: Recovery from abnormal end (system operator)...389
Example: Timing out while waiting for input from a device display...390
Example: Performing date arithmetic... 391
Debugging CL programs and procedures...391
Debugging ILE programs...392
Debug commands...393
Preparing a program object for a debug session... 394
Using a root source view to debug ILE programs...395
Using a listing view to debug ILE programs... 395
Encrypting the debug listing view...395
Using a statement view to debug ILE programs... 396
Starting the ILE source debugger...396
Adding program objects to a debug session...397
Removing program objects from a debug session...398
Viewing the program source...399
Changing a module object... 400
Changing the module object view...401
Setting and removing breakpoints... 401
Setting and removing unconditional breakpoints... 402

Setting and removing conditional breakpoints ... 403
Using the Work with Breakpoints display ... 403
Using the BREAK and CLEAR debug commands to set and remove conditional breakpoints ... 404
National Language Sort Sequence ... 404
Examples: Conditional breakpoint ... 406
Removing all breakpoints ... 406
Using instruction stepping ... 406
F10 (Step) to step over program objects or F22 (Step into) to step into program objects ... 407
Using the STEP debug command to step through a program object ... 407
Displaying variables ... 408
Example: Displaying logical variable ... 409
Examples: Displaying character variable ... 409
Example: Displaying decimal variable ... 409
Example: Displaying variables as hexadecimal values ... 409
Changing the value of variables ... 410
Example: Changing logical variable ... 411
Examples: Changing character variable ... 411
Examples: Changing decimal variable ... 411
Displaying variable attributes ... 412
Equating a name with a variable, an expression, or a command ... 412
Source debug and IBM i globalization ... 413
Working with *SOURCE view ... 413
Operations that temporarily remove steps ... 413
Debugging original program model programs ... 413
Starting debug mode ... 414
Adding programs to debug mode ... 415
Preventing updates to database files in production libraries ... 416
Displaying the call stack ... 416
Program activations ... 416
Handling unmonitored messages ... 417
Breakpoints ... 418
Adding breakpoints to programs ... 418
Adding conditional breakpoints ... 421
Removing breakpoints from programs ... 422
Traces ... 422
Adding traces to programs ... 423
Using instruction stepping ... 426
Using breakpoints within traces ... 426
Removing trace information from the system ... 426
Removing traces from programs ... 426
Displaying testing information ... 426
Displaying the values of variables ... 427
Changing the values of variables ... 428
Reasons for using a job to debug another job ... 429
Debugging batch jobs that are submitted to a job queue ... 429
Debugging batch jobs that are not started from job queues ... 430
Debugging a job that is running ... 430
Debugging another interactive job ... 431
Considerations when debugging one job from another job ... 431
Debugging at the machine interface level ... 432
Security considerations ... 432
Operations that temporarily remove breakpoints ... 432
Objects and libraries ... 433
Objects ... 433
Object types and common attributes ... 433
Functions performed on objects ... 434
Functions the system performs automatically ... 434

Functions you can perform using commands ... 434
Libraries ... 435
Library lists ... 435
Functions of using a library list ... 436
A job's library list ... 440
Changing the library list ... 441
Considerations for using a library list ... 443
Displaying a library list ... 443
Using library lists to search for objects ... 443
Using libraries ... 444
Creating a library ... 445
Authority for libraries specification ... 446
Object authority ... 446
Data authority ... 446
Combined authority ... 446
Security considerations for objects ... 447
The Display Audit Journal Entries command to generate security journal audit reports ... 447
Setting default public authority ... 448
Setting default auditing attribute ... 449
Placing objects in libraries ... 449
Deleting and clearing libraries ... 450
Displaying library names and contents ... 451
Displaying and retrieving library descriptions ... 451
Changing national language versions ... 452
Static prompt message for control language ... 452
Dynamic prompt message for control language ... 453
Describing objects ... 454
Displaying object descriptions ... 454
Retrieving object descriptions ... 458
Example: Using the Retrieve Object Description command ... 460
Creation information for objects ... 460
Detecting unused objects on the system ... 461
Moving objects from one library to another ... 466
Creating duplicate objects ... 468
Renaming objects ... 470
Object compression or decompression ... 471
Restrictions for compression of objects ... 472
Temporarily decompressed objects ... 472
Automatic decompression of objects ... 473
Deleting objects ... 474
Allocating resources ... 474
Lock states for objects ... 476
Displaying the lock states for objects ... 477
Accessing objects in CL programs ... 478
Accessing command definitions, files, and procedures ... 479
Accessing command definitions ... 479
Accessing files ... 480
Accessing procedures ... 480
Checking for the existence of an object ... 480
Working with files in CL programs or procedures ... 481
Data manipulation commands ... 483
Files in a CL program or procedure ... 484
Opening and closing files in a CL program or procedure ... 484
Declaring a file ... 485
Sending and receiving data with a display file ... 486
Example: Writing a CL program or procedure to control a menu ... 488
Overriding display files in a CL procedure or program (OVRDSPF command) ... 489
Working with multiple device display files ... 490

Receiving data from a database file (RCVF command) 493
Reading a file multiple times (CLOSE command) 493
Overriding database files in a CL procedure or program (OVRDBF command) 493
Output files from display commands 494
Messages 495
Defining message descriptions 496
Creating a message file 498
Message files in independent ASPs 498
Determining the size of a message file 498
Adding messages to a file 499
Assigning a message identifier 500
Defining messages and message help 501
Defining substitution variables 501
Assigning a severity code 504
Specifying validity checking for replies 504
Example: Sending an immediate message and handling a reply 505
Defining default values for replies 506
Specifying default message handling for escape messages 507
Example: Describing a message 508
Defining double-byte messages 509
Viewing messages 509
Message file searching 509
System message file searches 510
Overriding message files 510
Message queues 514
Types of message queues 514
Creating or changing a message queue 515
Message queues in independent ASPs 517
Message queues in break mode 517
Placing a message queue in break mode automatically 517
Job message queues 518
External message queue 518
Call message queue 519
Commands used to send messages to a system user 521
Commands used to send messages from a CL program 522
Inquiry and informational messages 524
Completion and diagnostic messages 524
Status messages 525
Escape and notify messages 525
Examples: Sending messages 526
Identifying a call stack entry 528
Using the Send Program Message command as the base 530
Identifying the base entry by name 532
Using the program boundary as a base (*PGMBDY) 534
Using the most recently called procedure as a base (*PGMNAME) 538
Using a control boundary as a base (*CTLBDY) 539
Considerations for service programs 541
Receiving messages into a CL procedure or program 542
Receiving request messages 543
Writing request-processor procedures and programs 546
Determining if a request processor exists 547
Retrieving message descriptions from a message file 548
Removing messages from a message queue 550
Monitoring for messages in a CL program or procedure 551
Watching for messages 554
CL handling for unmonitored messages 554
Monitoring for notify messages 556
Monitoring for status messages 557

Preventing the display of status messages 557
Receiving a message from a program or procedure that has ended 558
Break-handling programs 560
Ways of handling replies to inquiry messages 563
Using a sender copy message to obtain a reply 563
Finding the job that sent the reply 563
Using the system reply list 563
Using the reply handling exit program 567
Message subfiles in a CL program or procedure 567
Log messages 567
Job log 567
Writing a job log to a file 568
Controlling information written in a job log 568
Job log message filtering 569
Example: Controlling information written in a job log 569
Job log sender or receiver information 574
Displaying a job log 575
Preventing the production of job logs 576
Job log considerations 576
Interactive job log considerations 577
Batch job log considerations 578
Message filtering through the Control Job Log Output API 579
Job log output files 579
QHST history log 587
Format of the history log 589
QHST file processing 591
QHST job start and completion messages 591
QHST files deletion 592
QSYSMSG message queue 593
Messages sent to QSYSMSG message queue 593
Example: Receiving messages from QSYSMSG 609

Notices 613
Programming interface information 614
Trademarks 614
Terms and conditions 615

Control language

Control language (CL) allows system programmers and system administrators to write programs using IBM® i commands and other IBM-supplied commands.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 610.

Control language overview

The control language topic collection contains information about all control language (CL) commands that are part of the IBM i operating system. It also includes many CL commands for licensed programs that run on the IBM i operating system.

Before using CL commands, you should be familiar with the CL programming concepts discussed in the Control language topic collection.

Who should use commands

CL commands are typically used by programmers, data processing managers, and system administrators. Through IBM i command menus, powerful command prompter tools, and online command help, nontechnical users can also use CL commands. To use CL commands, you should have a general understanding of the IBM i operating system. If you want to combine several CL commands together to create a CL program, some background in programming is helpful.

How this information is organized

In the CL command finder, you can search for commands by product, by command name, by descriptive name, or by part of the name. You also can search for new commands and changed commands. You can also view an alphabetical list of all of the commands.

Related reference

PDF file for Control language
You can view and print a PDF file of this information.

What's new for IBM i 7.3

Read about new or significantly changed information for the Control language topic collection.

New built-in function %PARMS support for ILE CL

- %PARMS built-in function

Like other CL built-in functions, %PARMS built-in function can only be used in compiled CL programs or procedures. You can specify a `TGTRLS (Target release)` value of `V7R2M0` or `V7R1M0` when compiling your CL source that uses these new built-in function. The CL program can then be saved off your 7.3 system and restored and run on a system running IBM i 7.2 or IBM i 7.1.

IFS source files support for ILE CL

The ILE CL compiler can compile both main source files and INCLUDE files from the IFS. The CRTCLMOD, CRTBNDCL and INCLUDE CL commands are enhanced to support IFS file names.

IFS source files support for defining CL commands

You can define CL commands from the IFS source files. The CRTCMD CL command is enhanced to support IFS file names.

New and changed commands

© Copyright IBM Corp. 1998, 2013

You can display a list of all new commands or all changed commands using the CL command finder.

How to see what's new or changed

To help you see where technical changes have been made, this information uses:

- The » image to mark where new or changed information begins.
- The « image to mark where new or changed information ends.

In PDF files, you might see revision bars (|) in the left margin of new and changed information.

To find other information about what's new or changed this release, see the Memo to users.

PDF file for Control language

You can view and print a PDF file of this information.

To view or download the PDF version of overview and concept information for control language commands, select CL overview and concepts (about 4200 KB). PDFs for CL command descriptions are no longer provided. For information about printing command information from your system, see “Printing CL command descriptions on the system” on page 64.

Saving PDF files

To save a PDF on your workstation for viewing or printing:

1. Right-click the PDF link in your browser.
2. Click the option that saves the PDF locally.
3. Navigate to the directory in which you want to save the PDF.
4. Click **Save**.

Downloading Adobe Reader

You need Adobe Reader installed on your system to view or print these PDFs. You can download a free copy from the Adobe Web site (http://get.adobe.com/reader/) .

Related concepts

Control language overview

The control language topic collection contains information about all control language (CL) commands that are part of the IBM i operating system. It also includes many CL commands for licensed programs that run on the IBM i operating system.

Printing CL command descriptions on the system

To print the parameter and value descriptions for a CL command, follow these instructions.

CL concepts

The control language (CL) is the set of all commands with which a user requests system functions. This information describes the basic concepts of CL that you need to understand before you use the commands.

System operation control

The control language (CL), menu options, and system messages control the operation of the system.

Control language

Control language (CL) is the primary interface to the operating system. CL can be used at the same time by users at different workstations, in batch and interactive jobs and in CL programs and procedures.

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.

Related concepts

CL commands

A control language (CL) command is a single statement to request a system function.

Menus

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

Menus provide the following advantages:

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

Related information

Application Display Programming

IBM i globalization

Messages

A message is a communication sent from one user, program, or procedure to another.

System messages can report both status information and error conditions. Most data processing systems provide communications between the system and the operator to handle errors and other conditions that occur during processing. The IBM i operating system 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 IBM i 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.

Related tasks

Messages
Messages are used to communicate between users and programs.

Related information

IBM i globalization

Message descriptions

A message description defines a message to the IBM i operating system.

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.

The IBM i operating system provides message queues for the following items:

- 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.

CL commands

A control language (CL) command is a single statement to request a system function.

Related concepts

Control language

Control language (CL) is the primary interface to the operating system. CL can be used at the same time by users at different workstations, in batch and interactive jobs and in CL programs and procedures.

CL command names

The command name identifies the function that will be performed by the program that is called when the command is run. Most command names consist of a combination of a verb (or action) followed by a noun or phrase that identifies the receiver of the action (or object being acted on): (command = verb + object acted on).

Abbreviated words, typically one to three letters, make up the command name. This reduces the amount of typing that is required to enter the command.

For example, you can create, delete, or display a library; so the verb abbreviations CRT, DLT, and DSP are joined to the abbreviation for library, LIB. The result is three commands that can operate on a library: **CRTLIB**, **DLTLIB**, and **DSPLIB**. In another example, one of the CL commands is the Send Message command. You would use the **Send Message (SNDMSG)** command to send a message from a user to a message queue.

The conventions for naming the combination verb and object commands are as follows:

- The primary convention (as just shown) is to use three letters from each word in the descriptive command name to form the abbreviated command name that is recognized by the system.
- The secondary convention is to use single letters from the ending word or words in the command title for the end of the command name, such as the three single letters DLO on the **Delete Document Library Object (DLTDLO)** command.
- An occasional convention is to use single letters in the middle of the command name (typically between commonly used three-character verbs and objects), such as the letters CL in the **Create CL Program (CRTCLPGM)** command.

Some command names consist of the verb only, such as the **Move (MOV)** command, or an object only, such as the **Data (DATA)** command. A few commands have an IBM i command name, and can also be called using one or more alternate names that may be familiar to users of systems other than the IBM i system. An alternate name is known as an *alias*, such as the name CD is an alias for the **Change Current Directory (CHGCURDIR)** command.

Related concepts

Simple and qualified object names
The name of a specific object that is located in a library can be specified as a simple name or as a qualified name.

Object naming rules
These rules are used to name all IBM i objects used in control language (CL) commands. The parameter summary table for each CL command shows whether a simple object name, a qualified name, or a generic name can be specified.

Related reference

OBJTYPE parameter
The object type (OBJTYPE) parameter specifies the types of IBM i objects that can be operated on by the command in which they are specified.

Abbreviations used in CL commands and keywords

Most CL command names that are part of the IBM i operating system and other licensed programs that run on the operating system follow a consistent naming style.

The first three letters of the command name (known as the command verb) 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.

Related tasks

Defining CL commands

CL commands enable you to request a broad range of functions. You can use IBM-supplied commands, change the default values for command parameters, and define your own commands.

CL command verb abbreviations
The majority of all CL commands use one of these 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 table lists all the abbreviations that are used as command verbs.

Verb abbreviation	Meaning
ADD	add
ALC	allocate
ANS	answer
ANZ	analyze
APY	apply
ASK	ask
CFG	configure
CHG	change
CHK	check
CLO	close
CLR	clear
CMP	compare
CNL	cancel
CPR	compress
CPY	copy
CRT	create
CVT	convert
DCP	decompress
DLC	deallocate
DLT	delete
DLY	delay
DMP	dump

Verb abbreviation	Meaning
DSC	disconnect
DSP	display
DUP	duplicate
EDT	edit
EJT	eject
EML	emulate
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
PKG	package
POS	position
PRT	print
PWR	power
QRY	query
RCL	reclaim
RCV	receive
RGZ	reorganize
RLS	release
RMV	remove

Verb abbreviation	Meaning
RNM	rename
RPL	replace
RQS	request
RRT	reroute
RSM	resume
RST	restore
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
This table lists 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
ACP	accepted
ACNE	action entry
ACT	active, activity, activation
ADD	add
ADM	administration, administrative
ADP	adopt, adopting

Command abbreviation	Meaning
ADPI	adapter information
ADPP	adapter profile
ADPT	adapter
ADR	address
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
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

Command abbreviation	Meaning
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
CAD	cluster administrative domain
CAL	calendar
CALL	call
CAP	capture
CBL	COBOL language
CCF	credentials cache file
CCS	change control server
CDE	code, coded
CDS	coded data store
CFG	configuration, configure
CFGL	configuration list
CFGLE	configuration list entry
CGY	category
CHG	change
CHK	check
CHT	chart
CICS®	customer information control system
CKM	cryptographic key management
CL	control language
CLD	C locale description
CLG	catalog

Command abbreviation	Meaning
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	commitment
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

Command abbreviation	Meaning
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
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
DFR	defer, deferred
DFT	default
DFU	data file utility
DHCP	dynamic host configuration protocol
DIG	domain information groper
DIR	directory
DIRE	directory entry
DIRSHD	directory shadow
DKT	diskette
DKTF	diskette file
DL	DataLink, data library
DLC	deallocate

Command abbreviation	Meaning
DLF	DataLink file
DLFM	DataLink file manager
DLO	document library object
DLT	delete
DLY	delay
DMN	domain
DMP	dump
DNS	domain name service, Domain Name System
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
DW	Disk Watcher
DWN	down
E (suffix)	entry
EDT	edit
EDTD	edit description
EDU	education
EJT	eject
EML	emulate, emulation
ENC	encipher
END	end
ENR	enrollment

Command abbreviation	Meaning
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, expired, 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
FLT	filter
FMT	format
FMW	firmware
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
FTP	file transfer protocol
FTR	filter
GDF	graphics data format

Command abbreviation	Meaning
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
HOST	host
HST	history, historical
HT	host table
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
IGN	ignored
IMG	image
IMP	import
INF	information
INP	input
INS	install
INST	instance
INT	internal, integrated
INTR	intrasystem
INZ	initialize

Command abbreviation	Meaning
IPL	initial program load
IPS	internet protocol over SNA
ISDB	interactive source debugger
ISDN	integrated services digital network
ITF	interactive terminal facility
ITG	integrity
ITM	item
IWS	intelligent workstation
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
JW	Job Watcher
KBD	keyboard
KEY	key
KRB	Kerberos
KSF	keystore file
KTE	key tab entry
KVV	key verification value
L (suffix)	list
LAN	local area network
LANG	language
LBL	label
LCK	lock
LCL	local
LCLA	local attributes
LDIF	LDAP data interchange format
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
LNX	Linux®
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
MEM	memory
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

Command abbreviation	Meaning
MON	monitor
MOV	move
MRE	monitored resource entry
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
NET	network
NETF	network file
NFS	network file system
NODGRP	node group
NODL	node list
NTB	netbios
NTS	Notes®
NWI	network interface
NWS	network server
NWSAPP	network server application
NWSCFG	network server configuration
NWSD	network server description
NWSH	network server host adapter
OBJ	object
OCL	operation control language
OF	optical file
OFC	office
OFF	off
OMC	object management cycle
OND	OnDemand
OPC	OptiConnect
OPN	open
OPT	optical, options

Command abbreviation	Meaning
ORD	order
OSPF	open shortest path first
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	personal computer document
PCL	protocol
PCO	personal computer 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

Command abbreviation	Meaning
PIN	personal identification number
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, processing
PRD	product
PRF	profile
PRFL	profile list
PRI	primary
PRJ	project
PRM	promote
PRP	preparation
PRS	personal
PRT	print
PRTF	printer file
PRTQ	print queue
PRX	proxy
PSFCFG	print services facility configuration
PTC	portable transaction computer
PTF	program temporary fix
PTP	point-to-point
PTR	pointer
PVD	provider
PVT	private
PWD	password
PWR	power

Command abbreviation	Meaning
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
RIP	routing information protocol
RJE	remote job entry
RLS	release
RLU	report layout utility
RMC	report management cycle
RMT	remote
RMV	remove
RNDC	remote name daemon control
RNG	range
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

Command abbreviation	Meaning
RPYL	reply list
RQS	request
RRT	reroute
RSC	resource
RSE	Remote Systems Explorer
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

Command abbreviation	Meaning
SHRPOOL	shared pool
SIGN	sign
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
SUBR	subroutine
SVR	server
SWA	save while active

Command abbreviation	Meaning
SWL	stop word list
SYNC	synchronization
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
TIE	technical information exchange
TIEF	technical information exchange file
TIMZON	time zone
TKT	ticket
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
UDFS	user-defined file system
UPD	update

Command abbreviation	Meaning
UPG	upgrade
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 workstation
WAIT	wait
WCH	watch
WLS	wireless
WNT	Windows NT
WP	word processing
WRK	work with
WSE	workstation entry
WSG	workstation gateway
WSO	workstation object
WTR	writer
XREF	cross-reference
X25	X.25
ZNE	zone

CL command keyword abbreviations
Each command parameter has a keyword name associated with it. The keyword name can be up to 10 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 table contains 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
ADMDMN	administrative domain
ADP	adopt, adaptive
ADPT	adapter
ADR (or ADDR)	address
ADRLST	address list
ADV	advance
AFN	affinity
AGT	agent
AIX®	AIX operating system
AJE	autostart job entry
AFN	affinity
AFP	advanced function printing
ALC	allocate
ALM	alarm

Keyword Abbreviation	Meaning
ALR	alert
ALRD	alert description
ALS	alias
ALW	allow
ALWFWD	allow forwarding
ALWPRX	allow proxy
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
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, authentication
AUTH	authentication
AUTL	authorization list

Keyword Abbreviation	Meaning
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
BUF	buffer
C	C language
CAB	cabinet
CAL	calendar
CAP	capacity, capture
CB	callback
CBL	COBOL language
CCD	call control data

Keyword Abbreviation	Meaning
CCSID	coded character set identifier
CCT	circuit
CCF	credentials cache file
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
CHAP	challenge-handshake authentication protocol
CHG	change
CHK	check
CHKSUM	checksum
CHKVOL	check volume identifier
CHL	channel
CHR (or CHAR)	character
CHRSTR	character string
CHT	chart
CGY	category
CIM	common information model
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

Keyword Abbreviation	Meaning
CMNE	communications entry
CMP	compare
CMT	commit, 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
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

Keyword Abbreviation	Meaning
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
CTLP	control port
CTN	contention
CTS	clear to send
CTX	context
CUR	current
CURPWD	current password
CVG	coverage
CVN	conversion
CVR	cover
CVT	convert, converting
CXT (or CTX)	context
CYC	cycle
D (suffix)	description
DAP	directory access protocol
DAT	date
DB	database

Keyword Abbreviation	Meaning
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
DMD	demand
DMN	domain

Keyword Abbreviation	Meaning
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
DSP	display
DST	daylight savings time, dedicated service tools, distribution
DTA	data
DTE	data terminal equipment
DTL	detail
DUP	duplicate
DVL	development
DWN	down
DYN	dynamic
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, entries
ENV	environment

Keyword Abbreviation	Meaning
EOF	end of file
EOR	end of record
EOV	end of volume
ERR	error
ESC	escape
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
EXPTIME	expired time
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
FID	file identifier
FIL	file
FLD	field
FLG	flag
FLIB	files library
FLR	folder
FLW	flow

Keyword Abbreviation	Meaning
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
GOVR	governor
GPH	graph
GRP	group
GRT	grant
GSS	graphics symbol set
GVUP	give up
HA	high availability
HCP	host command processor
HDL (or HNDL)	handle
HDR	header

Keyword Abbreviation	Meaning
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
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, identification
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
IMMED	immediate
IN	input
INAC (or INACT)	inactivity
INACT (or INAC)	inactivity
INC	include
INCR	incremental

Keyword Abbreviation	Meaning
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
IP6	Internet protocol version 6, IPv6
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
JDFT	join default
JE (suffix)	job entry
JFLD	join field
JORDER	join file order
JRN	journal

Keyword Abbreviation	Meaning
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

Keyword Abbreviation	Meaning
LRC	longitudinal redundancy check
LRG	large
LRSP	local response
LSE	lease
LST	list, last
LTR	letter
LUW	logical unit of work
LVL	level
LWS	local workstation
LZYWRT	lazy write
M (suffix)	member, messages
MAC	macro, medium access control
MAINT	maintenance
MAJ	major
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

Keyword Abbreviation	Meaning
MNT	maintenance, mount, mounted
MNU	menu
MOD	mode, module
MODD	mode description
MODLVL	modification level
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
NBC	non-broadcast
NBR	number
NCK	nickname
NDE	node
NDM	normal disconnect mode
NDS	NetWare directory services
NEG	negative, negotiation
NEP	never-ending program
NET	network
NFY	notify

Keyword Abbreviation	Meaning
NGH	neighbor
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

Keyword Abbreviation	Meaning
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
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

Keyword Abbreviation	Meaning
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
PRB	problem
PRC	procedure, procedural, process, processed
PRD	product, productive
PREBLT	prebuilt
PRED	predecessor
PREEST	pre-established

Keyword Abbreviation	Meaning
PREF	preference, preferred
PREOPR	pre-operation
PREREQ	prerequisite
PRF	profile, profiling
PRI	primary
PRJ	project
PRL	parallel
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

Keyword Abbreviation	Meaning
PVY	privacy
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
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
RECOMMEND	recommendations
REF	reference
REINZ	reinitialize
REL	relations, release
REP	representation, representative
REQ (or RQS)	required, request, requester
RES	resident, resolution

Keyword Abbreviation	Meaning
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
RQD	required
RQS (or REQ)	request, requester
RQT	requisite
RRSP	remote response
RRT	reroute
RSB	reassembly
RSC	resource, resources
RSL	result, resolution
RSM	resume
RSP	response
RSRC	resource
RST	restore
RSTD	restricted
RTE	route
RTG	routing
RTL	retail

Keyword Abbreviation	Meaning
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

Keyword Abbreviation	Meaning
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
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
SP	service processor
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

Keyword Abbreviation	Meaning
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, static
STX	start-of-text character
SUB	substitution, subject
SUBADR	subaddress
SUBALC	suballocation
SUBDIR	subdirectory
SUBFLR	subfolder
SUBNET (or SUBN)	subnetwork
SUBPGM	subprogram
SUBR	subroutine
SUBST	substitution
SUCC	successor

Keyword Abbreviation	Meaning
SUN	Sunday
SUP	suppress, suppression
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, synchronization
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
THRESH (or THLD)	threshold
THRPUT	throughput
THS	thesaurus
THU	Thursday
TIE	technical information exchange

Keyword Abbreviation	Meaning
TIM	time
TIMMRK	timemark
TIMO	timeout
TIMOUT (or TIMO)	timeout
TIMZON	time zone
TKN	token
TKV	takeover
TL	transport-layer
TM	time
TMN	transmission
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

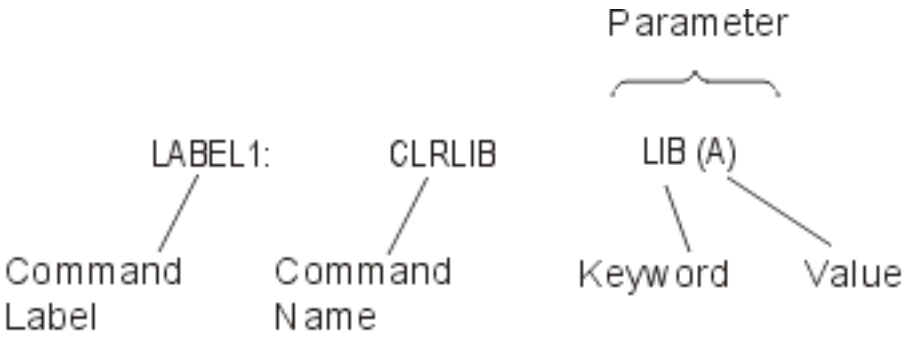
Keyword Abbreviation	Meaning
TXP	transport
TXT	text
TYP	type
T1	transport class 1
T2	transport class 2
T4	transport class 4
UDFS	user-defined file system
UDP	user datagram protocol
UI	user interface, unnumbered information
UID	user identifier number
UNC	unclassified
UNL	unlink
UNPRT	unprintable
UNQ	unique
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
VFYPWD	verify password
VLD	valid, validity, validation
VND	vendor
VOL	volume
VRF	verification
VRT	virtual
VRY	vary
VSN (or VER)	version
VWS	virtual workstation
WAN	wide area network

Keyword Abbreviation	Meaning
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

CL command parts

The parts of a command include a command label (optional), a command name (mnemonic), and one or more parameters. The parameter includes a keyword and a value.

This figure illustrates the parts of a command:



CL command syntax

A command name and parameters make up each CL command.

The parameters used in CL commands are keyword parameters. The keyword, typically 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).

Commands have the following general syntax. The brackets indicate that the item within them is optional; however, the parameter set may or may not be optional, depending upon the requirements of the command.

```
[//] [?] [label-name:][library-name/]command-name
[parameter-set]
```

Note: The // is valid only for a few batch job control commands, such as the DATA command. The // identifies those types of commands sent to the spooling reader that reads the batch job input stream.

CL command label

Command labels identify particular commands for branching purposes in a CL program. Labels can also be used to identify statements in CL programs that are being debugged. They can identify statements used either as breakpoints or as starting and ending statements for tracing purposes.

The label is typed just before the command name. The standard rules for specifying simple names (*SNAME) apply. The label is immediately followed by a colon. Blanks are allowed, though not required, between the colon and the command name. The label can contain as many as 10 characters, in addition to the colon. START: and TESTLOOP: are examples of command labels.

Command labels are not required, but a label can be placed on any command. For labels that are placed on commands that cannot be run (for example, the **Declare CL Variable (DCL)** command), when the program branches to that label, the next command following the label is run. If the command following the label cannot be run, the program will move to the next command that can be run. Similarly, you can specify only one label on a line. If a command is not located on that line, the program will jump to the next command that can be run.

To specify multiple labels, each additional label must be on a separate line preceding the command as shown:

```
LABEL1:
LABEL2:  CMDX
```

No continuation character (+ or -) is allowed on the preceding label lines.

Related concepts

Additional rules for unique names
Additional rules involve special characters (as an extra character) for object naming.

Naming within commands
The type of name you specify in control language (CL) determines the characters you can use to specify a name.

Communication names (*CNAME)
A communications name has a restrictive name syntax. It is typically used to refer to a device configuration object whose name length and valid character set is limited by one or more communication architectures.

Names (*NAME)
When you create basic names and basic names in quoted form, follow these rules.

Path names (*PNAME)
A *path name* is a character string that can be used to locate objects in the integrated file system.

Simple names (*SNAME)

Simple names are used for control language (CL) variables, labels, and keywords to simplify the syntax of CL. Simple names are the same as unquoted basic names but with one exception: periods (.) cannot be used.

CL command parameters

Most CL commands have one or more *parameters* that specify the objects and values used to run the commands.

When a command is entered, the user supplies the command object name, the parameter keyword names, and the parameter values used by the command. The number of parameters specified depends upon the command. Some commands (like the **DO (Do)** command and the **ENDBCHJOB (End Batch Job)** command) have no parameters, and others have one or more.

In this topic, the word *parameter* typically refers to the combination of the parameter keyword and its value. For example, the **Move Object (MOVOBJ)** command has a parameter called OBJ that requires an object name to be specified. OBJ is the parameter keyword, and the name of the object is the value entered for the OBJ parameter.

Related reference

List of values
A *list of values* is one or more values that can be specified for a parameter.

Required, optional, and key parameters
A CL command can have parameters that must be coded (required parameters) and parameters that do not have to be coded (optional parameters).

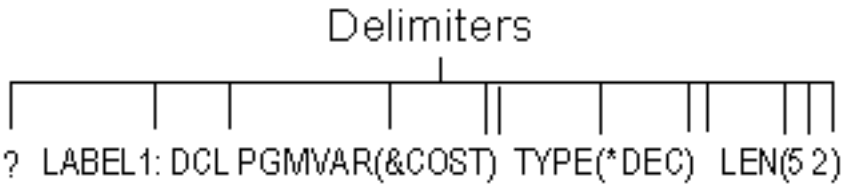
Parameters in keyword and positional form
You can specify parameters in CL using keyword form, positional form, or in a combination of the two.

Parameter values
A *parameter value* is user-supplied information that is used during the running of a command.

CL command delimiter characters

Command delimiter characters are special characters or spaces that identify the beginning or end of a group of characters in a command.

Delimiter characters are used to separate a character string into the individual parts that form a command: command label, command name, parameter keywords, and parameter values. Parameter values can be constants, variable names, lists, or expressions. The following figure shows various delimiter characters for a command.



The following delimiter characters are used in the IBM i control language:

- The colon (:) ends the command label, and it can be used to separate the command label from the command name.
- Blank spaces separate the command name from its parameters and separate parameters from each other. They also separate values in a list. Multiple blanks are treated as a single blank except in a quoted character string or comment enclosed in single quotation marks. A blank cannot separate a keyword and the left parenthesis for the value.
- Parentheses ( ) are used to separate parameter values from their keywords, to group lists of values, and to group lists within lists.
- Slashes (/) connect the parts of a qualified name or the parts of a path name.

  - For a qualified object name, the two parts are the library qualifier and the object name (LIBX/OBJA).
  - For a path name, the parts are the directory or directories searched and the object name ('/Dir1/Dir2/Dir3/ObjA').
- Either a period or a comma can be used as a decimal point in a decimal value (3.14 or 3,14). Only one decimal point is allowed in a value.
- Single quotation marks specify the beginning and ending of a quoted character string, which is a combination of any of the 256 extended binary-coded decimal interchange code (EBCDIC) characters that can be used as a constant. For example, 'YOU CAN USE $99@123.45 ()*></ and lowercase letters' is a valid quoted string that is a constant.

  Because a single quotation mark inside a quoted string is paired with the opening single quotation mark (delimiter) and is interpreted as the ending delimiter, a single quotation mark inside a quoted string must be specified as two single quotation marks. A pair of adjacent single quotation marks used this way is counted as a single character.
- A special character is used to separate a date into three parts: month, day, and year (two parts for Julian dates: year and day). The special characters that may be used as date separators are the slash (/), the hyphen (-), the period (.), a blank ( ), and the comma (,). The special character used to code as separators in a command date must be the same as the special character specified as the date separator for the job.
- The characters /* and */ can indicate the beginning and ending of a comment, or can be used in a character string. To begin a comment, the characters /* must begin in the first position of the command, be preceded by a blank, or be followed by either a blank or an asterisk. If the characters /* or */ occur in a later position of a command, they will typically be enclosed in single quotation marks and can represent, for example, all objects in the current directory for a path name.
- A question mark (?) preceding the command name indicates that the command is prompted. If the command is specified with a label, the question mark may either precede the label, or it may follow the label and precede the command name.

  Within a CL program, when a question mark precedes a command name, a prompt display is presented. You can enter parameter values not specified on the command in the program.

  Prompting characters may be put into a command in two forms. A single question mark (?) may be coded before the command name (either before or after the command label in a CL program) to cause the entire command to be prompted. Selective prompt characters (?? or ?*) may be coded before any parameter keyword to cause that parameter to be prompted when the command is run.

  If a question mark is entered before the command name on the command entry display, the effect is the same as pressing the F4 (Prompt) key after the command is entered.

  Within a CL program, when a question mark precedes the command name, a prompt display is presented. This display is of the same format as that presented when pressing the F4 key from the command entry display. Parameters of the command for which the program has coded values are shown for informational purposes, but the user cannot change the values supplied by the program. Parameters for which no value was coded are shown as input fields so you can enter values to be used in processing the command.

  Selective prompting allows you to identify specific command parameters to be prompted. To call selective prompting, the characters ??, ?*, or ?- are coded immediately preceding the keyword name of the parameter(s) to be prompted.

  Notes:

  1. Selective prompting is not allowed with command string (*CMDSTR) parameters.
  2. Parameters of the command that are preceded by the characters ?* are shown, but you cannot change the values that are supplied by the program. Parameters preceded by the characters ?? are shown as input fields containing the values coded in the program or command defaults so you can enter or change the values used in processing the command. Parameters preceded by the characters ?- are omitted from the display. All selectively prompted parameters must be coded in keyword or keyword-with-value form. Several parameters may be selectively prompted within one

command. When selective prompting is called, only keywords that are immediately preceded by the selective prompt characters are prompted. All other parameters are processed using the values as coded on the command or, if not coded, using command defaults.

Either form of prompting, but not both, is allowed on a single command in a CL program. If the character ? precedes the command name and selective prompt characters (except ?-) precede any keyword, an error message is returned and the program is not created.

Related concepts

Simple and qualified object names
The name of a specific object that is located in a library can be specified as a simple name or as a qualified name.

CL command coding rules
This summary of general information about command coding rules can help you properly code CL commands.

CL command definition
Command definition allows you to create additional control language commands to meet specific application needs. These commands are similar to the system commands.

Related tasks

Specifying prompt control for a CL command parameter
You can control which parameters are displayed for a command during prompting by using prompt control specifications.

Related reference

CL command definition statements
Command definition statements allows system users to create additional CL commands to meet specific application needs.

Related information

Integrated file system

CL command continuation

Commands can be entered in free format. This means that a command does not have to begin in a specific location on a coding sheet or on the display. A command can be contained entirely in one record, or it can be continued on several lines or in several records.

Whether continued or not, the total command length cannot exceed 32 702 characters. Either of two special characters, the plus sign (+) or the minus sign (-), is entered as the last nonblank character on the line to indicate that a command is continued. Blanks immediately *preceding* a + or - sign are always included; blanks immediately following a + or - in the *same record* are ignored. Blanks in the *next record* that precede the first non-blank character in the record are ignored when + is specified but are included when - is specified.

The + is generally useful between parameters or values. At least one blank must precede the sign when it is used between separate parameters or values. The difference between the plus and minus sign usage is particularly important when continuation occurs inside a quoted character string.

The example that follows shows the difference.

```
CRTLIB LIB(XYZ) TEXT('This is CONT+
   INUED')

CRTLIB LIB(XYZ) TEXT('This is CONT-
   INUED')

For + : CRTLIB LIB(XYZ) TEXT('This is CONTINUED')

For - : CRTLIB LIB(XYZ) TEXT('This is CONT   INUED')
```

Notes:

1. The minus sign causes the leading blanks on the next line to be entered.

2. Use continuation characters + and - in CL programs only. An error occurs if + or - is used on a command entry display.
3. The characters + and - are used for multiple-command examples, but not for single-command examples.

CL command comments

Comments within CL programs describe the expected behavior of the code.

Comments can be inserted either inside or outside a command's character string wherever a blank is permitted. However, because a continuation character must be the last non-blank character of a line (or record), comments may not follow a continuation character on the same line.

For readability, it is recommended that each comment be specified on a separate line preceding or following the command it describes, as shown here:

```
MOVOBJ   OBJA  TOLIB(LIBY)
        /* Object OBJA is moved to library LIBY. */
DLTLIB  LIBX
        /* Library LIBX is deleted. */
```

Comments can include any of the 256 EBCDIC characters. However, the character combination */ should not appear within a comment because these characters end the comment. To begin a comment, the characters /* must be placed in the first position of the command, be preceded by a blank, or be followed by either a blank or an asterisk.

CL command definition

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

Related concepts

CL command delimiter characters
Command delimiter characters are special characters or spaces that identify the beginning or end of a group of characters in a command.

Simple and qualified object names
The name of a specific object that is located in a library can be specified as a simple name or as a qualified name.

CL command coding rules
This summary of general information about command coding rules can help you properly code CL commands.

Related tasks

Specifying prompt control for a CL command parameter
You can control which parameters are displayed for a command during prompting by using prompt control specifications.

Related reference

CL command definition statements
Command definition statements allows system users to create additional CL commands to meet specific application needs.

Related information

Integrated file system

CL command coding rules

This summary of general information about command coding rules can help you properly code CL commands.

CL command coding rules for delimiter characters

- Blanks are the basic separators between the parts of a command:
  - Between command label and command name (not required, because the colon [:] is the delimiter).
  - Between command name and first parameter, and between parameters.
  - Between values in a list of values (not required between ending and beginning parentheses that enclose nested lists inside a list).
  - Between the slashes and the name or label of some job control commands, like // DATA (not required).
- Blanks cannot separate a parameter's keyword from the left parenthesis preceding its values. When a keyword is used, parentheses must be used to enclose the values; blanks can occur between the parentheses and the values. For example, KWD(A) is valid.
- Multiple blanks are treated as a single blank, unless they occur in a quoted string or a comment.
- A colon must immediately follow a command label. Only one label can be used on any command (LABEL1: DCLF).
- Single quotation marks must be used to specify the beginning and end of a quoted character string. If a character string contains special characters, such as blanks, single quotation marks are required. If a single quotation mark must be used in the quoted string, two single quotation marks must be entered consecutively to indicate that it is a single quotation mark and not the end of the quoted string.
- Parentheses must be used:
  - On parameters that are specified (coded) in keyword form
  - To group multiple values in a single list, in a positional parameter, or around expressions
  - To indicate a list (of none, one, or several elements) nested inside another list
- Sets of parentheses inside parentheses can be entered as long as they are paired, up to the maximum of five nested levels in logical expressions or three nested levels in lists of values.
- Comments can appear wherever blanks are permitted, except after a continuation character on the same line or record.
- A plus or minus sign at the end of a line indicates that the command is continued on the following line. Blanks following a + or - sign in the same record are ignored; blanks in the next record that precede the first nonblank character are ignored when + is specified and included when - is specified. One blank must precede the + sign when it is used between separate parameters or values.

CL command coding rules for parameters

- All required parameters must be coded.
- If an optional parameter is not coded, the system uses its default value if the parameter has one. A default value is indicated by showing the value as bold and underlined text in the Choices column of the parameter summary table.
- Words or abbreviations specified in capital letters in the command and parameter descriptions must be coded as shown. This is true of all command names, the keywords of parameters (if used), and many parameter values. Lowercase letters not coded in quoted strings or in comments are translated to uppercase letters. Lowercase letters specified for values on parameters defined as CASE(*MIXED) are not translated to uppercase letters.
- If there are key parameters, the values for the key parameters must be entered on the prompt before the remaining parameters will be shown. The Notes column in the parameter summary table indicates which parameters, if any, are key parameters.

- Parameters cannot be coded in positional form past the positional parameter limit defined in the command object. The Notes column in the parameter summary table indicates which parameters may be specified in positional form.

CL command coding rules for parameter values

- The first character in all names must be an alphabetic character (A through Z, $, #, @, or a double quotation mark (")). Names must not exceed 10 characters (CL variable names and built-in function names can have 11 characters maximum, including the preceding & or % characters). In some commands, the names of objects can be specified in qualified form (library-name/object-name) or in path name form (directory-name/object name).
- Predefined values that begin with an asterisk can be used only for the purposes intended, unless they are included in comments or quoted strings. These include predefined parameter values (*ALL, for example), symbolic operators (*EQ, for example), and the null value (*N).
- In a CL program, a variable can be specified for all parameters, except where that is explicitly restricted. The user-specified value of the variable is passed as if it had been specified on the command.
- In a CL program, a character string expression can be specified for any parameter defined with EXPR(*YES). The resulting value of the expression is passed as if the value had been specified on the command.
- Null (omitted) values are specified with the characters *N, which mean that no value was specified and the default value, if one exists, should be used. *N is needed only when another value following the omitted value is being specified as a positional parameter or an element in a list.
- Either a comma or a period can be used to indicate a decimal point in a numeric value. The decimal point is the only special character allowed between digits in the numeric string; there is no delimiter for indicating thousands.
- When repetition is indicated for a parameter:
  - A predefined value is not coded more than once in a series of values.
  - As many user-defined values (like names or numeric limits) can be entered as there are different values or names, up to the maximum number of repetitions allowed. For example, if a parameter description states "Up to 20 repetitions," you can specify a maximum of 20 values.

Note: When you are using parameters that have the same name in different commands, the meaning of (and the values for) that parameter in each command may be somewhat different. Refer to the correct command description for the explanation of the parameter you are using.

Related concepts

CL command delimiter characters
Command delimiter characters are special characters or spaces that identify the beginning or end of a group of characters in a command.

CL command definition
Command definition allows you to create additional control language commands to meet specific application needs. These commands are similar to the system commands.

Related reference

CL command definition statements
Command definition statements allows system users to create additional CL commands to meet specific application needs.

Commonly used parameters
Some parameters are commonly used in many CL commands.

Parameters in keyword and positional form
You can specify parameters in CL using keyword form, positional form, or in a combination of the two.

Parameter values

A *parameter value* is user-supplied information that is used during the running of a command.

CL command information and documentation

IBM provides documentation for CL commands. In addition, you can write documentation for your own commands.

Related tasks

Documenting a CL command

If you define and create your own CL commands, you can also create online command help to describe your commands.

CL command documentation format

Command documentation for CL commands is provided in the form of online help on the system and topics in the IBM i Information Center. In the information center, each command description follows the same format.

Each command description includes the parts discussed in the following subtopics. At the beginning of the command description documentation, there are links to the Parameters, Examples, and Error messages sections.

It should be noted that, because a command is an IBM i object, each command can be authorized for specific users or authorized for use by the public (all users authorized in some way to use the system). Because this is true for nearly every command, it is not stated in each command description.

Related information

Security reference

CL environment classification

At the very top of the command description documentation is the environment classification. The environment classification describes where the command is allowed to run.

Where allowed to run indicates in which environments the command can be entered. This information is the same information that is shown in the output of the **Display Command (DSPCMD)** command, which reflects what was specified for the ALLOW parameter when the command definition object was created. The **Where allowed to run** value includes the symbolic special values specified for the ALLOW parameter and a brief description that explains the environments where the command is allowed to run.

The majority of commands are created with ALLOW(*ALL); *ALL is also the shipped default value for the ALLOW parameter. In this case, the description will be "All environments (*ALL)".

For commands that must be run interactively, the ALLOW values specified when the command was created are typically (*INTERACT *IPGM *IREXX *EXEC) or (*INTERACT *IPGM *IMOD *IREXX *EXEC). In these two cases, the description shown will be "Interactive environments (*INTERACT *IPGM *IREXX *EXEC)" or "Interactive environments (*INTERACT *IPGM *IMOD *IREXX *EXEC)".

For commands that are created to be run only in a compiled CL or interpreted REXX program, the ALLOW values specified when the command was created are typically (*BPGM *IPGM *BREXX *IREXX) or (*BPGM *IPGM *BMOD *IMOD *BREXX *IREXX). In these two cases, the description shown will be "Compiled CL program or interpreted REXX (*BPGM *IPGM *BREXX *IREXX)" or "Compiled CL or interpreted REXX (*BPGM *IPGM *BMOD *IMOD *BREXX *IREXX)".

If the combination of values specified for the ALLOW parameter when the command was created is not one of the previous combinations, a bulleted list is shown that gives a brief description of each value that was specified.

- Batch job (*BATCH)
- Interactive job (*INTERACT)
- Batch ILE CL module (*BMOD)
- Interactive ILE CL module (*IMOD)
- Batch program (*BPGM)

- Interactive program (*IPGM)
- Batch REXX procedure (*BREXX)
- Interactive REXX procedure (*IREXX)
- Using QCMDEXEC, QCAEXEC, or QCAPCMD API (*EXEC)

Note: Some command definition objects shipped as part of the IBM i operating system are not intended to be used as CL commands. For example, the CMD and PARM command definition objects are used in command definition source. These special-purpose command objects will not have any **Where allowed to run** information.

Related concepts

CL threadsafe classification

The threadsafe classification indicates whether a command is threadsafe. Each command has a threadsafe classification.

CL threadsafe classification

The threadsafe classification indicates whether a command is threadsafe. Each command has a threadsafe classification.

The three types of threadsafe classifications are as follows:

- Threadsafe: Yes

  This classification indicates that you can safely call the command simultaneously in multiple threads without restrictions. This classification also indicates that all functions called by this command are threadsafe.

- Threadsafe: Conditional

  This classification indicates that not all functions provided by the command are threadsafe. The Restrictions section of the command provides information relating to thread safety limitations. Many commands are classified conditionally threadsafe because either some underlying system support is not threadsafe or the command can cause an exit point to be called.

- Threadsafe: No

  This classification indicates that the command is not threadsafe and should not be used in a multithreaded program. While some thread unsafe commands may deny access, most thread unsafe commands do not. A diagnostic message, CPD000D, may be sent to the job log to indicate that a non-threadsafe command has been called. Whether the diagnostic message CPD000D is sent to the job log depends on the "multithreaded job action" attribute of the command; that attribute can be determined by using the **Display Command (DSPCMD)** command. The possible values and actions are:

  – *SYSVAL - Action is based on system value QMLTTHDACN
  – *RUN - Command runs. No messages are sent.
  – *MSG - Diagnostic message CPD000D is sent to the job log. The command runs.
  – *NORUN - Diagnostic message CPD000D is sent to the job log, and escape message CPF0001 is sent. The command does not run.

  If the command is run, the results are unpredictable.

Note: Some command definition objects shipped as part of the IBM i operating system are not intended to be used as CL commands. For example, the CMD and PARM command definition objects are used in command definition source. These special-purpose command objects will not have any Threadsafe information.

Related concepts

CL environment classification

At the very top of the command description documentation is the environment classification. The environment classification describes where the command is allowed to run.

CL command description
The general description briefly explains the function of the command and any relationships it has with a program or with other commands. If there are restrictions on the use of the command, they are described under the heading Restrictions. The general description of the command follows the environment and threadsafe classification.

CL parameters
The **Parameters** section provides a parameter summary table.

The parameter summary table shows all the parameters and values that are valid for the command. Possible values are indicated in the Choices column. The default value, as shipped by IBM, is underlined in the Choices column. The default values are used by the system for parameters or parts of parameters that are not coded.

Parameter descriptions

Parameter descriptions follow the parameter summary table. Parameter descriptions are presented in the same order as the parameters are listed in the parameter summary table. Each parameter description includes an explanation of the function of the parameter, followed by a description of each possible parameter value. The default parameter value, if there is one, is typically described first and is shown as an underlined heading at the beginning of the text that describes the value.

The description of each parameter explains what the parameter means, what it specifies, and the dependent relationships it has with other parameters in the command. When the parameter has more than one value, the information that applies to the parameter as a whole is covered first, then the specific information for each of the values is described after the name of each value.

Parameter summary table

The parameter summary table summarizes parameters and values for CL commands.

Keyword column

This column shows the *parameter keyword* name. Every CL command parameter has a keyword name associated with it. When you are viewing the command documentation using a browser, you can click on the keyword name to link to the start of the information for the parameter within the command documentation file.

Description column

This column shows the prompt text defined for the parameter, a parameter qualifier, or a parameter element. *Qualifiers* are normally used for qualified object names or qualified job names. *Elements* are used to define multiple input fields for a single parameter. The description for a qualifier or element contains the qualifier or element number within the parameter.

Choices column

This column shows the possible values for the parameter, qualifier, or element.

- *Predefined values*, also known as special values, are listed in this column. Predefined values typically begin with an asterisk (*) or Q, followed by all uppercase letters.
- If the parameter, qualifier, or element allows *user-defined values*, a description of the parameter type appears in italics, for example *Name*.
- *Default values* may be defined for optional parameters. Default values are shown in bold, underlined text, for example **<u>*NO</u>** .
- For complex parameters that have multiple qualifiers or elements, or if the parameter or element supports a list of values, any *single value* choices are identified. Single value choices may be used only once.

- *Repeated values* can be specified for some parameters. For repeated values, this column indicates the number of allowed repetitions.

Notes column

This column shows additional information about each parameter.

- "Required" appears in this column to indicate *required parameters*, parameters for which you are always required to specify an input value.
- "Optional" appears in this column to indicate *optional parameters*, parameters for which no input value is required.
- "Key" appears in this column to indicate *key parameters*, which are used by commands that have prompt override programs.
- "Positional" appears in this column if the parameter is allowed to be specified positionally (without the associated parameter keyword) in a command string. The parameter's positional number appears following "Positional".

CL command coding examples
The **Examples** section provides at least one coded example for the command. Where necessary, several examples are provided for commands with many parameters and several logical combinations.

For clarity, examples are coded in keyword form only. The same examples could be coded either in positional form or in a combination of keyword and positional forms, for commands that support one or more positional parameters.

CL error messages
The **Error messages** section lists error messages that can be issued for the command.

CL command help

All IBM CL commands have online help available.

Online help contains the parameter and value descriptions for the command. To display help for a command, do one of the following:

- From an IBM i command line, type the command name (for example, CRTUSRPRF) and press F1. The display shows general help for the command and help for each command parameter.
- From an IBM i command line, type the command name (for example, CRTUSRPRF) and press F4 to display the command prompt display. On the prompt display, move the cursor to the top line and press F1.

Printing CL command descriptions on the system

To print the parameter and value descriptions for a CL command, follow these instructions.

To print help for an entire command, perform either of the following steps:

- From any command line, type the command name (for example, CRTUSRPRF) and press F1. The display shows general help for the command and help for each command parameter. Press F14 to print the command help.
- On a prompt display for a given command, move the cursor to the top line and press F1. Then press F14.

To print the help for one CL command keyword parameter, perform the following steps:

- From a command line, type the CL command name and press F4 to display the command prompt display. Position the cursor anywhere on the line of the keyword parameter for which you want help. Press F1 to display the help for the keyword parameter. Press F14 to print the help.

Related reference

PDF file for Control language

You can view and print a PDF file of this information.

CL command prompters

Command prompters allow you to prompt for CL command parameters and values.

Command 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 single quotation marks 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.

System i® Navigator provides a graphical CL command prompter for use on a client PC. IBM System i Access for Web provides an HTML form-based CL command prompter for use in a Web browser. The Remote System Explorer (RSE) function, which is a function of the IBM Rational® Development Studio for i licensed program, also provides a graphical CL command prompter.

The IBM i operating system 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.

CL commands that operate on IBM i objects

Each of the IBM i object types has a set of commands that operates on that object type.

Most IBM i object types have commands that perform the following actions:

- Create (CRT): Creates the object and specifies its attributes.
- Delete (DLT): Deletes the object from the system.
- Change (CHG): Changes the attributes, contents of the object, or both.
- Display (DSP): Displays the contents of the object. Display commands cannot be used to operate on objects.
- Work with (WRK): Works with the attributes, contents of the object, or both. Unlike display commands, work commands allow users to operate on objects and modify applications.

Related reference

CL commands that operate on multiple objects

In addition to the commands that operate on single object types, there are commands that operate on several object types. These commands are more powerful because they can operate on several objects of different types at the same time.

CL commands that operate on multiple objects

In addition to the commands that operate on single object types, there are commands that operate on several object types. These commands are more powerful because they can operate on several objects of different types at the same time.

For example:

- Display object description (DSPOBJD or DSPLNK) displays the common attributes of an object.
- Save object (SAVOBJ or SAV) saves an object and its contents on tape, optical media, or in a save file.
- Restore object (RSTOBJ or RST) restores a saved version of the object from tape, optical media, or from a save file.

Some of the commands operate on only one object at a time, but that object can be any one of several IBM i object types. For example:

- Move object (MOVOBJ or MOV) moves an object from one library or directory to another.
- Rename object (RNMOBJ or RNM) specifies the new name of an object.

The following tables list commands that perform an action on many of the object types.

Table 1. Commands operating on multiple object types (where object identified by object name, library and type)		
Item	**Actions**	**Identifier**
Object	ALC, CHK, CPR, CRTDUP, DCP, DLC, DLT, DMP, MOV, RNM, RST, SAV, SAVCHG, WRK,	OBJ
Object access	SET	OBJACC
Object auditing	CHG	OBJAUD
Object authority	DSP, EDT, GRT, RVK	OBJAUT
Object description	CHG, DSP, RTV	OBJD
Object journaling	CHG, END, STR	JRNOBJ
Object lock	WRK	OBJLCK
Object owner	CHG, WRK	OBJOWN
Object primary group	CHG, WRK	OBJPGP

Table 2. Commands operating on multiple object types (where object identified by path name)		
Item	**Actions**	**Identifier**
Object	CPY, MOV, RNM, RST, SAV	not applicable
Object auditing	CHG	AUD
Object authority	CHG, DSP, WRK	AUT
Object description	DSP, WRK	LNK
Object integrity	CHK	OBJITG
Object journaling	END, STR	JRN
Object owner	CHG	OWN
Object primary group	CHG	PGP

Related concepts

CL commands that operate on IBM i objects
Each of the IBM i object types has a set of commands that operates on that object type.

Related reference

OBJTYPE parameter
The object type (OBJTYPE) parameter specifies the types of IBM i objects that can be operated on by the command in which they are specified.

CL programs and procedures

CL programs and procedures are created from source statements that consist entirely of CL commands.

A *CL source program* is the set of CL source statements that can be compiled into either an original program model (OPM) program or an Integrated Language Environment (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 CL 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.

There are four types of CL programs and procedures: procedure, module, program, and service program.

CL 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 typically begins with a PGM statement and ends with an ENDPGM statement.

CL module

A *module* is the object that results from compiling high-level language source statements using an Integrated Language Environment (ILE) 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.

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 high-level languages (HLLs) (for example, C), a single module can contain multiple user-written procedures.

CL program

A *control language (CL) program* is a program that is created from source statements consisting entirely of CL commands. The operating system supports two types of programs: ILE program and OPM CL program.

An *ILE program* is a program written in an Integrated Language Environment (ILE)-conforming high-level language. ILE programs are IBM i 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.

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.

Related information

ILE Concepts

Service program

A service program is an IBM i 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.

CL parameters

A *parameter* is a value that is passed to a command or program to provide user input or control the actions of the command or program.

Parameter values

A *parameter value* is user-supplied information that is used during the running of a command.

An individual value can be specified in a constant value, a variable name, an expression, or a list of values. A parameter can specify one or a group of such values, depending on the parameter's definition in a

command. If a group of values is specified, the parameter is called a *list parameter* because it can contain a list of values.

On commands with key and positional parameters, values can be specified in keyword form, positional form, or a combination of both forms. Parameter values must be enclosed in parentheses if any of the following conditions are true:

- A keyword precedes the value.
- The value is an expression.
- A list of values is specified.

  Note: If only one value is specified for a list, no parentheses are required.

Related concepts

CL command coding rules

This summary of general information about command coding rules can help you properly code CL commands.

Related reference

CL command parameters

Most CL commands have one or more *parameters* that specify the objects and values used to run the commands.

Constant values

A *constant value* is the actual value. The types of constants are character string (includes names, date and hexadecimal values), decimal, and logical.

A constant value is an actual numeric value or a specific character string whose value does not change. Three types of constants can be used by the control language: character string (quoted or unquoted), decimal, and logical.

Character strings

A *character string* is a string of any EBCDIC characters (alphanumeric and special) that are used as a value.

These EBCDIC values can include date and hexadecimal values. A character string can have two forms: quoted string or unquoted string. Either form of character string can contain as many as 5000 characters.

A *quoted* character string is a string of alphanumeric and special characters that are enclosed in single quotation marks. For example, 'Credit limit has been exceeded' is a quoted character string. The quoted string is used for character data that is not valid in an unquoted character string. For example, user-specified text can be entered in several commands to describe the functions of the commands. Those descriptions must be enclosed in single quotation marks if they contain more than one word because blanks are not allowed in an unquoted string.

An *unquoted* character string is a string consisting of only alphanumeric characters and the special characters that are shown in the *Unquoted String* column in the following table. The table summarizes the main EBCDIC characters that are valid in unquoted and quoted character string values. An X in the last column indicates that the character on the left is valid; refer to the specific notes following the figure that indicate why the character is valid as described. The special characters allow the following to be unquoted character string values:

- Predefined values (* at the beginning)
- Qualified object names (/)
- Generic names (* at the end)
- Decimal constants (+, -, ., and ,)

Any of these unquoted strings can be specified for parameters defined to accept character strings. In addition, some parameters are defined to accept predefined values, names, or decimal values either singly or in combinations.

Table 3. Quoted and unquoted character strings

Name of character	Character	Unquoted string	Quoted string
Ampersand	&	See Note 5	X
Single quotation mark	'	See Note 7	-
Asterisk (*)	*	See Notes 5, 6	X
At sign	@	X	X
Blank			X
Colon	:		X
Comma	,	See Note 1	X
Digits	0-9	See Note 1	X
Dollar sign	$	X	X
Equal	=	See Notes 5, 8	X
Greater than	>	See Notes 5, 8	X
Left parenthesis	(	See Note 4	X
Less than	<	See Notes 5, 8	X
Letters (lowercase)	a-z	See Note 2	X
Letters (uppercase)	A-Z	X	X
Minus	-	See Notes 1, 5	X
Not	¬	See Notes 5, 8	X
Number sign	#	X	X
Percent	%		X
Period	.	See Notes 1, 11	X
Plus	+	See Notes 1, 5	X
Question mark	?		X
Quotation marks	" "	See Note 10	X
Right parenthesis	)	See Note 4	X
Semicolon	;		X
Slash	/	See Notes 3, 5	X
Underscore	_	See Note 9	X
Vertical bar	|	See Notes 5, 8	X

Notes:

1. An unquoted string of all numeric characters, an optional single decimal point (. or ,), and an optional leading sign (+ or -) are valid unquoted strings. Depending on the parameter attributes in the command definition, this unquoted string is treated as either a numeric or character value. On the CALL command or in an expression, this unquoted string is treated as a numeric value; a quoted string is required if you want character representation. Numeric characters used in any combination with alphanumeric characters are also valid in an unquoted string.

2. In an unquoted string, lowercase letters are translated into uppercase letters unless the string is specified for a parameter that has the attribute CASE(*MIXED).
3. A slash can be used as a connector in qualified names and path names.
4. In an unquoted string, parentheses are valid when used to delimit keyword values and lists, or in expressions to indicate the order of evaluation.
5. In an unquoted string, the characters +, -, *, /, &, |, ¬, <, >, and = are valid by themselves. If they are specified on a parameter that is defined in the command definition with the EXPR(*NO) attribute, they are treated as character values. If they are specified on a parameter that is defined in the command definition with the EXPR(*YES) attribute, they are treated as expression operators.

   In CL Prompter, if you enter the character >, < or + in the first column of a parameter value, they have special meanings to Prompter:

   - > : inserts an entry in a parameter list
   - < : removes an entry from a parameter list
   - + : expands a parameter list to allow more values

   So, when prompt the command, if >, < or + are specified on a parameter that is defined in the command definition with the EXPR(*NO) attribute, they are treated as character values only if they are not in the first column of a parameter value.
6. In an unquoted string, the asterisk is valid when followed immediately by a name (such as in a predefined value) and when preceded immediately by a name (such as in a generic name).
7. Because a single quotation mark within a quoted string is paired with the opening single quotation mark (delimiter) and is interpreted as the ending delimiter, an adjacent pair of single quotation marks ('') must be used inside a quoted string to represent a single quotation mark that is not a delimiter. When characters are counted in a quoted string, a pair of adjacent single quotation marks is counted as a single character.
8. In an unquoted string, the characters <, >, =, ¬, and | are valid in some combinations with another character in the same set. Valid combinations are: <=, >=, ¬=, ¬>, ¬<, ||, |<, and |>. If the combination is specified on a parameter that is defined in the command definition with the EXPR(*NO) attribute, it is treated as a character value. If it is specified on a parameter that is defined in the command definition with the EXPR(*YES) attribute, it is treated as an expression operator.
9. In an unquoted string, the underscore is not valid as the first character or when used by itself.
10. Quotation marks are used to delimit a quoted name.
11. A period is valid in a basic name, except as the first character.

The following table contains examples of quoted string constants.

Table 4. Quoted string constants

Constant	**Value**
'1,2,'	1,2,
'DON''T'	DON'T
'24 12 20'	24 12 20

The following table contains examples of unquoted string constants.

Table 5. Unquoted string constants

Constant	**Meaning**
CHICAGO	CHICAGO
FILE1	FILE1
*LIBL	Library list

Table 5. Unquoted string constants (continued)	
Constant	**Meaning**
LIBX/PGMA	Program PGMA in library LIBX
1.2	1.2

Related reference

Date values
A *date value* is a character string that represents a date.

Hexadecimal values
A *hexadecimal value* is a constant that is made up of a combination of the hexadecimal digits A through F and 0 through 9.

Expressions
An *expression* is a group of constants, variables, or built-in functions, separated by operators, that produces a single value.

Character string expressions
The operands in a character string expression must be quoted or unquoted character strings, character CL variables, the substring (%SUBSTRING or %SST) built-in function, or a trim (%TRIM, %TRIML, or %TRIMR) built-in function, or a conversion (%CHAR, %LOWER, or %UPPER) built-in function.

Hexadecimal values
A *hexadecimal value* is a constant that is made up of a combination of the hexadecimal digits A through F and 0 through 9.

All character strings except names, dates, and times can be specified in hexadecimal form. To specify a hexadecimal value, the digits must be specified in multiples of two, be enclosed in single quotation marks, and be preceded by an X. Examples are: X'F6' and X'A3FE'.

Note: Care should be used when entering hexadecimal values in the range of 00 through 3F, or the value FF. If these characters are shown or printed, they may be treated as device control characters producing unpredictable results.

Related reference

Character strings
A *character string* is a string of any EBCDIC characters (alphanumeric and special) that are used as a value.

Date values
A *date value* is a character string that represents a date.

A date value's format is specified by the system value QDATFMT. The length of the date value varies with the format used and whether a separator character is used. For example, if no separator character is used, the length of a date in a Julian format is five characters, and the length of a date in a non-Julian format is six characters. If a separator character is used, the length will be greater.

The system value QDATSEP specifies the optional separator character that can be used when the date is entered. If a separator character is used, the date must be enclosed in single quotation marks.

A date value can be specified for the parameters of type *DATE. A year value equal to or greater than 40 indicates a year from 1940 through 1999. A year value less than 40 indicates a year from 2000 through 2039.

Related reference

Character strings
A *character string* is a string of any EBCDIC characters (alphanumeric and special) that are used as a value.

Related information

System values

Data separator (QDATSEP) system value

Decimal values
A *decimal value* is a numeric string of one or more digits, optionally preceded by a plus (+) or minus (-) sign.

A decimal value can contain a maximum of 15 digits, of which no more than nine can follow the decimal point (which can be either a comma or a period). Therefore, a decimal value can have no more than 17 character positions including the plus or minus sign and decimal point (if any). The following are examples of decimal values.

Logical values
A *logical value* is a single character (1 or 0) that is enclosed in single quotation marks.

Logical values are often used as a switch to represent a condition such as on or off, yes or no, and true or false. When used in expressions, a logical value can be optionally preceded by *NOT or ¬. The following are examples of logical values:

Table 6. Logical values

Constant	Value	Meaning
'0'	0	Off, no, or false
'1'	1	On, yes, or true

Floating-point constants
A *floating-point constant* is a representation of a numerical constant, shown as an optional sign followed by one or more digits and a decimal point, which may be at the end.

This representation consists of:

- A significand sign: The significand sign may be + or -. The significand sign is optional; it is assumed to be + if no sign is specified.
- A significand: The significand must contain a decimal point. The maximum number of digits that can be specified for the significand is 253; however, only the first 17 significant digits are used.
- An exponent character: The exponent character must be E.
- An exponent sign: The exponent sign must be + or -. The significand sign is optional; it is assumed to be + if no sign is specified.
- An exponent: The exponent must be an integer; numbers 0 through 9 are valid. The maximum number of digits that can be specified is three.

All floating-point constants are stored as double-precision values. No blanks are allowed between any of the parts of a floating-point constant, and the parts must be in the order listed previously.

Some commands have parameters for which floating-point constants can be specified:

- **Call Program (CALL)** or **Call Procedure (CALLPRC)** command: You can use the PARM parameter to pass a floating-point constant to a called program. Any program you call must receive a floating-point constant as a double precision value.
- **Change Program Variable (CHGPGMVAR)** command: You can use the VALUE parameter to change a floating-point variable in a program.

- Copy File (CPYF) command: You can use floating-point construction in the FROMKEY, TOKEY, and INCREL parameters to select which records are copied from a database file.

Related information

DDS

Variable name

A variable contains a data value that can be changed when a program is run. A *variable name* is the name of the variable that contains the value.

The variable is used in a command to pass the value that it contains at the time the command is run. The change in value can result if one of the following conditions occur: the value is received from a data area, a display device file field, or a message; the value is passed as a parameter; a **Change Variable (CHGVAR)** command is run in the program; or another program that is called changes the value before returning it.

The types of variables are character string (includes names), decimal, logical, and integer. Decimal and logical values must match the type of value expected for the parameter. Character variables can specify any type of value. For example, if a decimal value is expected, it can be specified by a character variable as well as by a decimal variable.

The variable name identifies a value to be used; the name points to where the actual data value is. Because CL variables are valid only in CL programs, they are often called *CL program variables* or CL variables. CL variable names must begin with an ampersand (&).

CL variables can be used to specify values for almost all parameters of CL commands. When a CL variable is specified as a parameter value and the command containing it is run, the current value of the variable is used as the parameter value. That is, the variable value is passed as if the user had specified the value as a constant.

Because it is generally true that CL variables can be used for most parameters of commands in CL programs, the command descriptions typically do not mention CL variables. For parameters that are restricted to constants only (such as in the DCL command), to CL variables only (such as all of the parameters of the **Retrieve Job Attributes (RTVJOBA)** command), or to specific types of variables (such as on the RTVJOBA or **Retrieve Message (RTVMSG)** command), the individual parameter descriptions specify those limitations. Otherwise, if the command is allowed in a CL program, CL variables can be used in place of a value, even with parameters that accept only predefined values. For example, a KEEP parameter having only predefined values of *YES and *NO can have a CL variable specified instead; its value can then be changed to *YES or *NO, depending on its value when the command is run.

A CL variable must contain only one value; it may not contain a list of values separated by blanks.

The value of any CL program variable can be defined as one of the following types:

- Character: A character string that can contain a maximum of 9999 characters. The character string can be coded in quoted or unquoted form, but only the characters in the string itself are stored in the variable.
- Decimal: A packed decimal value that can contain a maximum of 15 digits, of which no more than nine can be decimal positions.
- Logical: A logical value of '1' or '0' that represents on/off, true/false, or yes/no.
- Integer: A two-byte or four-byte binary integer value that can be either signed (value may be positive or negative) or unsigned (value is always positive).

Table 7. CL program variables

If value is:	**CL variable can be declared as:**
Name	Character
Date or time	Character

Table 7. CL program variables (continued)	
If value is:	**CL variable can be declared as:**
Character string	Character
Numeric	Decimal or integer or character
Logical	Logical or character

Related concepts

Additional rules for unique names
Additional rules involve special characters (as an extra character) for object naming.

Expressions

An *expression* is a group of constants, variables, or built-in functions, separated by operators, that produces a single value.

The operators specify how the values are combined to produce the single value or result. The types of expressions are arithmetic, character string, relational, and logical. An expression can be used as a value for a CL command parameter only in CL source programs.

The operators can be arithmetic, character string, relational, or logical. The constants or variables can be character, decimal, integer, or logical. For example, the expression (&A + 1) specifies that the result of adding 1 to the value in the variable &A is used in place of the expression.

Character string expressions can be used in certain command parameters defined with EXPR(*YES) in CL programs. An expression can contain the built-in functions %BINARY (or %BIN), %CHAR, %CHECK, %CHECKR, %DEC, %INT, %LEN, %LOWER, %PARMS, %SCAN, %SIZE, %SUBSTRING (or %SST), %SWITCH, %TRIM, %TRIML, %TRIMR, %UINT(or %UNS) and %UPPER. The types of expressions and examples of each are described in “Expressions in CL commands” on page 115.

Related concepts

Symbolic operators
A variety of characters can be used as symbolic operators in CL commands.

Related reference

Character strings
A *character string* is a string of any EBCDIC characters (alphanumeric and special) that are used as a value.

Expressions in CL commands
A character string expression can be used for any parameter, element, or qualifier that is defined with EXPR(*YES) in the command definition object.

List of values

A *list of values* is one or more values that can be specified for a parameter.

Not all parameters can accept a list of values. A *list parameter* can be defined to accept a specific set of multiple values that can be of one or more types. Values in the list must be separated by one or more blanks. Each list of values is enclosed by parentheses, indicating that the list is treated as a single parameter. Parentheses are used even when a parameter is specified in positional form. To determine whether a list can be specified for a parameter, and what kind of list it can be, refer to the parameter description under the appropriate command description.

A list parameter can be defined to accept a list of multiple like values (a simple list) or a list of multiple unlike values (a mixed list). Each value in either kind of list is called a *list element*. List elements can be constants, variables, or other lists; expressions are not allowed.

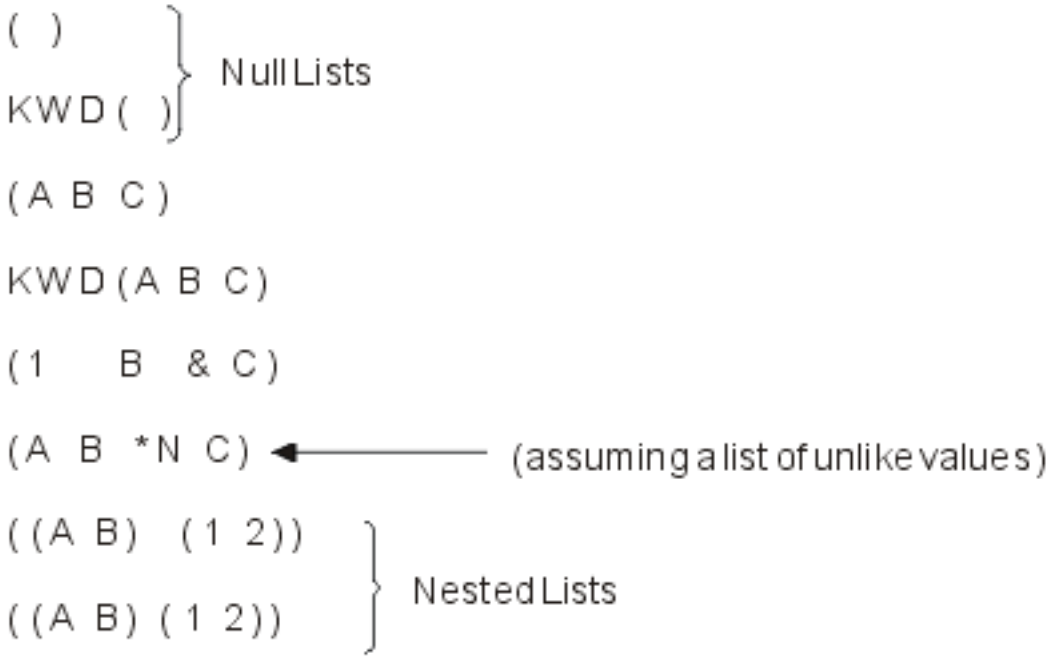
- A *simple list* parameter accepts one or more values of the type allowed by a parameter. For example, (RSMITH BJONES TBROWN) is a simple list of three user names.
- A *mixed list* parameter accepts a fixed set of separately defined values that are in a specific order. Each value can be defined with specific characteristics such as type and range. For example, LEN(5 2) is a

mixed list in which the first element (5) gives the length of a field and the second element (2) gives the number of decimal positions in that field.

- For many parameters defined to accept lists, predefined single values can be specified in place of a list of values. One of these single values can be the default value, which can be either specified or assumed if no list is specified for a simple or mixed list. To determine what defaults are accepted for a given list parameter, refer to the description of the parameter in the description of the command for which the parameter is defined and used.

  Note: *N cannot be specified in a simple list, but it can be specified in a mixed list. Also, individual parameters passed on the CALL and CALLPRC commands cannot be lists.
- The maximum level of nesting of lists inside lists is three, including the first. These are indicated by three nested levels of parentheses.

Here are examples of lists.



The last two examples contain two lists nested inside a list: the first list contains values of A and B, and the second list contains values of 1 and 2. The space between the two nested lists is not required. Blanks are the separators between the values inside each nested list, and the sets of parentheses group the nested values into larger lists.

Related reference

CL command parameters
Most CL commands have one or more *parameters* that specify the objects and values used to run the commands.

Parameters in keyword and positional form

You can specify parameters in CL using keyword form, positional form, or in a combination of the two.

Parameters in keyword form

A parameter in *keyword form* consists of a keyword followed immediately by a value (or a list of values separated by blank spaces) enclosed in parentheses. You cannot use blanks between the keyword and the left parenthesis preceding the parameter value. You can place blanks between the parentheses and the parameter value. For example, LIB(MYLIB) is a keyword parameter specifying that MYLIB is the name of the library that is used in some way, depending upon the command in which this LIB parameter is used.

When command parameters are all specified in keyword form, they can be placed in any order. For example, the following two commands are the same:

```
CRTLIB   LIB(MYLIB)  TYPE(*TEST)
CRTLIB   TYPE(*TEST)  LIB(MYLIB)
```

Parameters in positional form

A parameter in *positional form* does not have its keyword coded; it contains only the value (or values, if it is a list) whose function is determined by its position in the parameter set for that command. The parameter values are separated from each other and from the command name by one or more blank spaces. Because there is only one positional sequence in which parameters can be coded, the positional form of the previous CRTLIB (Create Library) command first example is:

```
CRTLIB   MYLIB  *TEST
```

If you do not want to enter a value for one of the parameters, the predefined value *N (null) can be entered in that parameter's position. The system recognizes *N as an omitted parameter, and either assigns a default value or leaves it null. In the previous CRTLIB command second example, if you coded *N instead of *TEST for the TYPE parameter, the default value *PROD is used when the command is run, and a production library named MYLIB is created. The description of the CRTLIB command contains the explanation for each parameter.

Notes:

- Parameters cannot be coded in positional form beyond the positional coding limit. If you attempt to code parameters in positional form beyond that point, the system returns an error message.
- Using positional form in your CL program source may save time when writing the program, but will be more difficult for you or someone else to maintain. Commands written using keyword form are generally easier to understand and enhance.

Combining keyword and positional forms

A CL command can also have its parameters coded in a combination of keyword and positional forms. The following examples show three ways to code the Declare CL Variable (DCL) command.

Keyword form:

```
DCL   VAR(&QTY)  TYPE(*DEC)  LEN(5)  VALUE(0)
```

Positional form:

```
DCL   &QTY  *DEC  5  0
```

Positional and keyword forms together:

```
DCL   &QTY  *DEC  VALUE(0)
```

In the last example, because the optional LEN parameter was not coded, the VALUE parameter must be coded in keyword form.

Note: You cannot specify parameters in position form after a parameter specified in keyword form.

Related concepts

CL command coding rules

This summary of general information about command coding rules can help you properly code CL commands.

Related reference

CL command parameters
Most CL commands have one or more *parameters* that specify the objects and values used to run the commands.

Required, optional, and key parameters

A CL command can have parameters that must be coded (required parameters) and parameters that do not have to be coded (optional parameters).

Optional parameters are typically assigned a system-defined default value if another value is not specified for the parameter when the command is entered.

A command can also have *key parameters* which are the only parameters shown on the display when a user prompts for the command. After values are entered for the key parameters, the remaining parameters are shown with actual values instead of the default values (such as *SAME or *PRV).

Related reference

CL command parameters
Most CL commands have one or more *parameters* that specify the objects and values used to run the commands.

Commonly used parameters

Some parameters are commonly used in many CL commands.

These commonly used parameters meet one or both of the following criteria:

- There is extensive information about how they are used.
- They are used in many of the CL commands (such as the AUT parameter), and the parameter description in the individual command description gives only the essential information.

The expanded descriptions of the applicable command parameters have been placed here for several reasons:

- To reduce the amount of material needed in the individual commands. Normally programmers familiar with a parameter's main function do not need the details.
- To provide the supplemental information that is useful to programmers in some instances.

The format for this information is designed for easy reference and includes a general description of each parameter that explains its function, states the rules for its use, and provides other helpful information. The values that can be specified for each parameter are also listed. Each value is followed by an explanation of what it means and (possibly) in which commands it is used. Not all of the values appear in every command. Refer to the individual command descriptions for the specific use of the value in that command parameter.

Related concepts

CL command coding rules
This summary of general information about command coding rules can help you properly code CL commands.

AUT parameter

The authority (AUT) parameter is used in create, grant, and revoke commands. It specifies the authority granted to all users of an object.

The AUT parameter also specifies an authorization list that is used to secure the object. Four object types allow the AUT parameter to contain an authorization list: LIB, PGM, DTADCT, and FILE. Public authority is an IBM i object attribute that controls the base set of rights to that object for all users having access to the system. These rights can be extended or reduced for specific users. If you specify an authorization list,

the public authority in the authorization list is the public authority for the object. The owner of an object has all authority to the object at its creation.

If the object is created as a private object or with the limited authority given to all users, the owner can grant more or less authority to specific users by specifically naming them and stating their authority in the **Grant Object Authority (GRTOBJAUT)** command. The owner also can withdraw specific authority from specific users, or from all users (publicly authorized, specifically authorized, or both) by using the **Revoke Object Authority (RVKOBJAUT)** command or the **Edit Object Authority (EDTOBJAUT)** command.

Values allowed

***LIBCRTAUT**
The public authority for the object is taken from the value on the CRTAUT parameter of the target library (the library that is to contain the object). The public authority is determined when the object is created. If the CRTAUT value for the library changes after the object is created, the new value does not affect any existing objects.

***USE**
You can perform basic operations on the object, such as running a program or reading a file. The user cannot change the object. *USE authority provides object operational authority, read authority, and execute authority.

***CHANGE**
You can perform all operations on the object except those limited to the owner or controlled by object existence authority and object management authority. You can change and perform basic functions on the object. Change authority provides object operational authority and all data authority.

***ALL**
You can perform all operations except those limited to the owner or controlled by authorization list management authority. Your can control the object's existence, specify the security for the object, change the object, and perform basic functions on the object. You also can change ownership of the object.

***EXCLUDE**
You cannot access the object.

***EXECUTE**
You can run a program or procedure or search a library or directory.

authorization-list-name
Specify the name of the authorization list whose authority is used.

Related information
Security reference

CLS parameter

The class (CLS) parameter identifies the attributes that define the run time environment of a job.

The following attributes are defined in each class:

- Run priority: A number that specifies the priority level assigned to all jobs running that use the class. The priority level is used to determine which job, of all the jobs competing for system resources, is run next. The value can be 1 through 99, where 1 is the highest priority (all jobs having a 1 priority are run first).
- Time slice: The maximum amount of processor time that the system allows the job to run when it is allowed to begin. The time slice indicates the amount of time needed for the job to accomplish a meaningful amount of work (the time used by the system for reading auxiliary storage is not charged against the time slice). When the time slice ends, the job waits while other queued jobs of the same or higher priority are allowed to run (up to the time specified in their time slices); then the job is given another time slice.

- Purge value: Indicates whether the job step is eligible to be moved from main storage to auxiliary storage while the job is waiting for some resource before it can continue, or when its time slice is used up and equal or higher priority jobs are waiting.
- Default wait time: The default amount of time that the system waits for the completion of an instruction that performs a wait. This wait time applies to times when an instruction is waiting for a system action, not to the time an instruction is waiting for a response from a user. Normally, this would be the amount of time you are willing to wait for the system before ending the request. If the wait time is exceeded, an error message is passed to the job. This default wait time applies only when a wait time is not specified in the CL command that causes the wait.

  The wait time used for allocating file resources is specified in the file description and can be overridden by an override command. It specifies that the wait time specified in the class object is used. If file resources are not available when the file is opened, the system waits for them until the wait time ends.

  Note: The class attributes apply to each routing step of a job. Most jobs have only one routing step, but if the job is rerouted (because of something like the **Remote Job** or **Transfer Job** command) the class attributes will be reset.
- Maximum CPU time: The maximum amount of processor time (the sum of the time slices if they differ, or time slice period multiplied by number of time slices if they are equal) allowed for a job's routing step to complete processing. If the job's routing step is not completed in this amount of time, it is ended, and a message is written to the job log.
- Maximum temporary storage: The maximum amount of temporary storage that can be used by a job's routing step. This temporary storage is used for the programs that run in the job, for the system objects used to support the job, and for temporary objects created by the job.

The system is shipped with a set of classes that define the attributes for several job processing environments. Other classes can be created by the **Create Class (CRTCLS)** command; any class can be displayed or deleted by the respective **Display Class (DSPCLS)** and **Delete Class (DLTCLS)** commands.

Values allowed

qualified-class-name
Specify the name of the class, optionally qualified by the name of the library in which the class is stored. If the class name is not qualified and the CLS parameter is in the CRTCLS command, the class object is stored in *CURLIB; otherwise, the library list (*LIBL) is used to find the class name.

Classes

The following classes (by name) are supplied with the system:

QGPL/QBATCH
For use by batch jobs

QSYS/QCTL
For use by the controlling subsystem

QGPL/QINTER
For use by interactive jobs

QGPL/QPGMR
For use by the programming subsystem

QGPL/QSPL
For use by the spooling subsystem printer writer

QGPL/QSPL2
For general spooling use in the base system pool

COUNTRY parameter

The COUNTRY parameter specifies the country or region code.

The country or region code is part of the X.400 O/R name. An ISO 3166 Alpha-2 code or an ITU-T country or region code can be specified. (The ITU-T country or region code is the data country or region or geographical area code published in the "International Numbering Plan for Public Data Networks," Recommendation X.121 (09/92), by the ITU-T (formerly CCITT). The following table contains a list of the possible country or region codes that can be specified.

Values allowed

***NONE**
No country or region code is specified.

country-code
Specify an ISO 3166 Alpha-2 code or a CCITT (also known as ITU-2) country or region code from the following table.

Table 8. ISO X.400 country or region codes

Country or region	ISO 3166 alpha-2 code	ITU-T [1] country or region code
Afghanistan	AF	412
Albania	AL	276
Algeria	DZ	603
American Samoa	AS	544
Andorra	AD	
Angola	AO	631
Anguilla	AI	
Antarctica	AQ	
Antigua and Barbuda	AG	344
Argentina	AR	722
Armenia	AM	283
Aruba	AW	362
Australia	AU	505
Austria	AT	232
Azerbaijan	AZ	400
Bahamas	BS	364
Bahrain	BH	426
Bangladesh	BD	470
Barbados	BB	342
Belarus	BY	257
Belgium	BE	206
Belize	BZ	702
Benin	BJ	616
Bermuda	BM	350

Table 8. ISO X.400 country or region codes (continued)		
Country or region	**ISO 3166 alpha-2 code**	**ITU-T [1] country or region code**
Bhutan	BT	
Bolivia	BO	736
Bosnia and Herzegovina	BA	
Botswana	BW	652
Bouvet Island	BV	
Brazil	BR	724
British Indian Ocean Terr.	IO	
Brunei Darussalam	BN	528
Bulgaria	BG	284
Burkina Faso	BF	613
Burundi	BI	642
Cambodia	KH	456
Cameroon	CM	624
Canada	CA	302, 303
Cape Verde	CV	625
Cayman Islands	KY	346
Central African Republic	CF	623
Chad	TD	622
Chile	CL	730
China	CN	460
Christmas Island	CX	
Cocos (Keeling) Islands	CC	
Colombia	CO	732
Comoros	KM	654
Congo	CG	629
Cook Islands	CK	548
Costa Rica	CR	712
Cote d'Ivoire	CI	612
Croatia	HR	
Cuba	CU	368
Cyprus	CY	280
Czech Republic	CZ	230
Denmark	DK	238
Djibouti	DJ	638

Table 8. ISO X.400 country or region codes (continued)		
Country or region	**ISO 3166 alpha-2 code**	**ITU-T [1] country or region code**
Dominica	DM	366
Dominican Republic	DO	370
East Timor	TP	
Ecuador	EC	740
Egypt	EG	602
El Salvador	SV	706
Equatorial Guinea	GQ	627
Eritrea	ER	
Estonia	EE	248
Ethiopia	ET	636
Falkland Islands (Malvinas)	FK	
Faroe Islands	FO	288
Fiji	FJ	542
Finland	FI	244
France	FR	208, 209
France, Metropolitan	FX	
French Antilles		340
French Guiana	GF	742
French Polynesia	PF	547
French Southern Terr.	TF	
Gabon	GA	628
Gambia	GM	607
Georgia	GE	282
Germany	DE	262 - 265
Ghana	GH	620
Gibralter	GI	266
Greece	GR	202
Greenland	GL	290
Grenada	GD	352
Guadeloupe	GP	
Guam	GU	535
Guatemala	GT	704
Guinea	GN	611
Guinea-Bissau	GW	632

Table 8. ISO X.400 country or region codes (continued)		
Country or region	**ISO 3166 alpha-2 code**	**ITU-T [1] country or region code**
Guyana	GY	738
Haiti	HT	372
Heard and Mc Donald Islands	HM	
Honduras	HN	708
China (Hong Kong S.A.R.)	HK	453, 454
Hungary	HU	216
Iceland	IS	274
India	IN	404
Indonesia	ID	510
Iran	IR	432
Iraq	IQ	418
Ireland	IE	272
Israel	IL	425
Italy	IT	222
Jamaica	JM	338
Japan	JP	440 - 443
Jordan	JO	416
Kazakhstan	KZ	401
Kenya	KE	639
Kiribati	KI	545
Korea, Democratic People's Republic	KP	467
Korea, Republic of	KR	450, 480, 481
Kuwait	KW	419
Kyrgyzstan	KG	437
Lao People's Democratic Rep.	LA	457
Latvia	LV	247
Lebanon	LB	415
Lesotho	LS	651
Liberia	LR	618
Libyan Arab Jamahiriya	LY	606
Liechtenstein	LI	
Lithuania	LT	246
Luxembourg	LU	270
China (Macau S.A.R.)	MO	455

Table 8. ISO X.400 country or region codes (continued)		
Country or region	**ISO 3166 alpha-2 code**	**ITU-T [1] country or region code**
Macedonia [2]	MK [2]	
Madagascar	MG	646
Malawi	MW	650
Malaysia	MY	502
Maldives	MV	472
Mali	ML	610
Malta	MT	278
Marshall Islands	MH	
Martinique	MQ	
Mauritania	MR	609
Mauritius	MU	617
Mayotte	YT	
Mexico	MX	334
Micronesia	FM	550
Moldova, Republic of	MD	259
Monaco	MC	212
Mongolia	MN	428
Montenegro [2]	ME [2]	
Montserrat	MS	354
Morocco	MA	604
Mozambique	MZ	643
Myanmar	MM	414
Namibia	NA	649
Nauru	NR	536
Nepal	NP	429
Netherlands	NL	204, 205
Netherlands Antilles	AN	362
New Caledonia	NC	546
New Zealand	NZ	530
Nicaragua	NI	710
Niger	NE	614
Nigeria	NG	621
Niue	NU	
Norfolk Island	NF	

Table 8. ISO X.400 country or region codes (continued)		
Country or region	**ISO 3166 alpha-2 code**	**ITU-T [1] country or region code**
Northern Mariana Islands	MP	534
Norway	NO	242
Oman	OM	422
Pakistan	PK	410
Palau	PW	
Panama	PA	714
Papua New Guinea	PG	537
Paraguay	PY	744
Peru	PE	716
Philippines	PH	515
Pitcairn	PN	
Poland	PL	260
Portugal	PT	268
Puerto Rico	PR	330
Qatar	QA	427
Reunion	RE	647
Romania	RO	226
Russian Federation	RU	250, 251
Rwanda	RW	635
St. Helena	SH	
St. Kitts and Nevis	KN	356
St. Lucia	LC	358
St. Pierre and Miquelon	PM	308
St. Vincent and the Grenadines	VC	360
Samoa, Western	WS	549
San Marino	SM	292
Sao Tome and Principe	ST	626
Saudi Arabia	SA	420
Senegal	SN	608
Serbia [2]	SP [2]	
Seychelles	SC	633
Sierra Leone	SL	619
Singapore	SG	525
Slovakia	SK	

Table 8. ISO X.400 country or region codes (continued)		
Country or region	**ISO 3166 alpha-2 code**	**ITU-T [1] country or region code**
Slovenia	SI	
Solomon Islands	SB	540
Somalia	SO	637
South Africa	ZA	655
South Georgia and the S.S.I	GS	
Spain	ES	214
Sri Lanka	LK	413
Sudan	SD	634
Suriname	SR	746
Svalbard and Jan Mayen Is.	SJ	
Swaziland	SZ	653
Sweden	SE	240
Switzerland	CH	228
Syrian Arab Republic	SY	417
Taiwan	TW	466
Tajikistan	TJ	436
Tanzania, United Republic of	TZ	640
Thailand	TH	520
Togo	TG	615
Tokelau	TK	
Tonga	TO	539
Trinidad and Tobago	TT	374
Tunisia	TN	605
Turkey	TR	286
Turkmenistan	TM	438
Turks and Caicos Islands	TC	376
Tuvalu	TV	
Uganda	UG	641
Ukraine	UA	255
United Arab Emirates	AE	424, 430, 431
United Kingdom	GB	234, 235, 236, 237
United States	US	310 - 316
United States Minor Outlying Is.	UM	
Uruguay	UY	748

Table 8. ISO X.400 country or region codes (continued)		
Country or region	**ISO 3166 alpha-2 code**	**ITU-T [1] country or region code**
Uzbekistan	UZ	434
Vanuatu	VU	541
Vatican City State (Holy See)	VA	225
Venezuela	VE	734
Viet Nam	VN	452
Virgin Is. (Brit.)	VG	348
Virgin Is. (U.S.)	VI	332
Wallis and Futuna Is.	WF	543
Western Sahara	EH	
Yemen	YE	421, 423
Yugoslavia, territories of the former	YU	220
Zaire	ZR	630
Zambia	ZM	645
Zimbabwe	ZW	648

Notes:

1. This International Telecommunication Union (ITU) committee was formerly known as CCITT.
2. At the time of publication, the ISO 3166 Alpha-2 Code for this country or region could not be confirmed. Before using this code, be sure to confirm with the latest ISO 3166 standard.

FILETYPE parameter

The FILETYPE parameter specifies whether the database file description describes data records or source records.

Further, the FILETYPE parameter specifies whether each member of a database file being created is to contain data records or source records (statements). For example, the file could contain RPG source statements for an RPG program or data description source (DDS) statements for another device or database file.

Note: If you are creating a source type *physical* database file and are not providing field-level descriptions of the file (through data description specifications (DDS)), you can use either the **Create Physical File (CRTPF)** command or the **Create Source Physical File (CRTSRCPF)** command. However, the CRTSRCPF command is typically more convenient and efficient, because it is designed to be used to create source physical files. If DDS is provided when you are creating a source type database file, you should use the CRTPF command or the **Create Logical File (CRTLF)** command, which both have the SRCFILE and SRCMBR parameters for specifying source input.

Records in a source file must have at least three fields: the first two are the source sequence number field and the date field; the third field contains the source statement. These three fields are automatically provided by the IBM i operating system when a source file is created for which no DDS is provided; additional source fields can be defined in DDS. The length of the sequence number field must be six zoned digits with two decimal places. The length of the date field must be six zoned digits with no decimal places.

The source sequence number and date fields are added to the source record when:

- Records are read into the system.

- Records are created by the Source Entry Utility (which is part of the licensed WebSphere® Development Studio program).

The fields are added when an inline data file (specified as the standard source file format) is read from the device. The spooling reader places a sequence number in the source sequence number field and sets up a zeroed date field.

If those fields already exist in records read from the device, they are not changed. If the records in a database file are in source format and are being read as an inline data file in data format, the source sequence number and date fields are removed.

Values allowed

***DATA**
The file created contains or describes data records.

***SRC**
The file created contains or describes source records. If the file is keyed, the 6-digit source sequence number field must be used as the key field.

Related information

Database programming

FRCRATIO parameter

The force write ratio (FRCRATIO) parameter specifies the maximum number of records that can be inserted, updated, or deleted before they are forced into auxiliary (permanent) storage.

The force write ratio ensures that all inserted, updated, or deleted records are written into auxiliary storage at least as often as this parameter specifies. In the event of system failure, the only records likely to be lost would be those that were inserted, updated, or deleted since the last force write operation.

The force write ratio is applied to all records inserted, updated, or deleted in the file through the open data path (ODP) to which the force write ratio applies. If two programs are sharing the file, SHARE(*YES), the force write ratio is not applied separately to the set of records inserted, updated, or deleted by each program. It is applied to any combination of records (from both programs) that equals the specified force write ratio parameter value. For example, if a force write ratio of 5 was specified for the file, any combination of five records from the two programs (such as four from one program and one from the other) forces the records to be written to auxiliary storage. If two or more programs are using the file through separate ODPs, the insertions, updates, and deletions from each program are accumulated individually for each ODP.

Each database file can have a force write ratio assigned to it. Logical files, which can access data from more than one physical file, can specify a more restrictive force write ratio (a smaller number of records) than that specified for the based-on physical files. However, a logical file cannot specify a less restrictive force write ratio. If a logical file specifies a less restrictive force write ratio than that specified for any of the physical files, the most restrictive force write ratio from the physical files is used for the logical file. For example, if the force write ratios of three physical files are 2, 6, and 8, the force write ratio of a logical file based on these physical files cannot be greater than 2. If no force write ratio is specified for the logical file, 2 is assumed. Thus, each time a program inserts, updates, or deletes two records in the logical file (regardless of which physical files are affected), those records are forced into auxiliary storage.

The FRCRATIO number overrides the SEQONLY number specified. For example, if you specify:

```
OVRDBF ... SEQONLY(*YES 20) FRCRATIO(5)
```

The value of 20 is overridden and a buffer of five records is used. When FRCRATIO(1) is used, a buffer still exists, but it contains only a single record.

Access paths associated with the inserted, updated, and deleted records are written to auxiliary storage only when all the records covered by the access path have been written to auxiliary storage. If only one ODP exists for the file, the access path is forced to auxiliary storage whenever a forced write occurs. If two

or more ODPs to the file exist, the access path is written to auxiliary storage whenever all the inserted, updated, and deleted records for all the ODPs have been forced.

Notes:

1. These rules apply only when a force write ratio of 2 or higher is specified. When a force write ratio of 1 is specified, the access path is not written to auxiliary storage until all the ODPs have been closed.
2. If the file is being recorded in a journal, FRCRATIO(*NONE) should be specified.

Values allowed

***NONE**
There is no specified ratio; the system determines when the records are written to auxiliary storage.

number-of-records-before-force
Specify the number of updated, inserted, or deleted records that are processed before they are explicitly forced to auxiliary storage.

Related information

Journal management and system performance

IGCFEAT parameter

The IGCFEAT parameter specifies which double-byte character set (DBCS) table is used, according to device and language.

The following table indicates the corresponding IGCFEAT parameter and DBCS font table for the double-byte character set device being configured.

Table 9. DBCS features configurable on the IGCFEAT parameter

Language/device	Type of physical DBCS workstation	Configure as type-model	Configure with DBCS feature
Japanese Display Stations	5295-001 Display, 5295-002 Display, InfoWindow 3477-J Display, PS/55 with 5250PC, PS/55 with graphics 5250PC, PS/55 with graphics 5250PC, PS/55 with 5250PC/2, 3270-type Display, PS/55 with IBM System i Access	5555-B01 5555-B01 5555-B01, C01 5555-B01 5555-G01 5555-G02 5555-E01 3279-0 5555-B01	((2424J4 55FE)) ((2424J4 68FE)) ((2424J4 68FE)) ((2424J4 68FE)) ((2424J4 68FE)) ((2424J4 68FE)) ((2424J0 (1))) ((2424J0 (1))) ((2424J0 (1)))
Japanese 24x24 Printers	Attached to 5295-001 Display Attached to 5295-002 Display Attached to PS/55 5227-001 Printer 5327-001 Printer	5553-B01 5553-B01 5553-B01 5553-B01 5553-B01	((2424J1 55FE)) ((2424J1 68FE)) ((2424J1 68FE)) ((2424J2 55FE)) ((2424J2 68FE))
Japanese 32x32 Printers	5337-001 Printer 5383-200 Printer	5553-B01 5583-200	((3232J0 (1))) ((3232J0 (1)))
Korean Display Stations	5250-Type Display 3270-Type Display	5555-B01 3279-0	((2424K0 (1))) ((2424K0 (1)))
Korean 24x24 Printers	Attached to 5295 Display Attached to PS/55 5227-002 Printer	5553-B01 5553-B01 5553-B01	((2424K0 (1))) ((2424K0 (1))) ((2424K2 52FE))

Table 9. DBCS features configurable on the IGCFEAT parameter (continued)			
Language/device	**Type of physical DBCS workstation**	**Configure as type-model**	**Configure with DBCS feature**
Traditional Chinese Display Stations	5250-Type Display 3270-Type Display	5555-B01 3279-0	((2424C0)) ((2424C0))
Traditional Chinese 24x24 Printers	Attached to 5295 Display Attached to PS/55 5227-003 Printer	5553-B01 5553-B01 5553-B01	((2424C0)) ((2424C0)) ((2424C2 5CFE))
Simplified Chinese Display Stations	5250-Type Display 3270-Type Display	5555-B01 3279-0	((2424S0)) ((2424S0))
Simplified Chinese 24x24 Printers	Attached to PS/55 5227-005 Printer	5553-B01 5553-B01	((2424S0)) ((2424S2 6FFE))

JOB parameter

The JOB parameter specifies the name of the job to which the command is applied.

The job name identifies all types of jobs on the system. Each job is identified by a qualified job name, which has the following format:

`job-number/user-name/job-name`

Note: Although the syntax is similar, job names are qualified differently than IBM i object names.

The following list describes the pieces of the qualified job name:

- Job number: The job number is a unique 6-digit number that is assigned to each job by the system. The job number provides a unique qualifier if the job name is not otherwise unique. The job number can be determined by means of the **`Display Job (DSPJOB)`** command. If specified, the job number must have exactly six digits.
- User name: The user name identifies the user profile under which the job is to run. The user name is the same as the name of the user profile and contains a maximum of 10 alphanumeric characters. The name can come from one of several sources, again, depending on the type of job:
  - Batch job: The user name is specified on the SBMJOB command, or it is specified in the job description referenced by the BCHJOB or SBMJOB commands.
  - Interactive job: The user name is specified at sign-on, or the user name is provided from the default in the job description referred to by the work station's job entry.
  - Autostart job: The user name is specified in the job description referred to by the job entry for the autostart job.
- Job name: The job name can contain a maximum of 10 alphanumeric characters, of which the first character must be alphabetic. The name can come from one of three sources, depending on the type of job:
  - Batch job: The job name is specified on the **`Batch Job (BCHJOB)`** or **`Submit Job (SBMJOB)`** commands or, if not specified there, the unqualified name of the job description is used.
  - Interactive job: The job name is the same as the name of the device (work station) from which the sign-on was performed.
  - Autostart job: The job name is provided in the autostart job entry in the subsystem description under which the job runs. The job name was specified in the **`Add Autostart Job Entry (ADDAJE)`** command.

Commands only require that the simple name be used to identify the job. However, additional qualification must be used if the simple job name is not unique.

Duplicate job names

If a duplicate job name is specified in a command in an *interactive* job, the system displays all of the duplicates of the specified job name to the user in qualified form. The job names are displayed in qualified form along with the user name and job number so that you can further identify the job that is to be specified in a command. You can then enter the correct qualified job name.

If a duplicate job name is used in a command in a *batch* job, the command is not processed. Instead, an error message is written to the job log.

The JOB parameter can have one or more of the following values, depending upon the command:

Values allowed

*****
The job is the one in which the command is entered; that is, the command with JOB(*) specified on it.

***JOBD**
The simple job name is the unqualified name of the job description.

***NONE**
No job name is specified as in the Display Log (DSPLOG) command.

job-name
A simple job name is specified.

qualified-job-name
You must specify a qualified job name. If no job qualifier (user name and job number) is given, all of the jobs currently in the system are searched for the job name. If duplicates of the specified name are found, a qualified job name must be specified.

Related concepts

Simple and qualified object names
The name of a specific object that is located in a library can be specified as a simple name or as a qualified name.

LABEL parameter

The LABEL parameter specifies the data file identifier of the data file on tape used in input and output operations.

The data file can be in either the exchange format or the save/restore format.

Note: The device file commands are used for tapes that are in the exchange format only, not for those in the save/restore format; user-defined device files are not used in save/restore operations.

Each data file on tape has its data file identifier stored in its own file label. The data file label (or header label) of each data file is stored on the tape just before the data in the file. That is, each file on the tape has its own header label and its own data records together as a unit, and one file follows another. In addition to the data file identifier, each label also contains other information about the file, such as the file sequence number, record and block attributes, and whether it is a multivolume data file.

Generally, the data file identifier is an alphanumeric character string that contains no more than 8 characters. However, the maximum length actually depends on several things: what data format is used for the files and CL commands in which the identifiers are specified. The unused portion of the file identifier field should be left blank.

The first character of the data file identifier must be alphabetic (A through Z, $, #, or @) and the rest of the characters should be alphanumeric (A through Z, 0 through 9, $, #, _, ., and @). You can use special characters if the identifier is enclosed in single quotation marks. However, if the tape is used on an operating system other than IBM i, the requirements for specifying identifiers must be considered.

Tape data file identifiers

Tape data file identifiers can have as many as 17 characters. However, if the tape is used on an operating system other than IBM i, a maximum of 8 characters or a qualified identifier of no more than 17

characters should be used. If more than 8 characters are used, the identifier should be qualified and enclosed in single quotation marks so that no more than 8 characters occur in either part, and the parts are separated by a period; for example, LABEL('TAXES.JAN1980'). This limitation applies to the following commands: **Create Tape File (CRTTAPF)**, **Change Tape File (CHGTAPF)**, **Override Tape File (OVRTAPF)**, and **Display Tape (DSPTAP)**.

The data file identifier is put on the volume when the data file is put on the volume. For input/output operations, the identifier can be specified in one of tape device file commands, or it can be passed as a parameter when the device file is opened by the high-level language program that uses the file.

Save/restore format

For tapes in the save/restore format, the identifier can have a maximum of 17 characters. If a library name is used to generate the label, the identifier cannot exceed 10 characters. You may specify a label other than a library name.

Values allowed

One of the following values can be specified for the LABEL parameter, depending upon the command.

***ALL**
Labels for all the data file identifiers in the specified tape volumes are shown on the display.

***NONE**
The data file identifier is not specified. It must be supplied before the device file, database file, or both are opened to be used in the tape operation.

***SAME**
The data file identifier already present in the tape device file does not change.

data-file-identifier
Specify the identifier of the data file used or displayed with the device file description.

***LIB**
The file label is created by the system and the name of the library specified on the LIB parameter is used as the qualifier for the file name.

***SAVLIB**
The file label is created by the system, and the name of the library specified on the SAVLIB parameter is used as the qualifier for the file name.

LICOPT parameter

The Licensed Internal Code options (LICOPT) parameter allows you to specify individual compile-time options.

This parameter is intended for the advanced programmer who understands the potential benefits and drawbacks of each selected type of compiler option.

The following table shows the strings that are recognized by the Licensed Internal Code option (LICOPT) parameter. These strings are not case sensitive, but they are shown as mixed case for readability.

Table 10. LICOPT parameter strings

String	Description
AllFieldsVolatile	If set, treats all fields as volatile.
NoAllFieldsVolatile	If set, no fields are treated as volatile.
AllowBindingToLoadedClasses	Indicates that temporary class representations that were created as a result of defineClass calls within a running Java virtual machine may be tightly bound to other class representations within the same Java virtual machine.

Table 10. LICOPT parameter strings (continued)	
String	**Description**
NoAllowBindingToLoadedClasses	Indicates that temporary class representations that were created as a result of defineClass calls within a running Java virtual machine may not be tightly bound to other class representations within the same Java virtual machine.
AllowClassCloning	When multiple Java programs are generated for a JAR file, allows copies of classes from one program to be included in the generated code for another program. Facilitates aggressive inlining.
NoAllowClassCloning	Does not allow copies of classes from one program to be included in the generated code for another program.
AllowInterJarBinding	Allows tight binding to classes outside the class or JAR file being compiled. Facilitates aggressive optimizations.
NoAllowInterJarBinding	Does not allow tight binding to classes outside the class or JAR file being compiled. This overrides the presence of the CLASSPATH and JDKVER parameters on CRTJVAPGM.
AllowMultiThreadedCreate	CRTJVAPGM uses multiple threads, if they are available, during creation. On multiprocessor systems this enables the use of more than one processor at a time, reducing the overall time required for a long CRTJVAPGM operation. However, the CRTJVAPGM will use more system resources, leaving fewer resources available for other applications.
NoAllowMultiThreadedCreate	Indicates that CRTJVAPGM performs as usual, using only one thread.
AnalyzeObjectLifetimes	Performs analysis using visible classes to determine which objects are short-lived. A short-lived object does not outlive the method in which it is allocated, and may be subject to more aggressive optimizations.
NoAnalyzeObjectLifetimes	Does not perform analysis of short-lived objects.
AllowBindingWithinJar	Indicates that class representations within a ZIP file or JAR file may be tightly bound to other class representations within the same ZIP file or JAR file.
NoAllowBindingWithinJar	Indicates that class representations within a ZIP file or JAR file may not be tightly bound to other class representations within the same ZIP file or JAR file.
AllowInlining	Tells the translator that it is permitted to inline local methods. This is the default for optimization levels 30 and 40.
NoAllowInlining	Does not tell the translator that it is permitted to inline local methods.
AssumeUnknownFieldsNonvolatile	When the attributes of a field in an external class cannot be determined, this parameter generates code by assuming that the field is non-volatile.
NoAssumeUnknownFieldsNonvolatile	When the attributes of a field in an external class cannot be determined, this parameter generates code by assuming that the field is volatile.

Table 10. LICOPT parameter strings (continued)

String	Description
BindErrorHandling	Specifies what action should be taken if, as a result of honoring the AssumeUnknownFieldsNonvolatile, PreresolveExtRef, or PreLoadExtRef Licensed Internal Code option, the Java virtual machine class loader detects that a class representation contains method representations, which cannot be used in the current context.
BindInit	Uses bound call to local init methods.
NoBindInit	Does not use bound call to local init methods.
BindSpecial	Uses bound call to local special methods.
NoBindSpecial	Does not use bound call to local special methods.
BindStatic	Uses bound call to local static methods.
NoBindStatic	Does not use bound call to local static methods.
BindTrivialFields	Binds trivial field references during program creation.
NoBindTrivialFields	Resolves field references at first touch.
BindVirtual	Uses bound call to local final virtual methods.
NoBindVirtual	Does not use bound call to local final virtual methods.
DeferResolveOnClass	Takes a string parameter that is presumed to be the name of a class (for example, java.lang.Integer). When you set PreresolveExtRef to optimization level 40, classes that are specified with DeferResolveOnClass are not in the preresolve operation. This is useful if some classes in unused paths in the code are not in the CLASSPATH. It allows you to use optimization level 40 regardless of this by specifying a "DeferResolveOnClass='somepath.someclass'" for each missing class. Multiple DeferResolveOnClass entries are allowed.
DevirtualizeFinalJDK	Allows CRTJVAPGM to use knowledge of the standard JDK to devirtualize calls to those JDK methods that are known to be final methods or members of final classes. It is the default at optimization levels 30 and 40.
NoDevirtualizeFinalJDK	Does not allow CRTJVAPGM to use knowledge of the standard JDK to devirtualize calls to those JDK methods that are known to be final methods or members of final classes.
DevirtualizeRecursive	Causes special code to be generated in the case of some recursive methods and eliminates much of the overhead of the recursive method calls. However, additional checking logic is generated on initial entry to the recursive method, so performance may not improve in cases of shallow recursion.
NoDevirtualizeRecursive	Does not cause special code to be generated in the case of some recursive methods.
DisableIntCse	Causes certain common subexpression optimizations to be disabled when generating code for certain types of integer expressions. This may improve overall optimization by exposing other optimization opportunities to the Optimizing Translator.

Table 10. LICOPT parameter strings (continued)	
String	**Description**
NoDisableIntCse	Causes certain common subexpression optimizations to not be disabled when generating code for certain types of integer expressions. This generally results in better performing code at lower optimization levels.
DoExtBlockCSE	Performs extended basic block common subexpression elimination.
NoDoExtBlockCSE	Does not perform extended basic block common subexpression elimination.
DoLocalCSE	Performs local common subexpression elimination.
NoDoLocalCSE	Does not perform local common subexpression elimination.
EnableCseForCastCheck	If set, generates code for castcheck that can be DAGed to an earlier instance.
NoEnableCseForCastCheck	Is not set; does not generate code for castcheck that can be DAGed to an earlier instance.
ErrorReporting	Runtime error reporting field**: Provides the option to fail the compile when encountering verification or class format errors. 0=Report all errors immediately; 0=Report all errors immediately; 1=Defer reporting of bytecode verification errors; 2=Defer reporting of bytecode verification errors and class format errors to runtime.
HideInternalMethods	Causes methods in cloned classes to be made internal, allowing the methods to be omitted if there are no references to them or if all references are inlined. The default is HideInternalMethods for optimization 40 and NoHideInternalMethods for optimization between 0 and 30.
InlineArrayCopy	Causes the inlining of the System.arraycopy method in some cases of scalar arrays.
NoInlineArrayCopy	Prevents the inlining of the System.arraycopy method.
InlineInit	Inlines init methods for java.lang classes.
NoInlineInit	Does not inline init methods.
InlineMiscFloat	Inlines miscellaneous float/double methods from java.lang.Math.
NoInlineMiscFloat	Does not inline miscellaneous float/double methods.
InlineMiscInt	Inlines miscellaneous int/long methods from java.lang.Math.
NoInlineMiscInt	Does not inline miscellaneous int/long methods.
InlineStringMethods	Permits inlining of certain methods from java/lang/String.
NoInlineStringMethods	Inhibits inlining of certain methods from java/lang/String.
InlineTransFloat	Inlines transcendental float/double methods from java.lang.Math.
NoInlineTransFloat	Does not inline transcendental float/double methods.
OptimizeJsr	Generates better code for "jsr" bytecodes that have a single target.

Table 10. LICOPT parameter strings (continued)	
String	**Description**
NoOptimizeJsr	Suppresses generation of better code for "jsr" bytecodes that have a single target.
PreloadExtRef	Indicates that referenced classes may be preloaded (without class initialization) upon method entry.
NoPreloadExtRef	Indicates that referenced classes may not be preloaded upon method entry. However, the PreresolveExtRef parameter overrides this setting and causes referenced classes to be preloaded and initialized.
PreresolveExtRef	Preresolves referenced methods at method entry.
NoPreresolveExtRef	Resolves method references at first touch. Use to resolve "class not found" exceptions on programs that run on other machines.
ProgramSizeFactor	When a JAR file may be large enough to require multiple Java programs, this numeric value (default 100) is used to determine how large each program can grow.
ShortCktAthrow	If set, attempt to short-circuit athrows.
NoShortCktAthrow	Is not set, does not attempt to short-circuit athrows.
ShortCktExSubclasses	If set, recognizes some subclasses of Exception and short-circuit them directly.
NoShortCktExSubclasses	If not set, does not recognize some subclasses of Exception and short-circuit them directly.
StrictFloat	Inhibits floating-point optimizations that are not strictly compliant with the Java specification.
NoStrictFloat	Permits floating-point optimizations that are not strictly compliant with the Java specification.

The double asterisk (**) signifies that these strings require a numerical value for input in the syntax of stringname=number (with no spaces in between).

MAXACT parameter

The maximum activity level (MAXACT) parameter specifies the maximum number of jobs that can be concurrently started and that remain active through a job queue entry, communications entry, routing entry, or workstation entry.

A job is considered active from the time it starts running until it is completed. This includes time when:

- The job is actually being processed.
- The job is waiting for a response from a workstation user.
- The job is started and available for processing but is not actually using the processor. For example, it might have used up its time slice and is waiting for another time slice.
- The job is started but is not available for processing. For example, it could be waiting for a message to arrive on its message queue.

Values allowed

***NOMAX**
There is no maximum number of jobs that can be active at the same time.

maximum-active-jobs
Specify a value that indicates the maximum number of jobs that can be concurrently active through this entry.

Related information
Work management

OBJ parameter

The object (OBJ) parameter specifies the names of one or more objects affected by the command in which this parameter is used.

If the OBJ parameter identifies objects that must exist in an IBM i library, all of the objects must be in one of the following, depending upon which command is used:

- the library specified in the LIB parameter,
- the SAVLIB parameter,
- the library qualifier in the OBJ parameter,
- or the library part of the path name in the OBJ parameter

On some commands, the generic name of a group of objects can be specified. To form a generic name, add an asterisk (*) after the last character in the common group of characters; for example, ABC*. If an * is not included with the name, the system assumes that the name is a complete object name.

Values allowed

Depending on the command, the following types of values can be specified on the OBJ parameter:

- *ALL
- Simple object name
- Qualified object name
- Generic object name
- Qualified generic object name
- Path name

Related concepts
Path names (*PNAME)
A *path name* is a character string that can be used to locate objects in the integrated file system.

OBJTYPE parameter

The object type (OBJTYPE) parameter specifies the types of IBM i objects that can be operated on by the command in which they are specified.

The object types that can be specified in the OBJTYPE parameter vary from command to command.

The object-related commands allow you to perform general functions on most objects without knowing the special commands related to the specific object type. For example, you could use the **Create Duplicate Object (CRTDUPOBJ)** command to create a copy of a file or library instead of the specific commands **Copy File (CPYF)** or **Copy Library (CPYLIB)**.

Object-related commands

This section lists commands containing the OBJTYPE parameter. See the information for the individual commands listed to find out which object types can be operated on using the commands.

The following commands contain the OBJTYPE parameter but operate on only a few object types.

- CHKDLO operates on *DOC and *FLR.
- CPROBJ and DCPOBJ operate on *FILE, *MENU, *MODULE, *PGM, *PNLGRP, and *SRVPGM.
- CRTSQLPKG operates on *PGM and *SRVPGM.
- DSPPGMADP operates on *PGM, *SQLPKG, and *SRVPGM.

- DSPPGMREF operates on *PGM and *SQLPKG.
- RSTCFG operates on *CFGL, *CNNL, *COSD, *CTLD, *DEVD, *LIND, *MODD, and *NWID.
- SAVLICPGM operates on *LNG and *PGM.
- SETOBJACC operates on *FILE and *PGM.

The DSPLNK and WRKLNK commands operate on all object types.

The ALCOBJ and DLCOBJ commands also require that an object type value is specified. However, for these commands, the object type value is specified as one of four values (in a list of values) on the required parameter OBJ.

The following object-related commands operate on many object types.

Object	Object authority	Save/restore	Journal	Other
CHGOBJD CHKOBJ CRTDUPOBJ DLTOBJ DSPOBJDMOVOBJ RNMOBJ RTVOBJD WRKOBJ	CHGOBJAUD CHGOBJOWN CHGOBJPGP DSPOBJAUT EDTOBJAUT GRTOBJAUT RVKOBJAUT	RSTOBJ SAVCHGOBJ SAVOBJ	CHGJRNOBJ ENDJRNOBJ STRJRNOBJ	DMPOBJ DMPSYSOBJ PRTDSKINF WRKOBJLCK

Related concepts

External object types
Many types of external objects are stored in libraries. Some types of external objects can only be stored in the integrated file system in directories.

Related reference

CL command names
The command name identifies the function that will be performed by the program that is called when the command is run. Most command names consist of a combination of a verb (or action) followed by a noun or phrase that identifies the receiver of the action (or object being acted on): (command = verb + object acted on).

TEXT parameter
The TEXT parameter specifies the user-defined description that briefly describes the object that is being created or changed.

CL commands that operate on multiple objects
In addition to the commands that operate on single object types, there are commands that operate on several object types. These commands are more powerful because they can operate on several objects of different types at the same time.

OUTPUT parameter

The OUTPUT parameter specifies whether the output from the display command is displayed, printed, or written to an output file.

Basically, the same information is provided whether the output is displayed, printed, or written; only the format is changed as necessary to present the information in the best format for the device. For example, because there are more lines on a printed page than on a display, column headings are not repeated as often in printed output.

If the output is to be shown on the display, it will be sent to the workstation that issued the display command. It will be shown in the format specified in the display device file used by that display command. A different device file is used for the output of each display command, and the file is different for displayed, printed, or written file output. In most cases, the name of the command is part of the file names of either type of device file.

If the output will be printed, it is spooled and an entry is placed on the job's output queue. The output can be printed depending on which device is specified in the **Start Printer Writer (STRPRTWTR)** command.

Note: Although the IBM -supplied printer files are shipped with SPOOL(*YES) specified, they can be changed to SPOOL(*NO) by the **Override with Printer File (OVRPRTF)** and **Change Printer File (CHGPRTF)** commands.

If the OUTPUT parameter is not specified in the display command, the default value * is assumed. The output resulting from this value depends on the type of job that entered the command. The following table shows how the output is produced for interactive and batch jobs.

Table 11. Output for interactive and batch

Output	Interactive job	Batch job
*	Displayed	Printed
*PRINT	Printed	Printed

Values allowed

*****
Output requested by an interactive job is shown on the display. Output requested by a batch job is printed with the job's spooled output.

***PRINT**
The output is printed with the job's spooled output.

***OUTFILE**
The only output is to be written to a specified database file.

PRTTXT parameter

The print text (PRTTXT) parameter specifies the text that appears at the bottom of listings and on separator pages.

Print text is copied from the job attribute when the job enters the system. Print files that originate on another system do not use the print text on the target system. Print text exists as a job attribute (PRTTXT) for defining the print text of a specific job, and as a system value (QPRTTXT) for the default of jobs with *SYSVAL specified. QPRTTXT is the system-wide default for all jobs.

The print text can be up to 30 characters in length. The text should be centered in the form's width and printed in the overflow area. You should center the required text within the 30 character field.

If the print text is not blank, the system prints 30 characters of text on the bottom of each page. This text normally follows the overflow line and is preceded by a blank line (if the form's length permits). If the user prints past the overflow line, the print text follows the last line of the user text, again preceded by a blank line when possible. If the overflow line is the last line of the form, the print text also prints on the last line of the form, which may result in the typing over of user text.

The print text for job and file separators is put on the first line of the separator page. A job separator contains print text of the job that created the separator at the time the file was printed. A file separator contains the same print text as the spooled file it precedes.

The print text can be specified for all job types. System and subsystem monitor jobs use the system value. Reader and writer jobs use the system value unless print text is changed in the QSPLxxxx job description associated with the reader or writer.

The print text is determined from several places by using the following hierarchical order. If print text is not specified in one place, the next place in the order is used.

The hierarchical order, beginning with the highest priority, is as follows:

- Override print file value
- Print file value

- Job attribute changed by the **Change Job (CHGJOB)** command
- Job attribute set by the **Submit Job (SBMJOB)** or **Batch Job (BCHJOB)** command
- Job description
- System value

Values allowed

For the system value QPRTTXT, any character string can be specified, with the exception of *SYSVAL. If *BLANK is specified, there will be no print text. For PRTTXT, some of the following values can be selected, depending on the command:

***SAME**
The print text does not change.

***CURRENT**
The print text is taken from the submitting job.

***JOBD**
The print text is taken from the job description under which the job is run.

***SYSVAL**
The print text is taken from the system value QPRTTXT.

***BLANK**
There is no text or blanks printed.

'print-text'
Specify 30 characters of text. If there are blanks in the text, then single quotation marks must be used around the entry. The text should be centered within the field for the text to be centered on the page.

REPLACE parameter

The replace (REPLACE) parameter is used on create commands. It specifies that the existing object, if one exists, is replaced by the object of the same name, library, and object type that is being created.

The user of the new object is granted the same authority as for the object being replaced. If the object being replaced is secured by an authorization list, then the new object is secured by the same authorization list. The public authority of the new object is the same as the public authority of the replaced object. The AUT parameter from the create command is ignored. All private authorities from the replaced object are copied to the new object. The owner of the new object is not copied from the replaced object. The owner of the new object is the creator of the new object or the creator's group profile. Some objects such as panel groups, display files, and menus cannot be replaced if they are in use by the current job or another job.

If the object being created is a program or service program, then the user profile (USRPRF parameter) value from the replaced program is used. The user profile (USRPRF parameter) value from the **Create Program (CRTPGM)** or **Create Service Program (CRTSRVPGM)** command is ignored. If the value of the user profile (USRPRF parameter) of the program or service program being replaced is *OWNER, then only the current owner of the program or service program being replaced can create the new program or service program that replaces the existing program or service program. If the owner of the existing object and the object being created do not match, the object is not created and message CPF2146 is sent.

If the object being created is a program or service program, then the use adopted authority (USEADPAUT) value from the replaced program or service program is used as long as the user creating the object can create programs/service programs with the USEADPAUT(*YES) attribute. The QUSEADPAUT system value determines whether users can create programs or service programs to use adopted authority. For example, if the existing object being replaced has USEADPAUT(*YES) and you do not have authority to create a program or service program that uses adopted authority, the program or service program created will have USEADPAUT(*NO). In this case, the USEADPAUT value was not copied. If you have authority to create programs or service programs that use adopted authority, the created program or service program will have the same USEADPAUT value as the program or service program being replaced. An informational message is sent which indicates whether the USEADPAUT value was copied to the object being replaced.

If the object being created is a file, and the default, or *YES, is specified on the REPLACE parameter, an existing device file other than save file and a DDM file with the same qualified name will be replaced by the new file. For example, an existing display file can be replaced by a new printer file, tape file, and so on.

Object management (*OBJMGT), object existence (*OBJEXIST), and read (*READ) authorities are required for the existing object to allow replacement of the existing object with a new object.

The existing object is renamed and moved to library QRPLOBJ or library QRPLxxxxx if the object resides on an Independent ASP (where 'xxxxx' is the number of the primary ASP of the ASP group) when the creation of the new object is successful. The replaced object is renamed with a Q appended to a time stamp and moved to library QRPLOBJ or library QRPLxxxxx if the object resides on an Independent ASP. If the existing object could not be moved to the QRPLOBJ library because QRPLOBJ was locked by another job, for example, the existing object will be moved to the QTEMP library for the job. If the existing object could not be moved to the QTEMP library for the job, the existing object will be deleted. If the existing object could not be moved to the QRPLxxxxx library because QRPLxxxxx was locked by another job, for example, the existing object will be deleted.

Restriction

Programs can be replaced while they are being run; however, if the replaced program refers to the program message queue after the renaming of the replaced program to the Qtimestamp name, the program fails and an error message is sent stating that the program message queue is not found.

A database file, physical or logical, and a save file cannot be replaced by any file.

Library QRPLOBJ is cleared when an initial program load (IPL) of the system is done. Library QRPLxxxxx is cleared when the primary ASP of the ASP group is varied on.

Values allowed

***YES**
The system replaces the existing object with the new object being created that has the same name, library, and object type.

***NO**
The system does not replace the existing object that has the same name, library, and object type with the object being created.

JOBPTY, OUTPTY, and PTYLMT scheduling priority parameters

The scheduling priority parameters specify the priority values used by the system to determine the order in which the jobs and spooled files are selected for processing.

Each job is given a scheduling priority that is used for both job selection and spooled file output. The job scheduling priority is specified by the JOBPTY parameter in commands like the **Batch Job (BCHJOB)**, **Submit Job (SBMJOB)**, **Create Job Description (CRTJOBD)**, and **Change Job Description (CHGJOBD)** commands. The priority for producing the spooled output from a job is specified by the OUTPTY parameter in the same commands.

In addition, because every job is processed under a specific user profile, the priority for jobs can be limited by the PTYLMT parameter specified on the **Create User Profile (CRTUSRPRF)** and **Change User Profile (CHGUSRPRF)** commands. This parameter value controls the maximum job scheduling priority and output priority that any job running under a user profile can have; that is, the priority specified in the JOBPTY and OUTPTY parameters of any job command cannot exceed the priority specified in the PTYLMT parameter for that user profile. The scheduling priority is used to determine the order in which jobs are selected for processing and is not related to the process priority specified in the class object.

The three scheduling priority parameters specify the following:

- The PTYLMT parameter specifies the highest scheduling priority for any job that you submit. In the commands that affect the user profile, the PTYLMT parameter specifies the highest priority that can be specified in another JOBPTY parameter on commands relating to each specific job. You can specify a lower priority for a job on the command used to submit the job. If you specify a higher priority for

JOBPTY in the BCHJOB or SBMJOB command than is specified for PTYLMT in the associated user profile, an error message is shown on the display and the maximum priority specified in PTYLMT is assumed. If a higher job priority is specified in the CHGJOB or CHGJOBD command, an error message is shown and the attributes are not changed.

- The JOBPTY parameter specifies the priority value to be used for a specific job being submitted. In the commands relating to a specific job being submitted, the JOBPTY parameter specifies the actual scheduling priority for the job.
- The OUTPTY parameter specifies the priority for producing the output from all spooled output files from the job. The priority value specified in the OUTPTY parameter determines the order in which spooled files are handled for output. The same value is applied to all the spooled files produced by the job.

The scheduling priority can have a value ranging from 0 through 9, where 1 is the highest priority and 9 is the lowest priority. Any job with a priority of 0 is scheduled for processing before all other jobs that are waiting and that have priorities of 1 through 9.

The priority parameters can be specified on the following commands.

Table 12. Priority parameter

Priority parameter	Commands on which it can be specified
JOBPTY	ADDJOBJS, BCHJOB, CHGJOB, CHGJOBD, CHGJOBJS, CRTJOBD, SBMJOB, SBMJOBJS
OUTPTY	ADDJOBJS, BCHJOB, CHGDKTF, CHGJOBD, CHGJOBJS, CHGJOB, CHGJOBD, CHGJOBJS, CHGPJ, CHGPRTF, CHGSPLFA, CRTDKTF, CRTJOBD, CRTPRTF, OVRDKTF, OVRPRTF, SBMJOB, SBMJOBJS
PTYLMT	CHGUSRPRF, CRTUSRPRF, RTVUSRPRF

Values allowed

Depending upon the command, one or more of the following values apply to the parameter.

5
If a value is not specified in the CRTUSRPRF command, five is the default value that is assumed for the priority limit for the user profile. That would be the highest priority that the user could specify for any job he submits for processing. If not specified in the CRTJOBD command, five is the default value for both the job scheduling priority and the output priority.

***SAME**
The priority assigned, or the highest priority that can be assigned, does not change.

***JOBD**
The scheduling priority for the job is obtained from the job description under which the job runs.

scheduling-priority:
Specify a priority value ranging from 0 through 9, where 0 is the highest priority and 9 is the lowest priority. Priority 0 is allowed only on CHGJOB.

SEV parameter

The severity (SEV) parameter specifies a severity code.

The severity (SEV) parameter specifies the severity code that:

- Describes the level of severity associated with an error message.
- Indicates the minimum severity level that causes a message to be returned to a user or program.
- Causes a batch job to end.
- Causes processing of a command to end if a syntax error of sufficient severity occurs.

Note: The LOG parameter on some commands also uses these severity codes for logging purposes (to control which job activity messages and error messages are logged in the job log).

The severity code is a 2-digit number that can range from 00 through 99. The higher the value, the more severe or important the condition. The severity code of a message that is sent to a user indicates the severity of the condition described by the message. More than one message can have the same severity code. If a severity code is not specified for a predefined message, it is assumed to be 00 (information only).

You can specify a severity code for any message when it is defined by the **Add Message Description (ADDMSGD)** command. To change the severity code of a message, use the **Change Message Description (CHGMSGD)** command.

IBM -defined severity codes are used in all of the IBM -supplied messages that are shipped with the system.

00 - Information:
A message of this severity is 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 it has reached a successful completion.

10 - Warning:
A message of this severity indicates a potential error condition. The program may have taken a default, such as supplying missing input. The results of the operation are assumed to be what was intended.

20 - Error:
An error has been detected, but it is one for which automatic recovery procedures probably were applied, and processing has continued. A default may have been taken to replace input that was in error. The results of the operation may not be valid. The function may be only partially complete; for example, some items in a list may be processed correctly while others may fail.

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 program processing, it leads to an abnormal end of the program (severity 40). The results of the operation are not valid.

40 - Abnormal End of Program or Function:
The operation has ended, possibly because it was unable to handle invalid data, or possibly because the user ended 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 ended.

60 - System Status:
A message of this severity is issued only to the system operator. It gives either the status of or a warning about a device, a subsystem, or the whole system.

70 - Device Integrity:
A message of this severity is issued only to the system operator. It indicates that a device is malfunctioning or in some way is no longer operational. You may be able to restore system operation, or the assistance of a service representative may be required.

80 - System Alert:
A message of this severity is issued only to the system operator. It 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:
A message of this severity is issued only to the system operator. It describes a condition that renders either a subsystem or the whole system inoperative.

99 - Action:
A message of this severity indicates that some manual action is required, such as specifying a reply or changing printer forms.

Related tasks

Assigning a severity code

The severity code you assign to a message on the **Add Message Description (ADDMSGD)** command indicates how important the message is.

SPLNBR parameter

The spooled file number (SPLNBR) parameter is used when more than one spooled file is created by a job and the files all have the same name. The files are numbered, starting with 1, in the order that they are opened by the job.

The job log is always the last file for a job.

A file number is generated for each file when it is opened within a job (when output records are produced) and it is used by the system as long as the job, the files, or both are on the system. If the files are not uniquely named because they were opened more than once, this file number is used to specify which file (or group of records, if the complete file has not yet been produced) is acted upon by a CL command.

TEXT parameter

The TEXT parameter specifies the user-defined description that briefly describes the object that is being created or changed.

The description can include up to 50 characters; if it is a quoted string (that is, enclosed in single quotation marks), any of the 256 EBCDIC characters can be used. The single quotation marks are not required if the string does not contain any blanks or other special characters. Any of the 50 character positions not filled by the specified description are padded with blanks.

The description is used to describe any of the IBM i objects when the named object is shown on the display by the **Display Object Description (DSPOBJD)** command. Only objects for which object operational authority has been obtained can be displayed by a user.

For commands that use a database source file to create some type of object, you can (by default) use the text from the source file member as the text for the newly-created object. For example, if you use the **Create Control Language Program (CRTCLPGM)** command to create a CL program, but you do not specify a description in the TEXT parameter, the text specified for the source file member (SRCMBR parameter) of the source file (SRCFILE parameter) is assumed as the descriptive text for the CL program.

Values allowed

Depending upon the command, one or more of the following values apply to the TEXT parameter.

***SRCMBRTXT**
: For commands that create objects based on database source files only, the text is taken from the source member. If a device or an inline file is used for source input or if source is not used, the text is left blank.

***BLANK**
: The user description of the object being created or changed is left blank.

***SAME**
: The user-defined description does not change.

'description'
: Specify the description of the object being created or changed. Up to 50 characters enclosed in single quotation marks (required for blanks and other special characters) can be specified to describe the object. If a single quotation mark is one of the 50 characters, two single quotation marks ('') must be used instead of one to represent the single quotation mark character.

Related reference

OBJTYPE parameter

The object type (OBJTYPE) parameter specifies the types of IBM i objects that can be operated on by the command in which they are specified.

VOL parameter

The volume (VOL) parameter specifies the volume identifiers of the volumes used in a tape or optical operation.

A tape volume consists of a tape cartridge or reel. An optical volume consists of a single side of an optical cartridge or a single CD-ROM. Optical cartridges are dual-sided and each side is a separate volume.

The volume identifier is the identifier stored on each tape or optical disk (in the volume label area) that it identifies. An inquiry message is sent to the system operator if a volume identifier is missing or out of order.

Tape volumes must be on the tape units in the same order as their identifiers are specified in the VOL parameter and as the device names are specified in the DEV parameter of the tape device file commands. However, if the tapes are read backward (a function supported in COBOL), the volumes must be in reverse order to that specified in the VOL parameter. Nevertheless, the device names are still specified in forward order in the DEV parameter.

In general, the rule for specifying tape volume identifiers is that as many as 6 characters, containing any combination of letters and digits, can be used. Special characters can be used if the identifier is enclosed in single quotation marks. However, if the tape is used on an operating system other than the IBM i operating system, the requirements for specifying identifiers must be considered.

Optical volume identifiers can be up to 32 characters long and can contain any combination of digits and uppercase letters. Each optical volume identifier must be unique. No two optical volumes with the same identifier can be present on the system at the same time.

For labeled tapes, the following rules apply:

- Characters: A maximum of 6 characters, or fewer, can be specified for each volume identifier. Alphabetic and numeric characters can be used in any order.
- Uniqueness: More than one volume can have the same identifier. You may have a file using the same identifier for several volumes; in this case, the system keeps track of the order internally with a sequence number written on the volumes. However, volume identifiers should be unique whenever possible.
- Order: When multiple volumes (with different identifiers) are used in a single operation, they must be in the same order as the volume identifiers specified in the VOL parameter.

Multivolume files

If multiple volumes (tapes) are used in an operation and all have the same volume identifier, that identifier must be specified in the VOL parameter once for each volume used. For example, if three tapes named QGPL are used in a save operation, VOL(QGPL QGPL QGPL) must be specified.

When a multivolume file on *tape* is processed and multiple tape units are used, the tape volumes must be placed in the tape devices in the same order as they are specified in the VOL parameter. For example, if five volumes and three tape units are used, they are mounted as follows: VOL1 on unit 1, VOL2 on unit 2, VOL3 on unit 3, VOL4 on unit 1, and VOL5 on unit 2.

Values allowed

***MOUNTED**
The volume currently placed in the device is used.

***NONE**
No volume identifier is specified.

***SAME**
Previously specified volume identification does not change.

***SAVVOL**
The system, using the save/restore history information, determines which tape volumes contain the most recently saved version. If the device specified in the DEV parameter of the restore command does not match the device of the most recently saved version of the object, an error message is returned to the user, and the function is ended. If the wrong volume is mounted in the unit specified by the command, a message is returned to the system operator that identifies the first volume that must be placed in the device before the restore operation can begin.

volume-identifier
Specify the identifiers of one or more volumes in the order in which they are put on the device and used. Each tape volume identifier contains a maximum of 6 alphanumeric characters. Each optical volume identifier contains a maximum of 32 characters. A blank is used as a separator character when listing multiple identifiers.

Related information

Optical device programming

WAITFILE parameter

The WAITFILE parameter allows you to specify whether and how long a program waits for resources.

You use the WAITFILE parameter to specify the maximum number of seconds that a program waits for the following:

- File resources to be allocated when the file is opened
- Session resources when the evoke function is issued for an APPC device
- The device to be allocated when an acquire operation is performed to read the file

If the program must wait, it will be placed in a wait state until the resources are available or until the wait time expires. If two or more file resources are needed and are not available because they are being used by different system users, the acquisition of each resource might require a wait. This maximum is applied to each wait.

The length of the wait can be specified in this parameter, or the default wait time of the class that applies to the object can be used. If the file resources cannot be allocated in the specified number of seconds, an error message is returned to the program.

The file resources that must be allocated depend on the type of file being opened. File resources consist of the following.

- For device files that are not spooled (SPOOL(*NO)), the file resources include the file description and device description. Because the device description must be allocated, the device itself must also be available.
- For device files that are spooled (SPOOL(*YES)), the file resources include the file description, the specified output queue, and storage in the system for the spooled data. Because the data is spooled, the device description (and thus the device itself) need not be available.
- For database files, the file resources consist of the file and member data. The file's associated member paths are not accessed, and therefore, the system does not wait for them. A file open exception error can occur before the WAITFILE time has expired when an access path is not available (for example, when the access path is being rebuilt).

  The **`Allocate Object (ALCOBJ)`** command can be used to allocate specific file resources before the file is opened.

The session resources that were allocated for an APPC device conversation can be lost between the time the application issues a detach function or receives a detach indication and the time another evoke function is issued. If the session resource is lost, this parameter is used to determine the length of time that the system waits for another session resource.

Values allowed

***IMMED**
The program does not wait; when the file is opened, an immediate allocation of the file resources is required.

***CLS**
The default wait time specified in the class description is used as the wait time for the file resources to be allocated.

1-32767
Specify the number of seconds that the program waits for the file resources to be allocated.

Parameter values used for testing and debugging

The IBM i operating system includes functions that let a programmer observe operations performed as a program runs.

Testing functions can be used to locate operations that are not performing as intended. These 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 CL 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 IBM i testing functions let you:

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

This section contains expanded descriptions of the program variable, basing pointer, subscript, and qualified-name parameter values. These values can be specified on the **Add Breakpoint (ADDBKP)**, **Add Trace (ADDTRC)**, **Change High-Level Language Pointer (CHGHLLPTR)**, **Change Program Variable (CHGPGMVAR)**, and **Display Program Variable (DSPPGMVAR)** commands.

See the appropriate Integrated Language Environment (ILE) guide for debugging information with other ILE languages.

Related tasks

Debugging ILE programs
To debug your Integrated Language Environment (ILE) programs, use the ILE source debugger.

Debugging original program model programs
To debug your original program model (OPM) programs, use testing functions. These functions are available through a set of commands that can be used interactively or in a batch job.

Program-variable description

The program variable must be enclosed in single quotation marks if it contains special characters. Up to 132 characters can be specified for a program variable name. This includes any subscripts, embedded blanks, parentheses, and commas. It does not include the enclosing single quotation marks when special characters are used.

```
Program Variable

                        .-------------------.
                        V                   |
```

```
>>-qualified-name---subscript------,subscript-------+----------><
```

Note: The maximum is 14 repetitions.

Some examples follow:

```
        COUNTA
        'VAR1(2,3)'
        'A.VAR1(I,3,A.J,1)'
        'VAR1 OF A(I,3,J OF A)'
        '&LIBNAME'
```

Basing-pointer description

The basing pointer must be enclosed in single quotation marks if it contains special characters. Up to 132 characters can be specified for a basing pointer name. This includes any subscripts, embedded blanks, parentheses, and commas. It does not include the enclosing single quotation marks when special characters are used.

```
Basing Pointer

                        .-------------------.
                        V                   |
>>-qualified-name---subscript------,subscript-------+----------><
```

Note: The maximum is 14 repetitions.

Some examples follow:

```
        PTRVAR1
        'ABC.PGMPTR(5,B.I)'
```

If more than one basing pointer is specified for a variable, the list of basing pointers must be enclosed in parentheses. When multiple basing pointers are specified, they must be listed in order, from the first basing pointer to the last, when used to locate the variable. In the following example, the PTR_1 basing pointer is the first basing pointer used to locate the variable; it either must have a declared basing pointer, or it must not be a based variable. The address contained in the PTR_1 pointer is used to locate the A.PTR_2 pointer (which must be declared as a based pointer variable). The contents of the A.PTR_2 pointer are used to locate the PTR_3 pointer array (which must also be declared based), and the contents of the specified element in the last pointer array are used to locate the variable. An example is:

```
        ('PTR_1' 'A.PTR_2' 'PTR_3(1,B.J)')
```

Subscript description

An integer number contains from 1 through 15 digits with an optional leading sign (either plus or minus). A decimal point is not allowed in an integer-number subscript. If a decimal point is specified, the subscript value is not interpreted as the correct numeric value by the system, and an error message is returned.

```
Subscript

   .-integer-number--.
>>-+-qualified-name--+----------------------------------------><
   '-*---------------'
```

An asterisk (*) can be used to request a single-dimensional cross-section display of an array program variable. An asterisk can only be specified for a subscript on the primary variable (not on a basing pointer) for the PGMVAR keyword on the Add Break Point (ADDBKP), Add Trace (ADDTRC), and Display Program

Variable (DSPPGMVAR) commands. In addition, if the variable has multiple dimensions, only one of the subscript values can be an asterisk. An example of a request to display an array cross-section is as follows:

```
        DSPPGMVAR  PGMVAR('X1(*,5,4)')
```

This display shows the values of all elements of the array that have the second subscript equal to five, and the third subscript equal to four.

Qualified-name description

Some high-level languages might allow you to declare more than one variable with the same fully qualified name (although you generally are not able to refer to these variables in the high-level language program after they are declared). If you attempt to refer to such a variable using an IBM i test facility command, the system selects one of the variables and uses it for the operation. No error is reported when a duplicate fully qualified name is selected.

```
Qualified-Name

>>--+-/ODV-number-----------------------------+---------------><
    |             .-------------------------.   |
    |             V                         |   |
    '-variable-+----+-OF-+--variable-------+-+-'
               |    '-IN-'                  |
               |  .---------------.         |
               |  V               |         |
               '---variable-------+---------'
```

Note: The maximum is 19 repetitions.

Rules for qualified name description

- An ODV number is a slash (/) followed by 1 to 4 hexadecimal digits (0 through 9, and A through F).
- The variable-name must be the name of a variable in the program. This name must be specified the same way in the high-level language. Some high-level languages introduce qualifier variable names in addition to the ones you specified in the source for your program. See the appropriate high-level language manual for more information about variable names.
- Blanks must separate the variable-names from the special words OF and IN.
- When a period is used to form a qualified name, no blanks can appear between it and the variable-names.
  - The ordering of the variable names must follow these rules:
    - For qualified names that contain no embedded period, the variable names are assumed to be specified from the lowest to the highest levels in the structure.
    - For qualified names that contain one or more embedded periods, the variable names are assumed to be specified from the highest to the lowest levels in the structure.
  - When an ODV number is not used for the qualified name, enough qualifier variable names must be specified so that a single unique variable can be identified in the program. Whether the qualified name is a simple name (only one variable name specified) or a name with multiple qualifier variable names, the variable in the program is uniquely identified if either of the following conditions is true (these conditions may require you to specify more qualifier variable names for IBM i test facility commands than you need to specify in the high-level language program to uniquely select a program variable):
    - A variable is uniquely identified if there is one and only one variable in the program with a set of qualifier variables matching the qualified variable name specified.
    - A variable is uniquely identified in the program if it has exactly the same set of qualifier variables as the qualifier variable names specified. When the complete set of qualifiers is specified, the variable

name is said to be *fully qualified*. A variable that is a *fully qualified* match for the qualified-name is selected even if there are other variables with names that match the qualified name but have additional qualifier variables which were not specified.

Control language elements

Control language (CL) contains elements such as character sets and values, expressions, built-in functions, and naming within commands.

CL character sets and values

The EBCDIC character sets, special characters, and IBM-defined fixed values called *predefined values* can be used in CL.

Character sets

Control language (CL) uses the extended binary-coded decimal interchange code (EBCDIC) character set.

For convenience in describing the relationship between characters used in the control language and the EBCDIC character set, the following control language categories contain the EBCDIC characters shown.

Category	Characters included
Alphabetic [1]	26 letters (A through Z), $, #, and @[3]
Numeric	10 digits (0-9)
Alphanumeric [1,2]	A through Z, 0 through 9, $, #, @[3], period (.), and underscore (_)
Special Characters	All other EBCDIC characters

Notes:

1. Lowercase letters (a through z) are accepted, but are generally translated into the corresponding uppercase letters by the system. No translation of lowercase letters is done for letters included within a quoted character string or a comment. No translation is done for lowercase letters specified for a value on a parameter that has the character (*CHAR) or the path name (*PNAME) attribute for its TYPE parameter and the mixed case (*MIXED) attribute for its CASE parameter in the command definition.
2. The underscore (_) is an alphanumeric connector that can be used in IBM i CL to connect words or alphanumeric characters to form a name (for example, PAYLIB_01). This use of the underscore might not be valid in other high-level languages.
3. The $, #, and @ extended alphabetic characters are variant characters across the EBCDIC CCSIDs. CL assumes CCSID 37 when processing CL commands so $ is assumed to be X'5B', # is assumed to be X'7B' and @ is assumed to be X'7C'. If your EBCDIC job CCSID is not 37, you would have to use the character which maps to one of these three hexadecimal code points.

The first three categories contain the characters that are allowed in quoted and unquoted character strings, in comments, and in CL names, such as in names of commands, labels, keywords, variables, and IBM i objects. Special characters in the last category can only be used in quoted character strings and comments; they cannot be used in unquoted strings. However, some have special syntactical uses when coded in the proper place in CL commands.

Related concepts

Special character use
Special characters can be used only in these special ways or inside quoted character strings or comments.

Double-byte character text in CL commands
You can use double-byte character data anywhere in a CL command that descriptive text can be used.

Enter double-byte character text as follows:

1. Begin the double-byte character text with a single quotation mark (').
2. Enter a shift-out character.
3. Enter the double-byte character text.
4. Enter a shift-in character.
5. End the double-byte character text with a single quotation mark (').

For example, to enter the double-byte character literal ABC, enter the following, where represents the shift-out character and represents the shift-in character:

' ABC '

Limit the length of a double-byte character text description of an object to 14 double-byte characters, plus the shift control characters, to make sure that the description is properly displayed and printed.

Special character use

Special characters can be used only in these special ways or inside quoted character strings or comments.

The special EBCDIC characters are used by the CL in various ways. They are most frequently used as delimeter characters and as symbolic operators in expressions. The special characters, as shown in the following table, have assigned meanings when coded in control language commands.

Table 13. Delimiters

Name	**Symbol**	**Meanings**
Single quotation mark	' '	A single quotation mark delimiters indicate the beginning and end of a quoted character string (a constant).
Begin and end comment	/* */	Indicates the beginning and end of a comment.
Blank	b [1]	Basic delimiter for separating parts of a command (label, command name, and its parameters), and for separating values inside lists.
Colon	:	Ending delimiter for command labels. Separates parts of time values. [3]
Comma	,	In many countries, used as decimal point in numeric values. Separates parts of date values. [2]
Left and right parentheses	( )	Grouping delimiter for lists and parameter values, and for evaluating the order of expressions.
Period	.	Decimal point. Used to separate the name and extension of a document and folder name and to separate the parts of date values. [2]
Quote	" "	Start of a quoted object name.
Slash	/	Connects parts of qualified names or path names.
Slashes	//	Identifying characters used in positions 1 and 2 of BCHJOB, ENDBCHJOB, and DATA commands in the job stream. Also, used as a default delimiter on inline data files.

Table 13. Delimiters (continued)		
Name	**Symbol**	**Meanings**

Notes:

1. Because this character does not resolve in the online version of this book, [b] is used to represent a blank space only when the character cannot be clearly explained in another manner.
2. Valid only when the job date separator value specifies the same character.
3. Valid only when the job time separator value specifies the same character.

Related concepts

Character sets
Control language (CL) uses the extended binary-coded decimal interchange code (EBCDIC) character set.

Symbolic operators
A variety of characters can be used as symbolic operators in CL commands.

Related reference

Operators in expressions
Operators are used in expressions to indicate an action to be performed on the operands in the expression or the relationship between the operands.

Symbolic operators

A variety of characters can be used as symbolic operators in CL commands.

The following characters are used as symbolic operators in CL commands.

Table 14. Symbolic operators in CL commands		
Name	**Symbol**	**Meanings**
And	&	Symbolic logical operator for AND.
Asterisk	*	Multiplication operator. It indicates a generic name when it is the last character in the name. It indicates IBM i reserved values (predefined parameter values and expression operators) when it is the first character in a string.
Concatenation	|>, |<, and || [3]	Character string operator (indicates both values are to be joined).
Equal	=	Symbolic *equal* relational operator.
Greater than	>	Symbolic *greater than* relational operator.
Less than	<	Symbolic *less than* relational operator.
Minus (hyphen)	-	Subtraction operator, command continuation operator, and negative signed value indicator. Separates parts of date values. [1]
Not	¬ [2]	Symbolic NOT relational operator.
Or	| [3]	Symbolic logical operator for OR.
Plus	+	Addition operator, command continuation character, and positive signed value indicator.
Slash	/	Division operator. Separates parts of date values. [1] Used as the separator between parts of a qualified name.

Table 14. Symbolic operators in CL commands (continued)		
Name	**Symbol**	**Meanings**
Notes: 1. Valid only when the job date separator value specifies the same character. 2. In some character sets, including the multinational character set, the character ^ replaces the ¬ character. Either ^ or *NOT can be used as the logical NOT operator in those character sets. 3. In some character sets, including the multinational character set, the character ! replaces the | character. Either ! or *OR can be used as the logical OR operator, and either || or *CAT can be used as the concatenation operator in those character sets.		

Note: The symbolic operators can also be used in combinations.

Symbolic operators can also be used in the following ways.

Table 15. Symbolic operators: Other uses		
Name	**Symbol**	**Meanings**
Ampersand	&	It identifies a CL variable name when it is the first character in the string.
Percent	%	It identifies a built-in system function when it is the first character in the string.
Question mark	?	It specifies a prompt request when it precedes a command name or keyword name.

Related concepts

Special character use
Special characters can be used only in these special ways or inside quoted character strings or comments.

Related reference

Expressions
An *expression* is a group of constants, variables, or built-in functions, separated by operators, that produces a single value.

Operators in expressions
Operators are used in expressions to indicate an action to be performed on the operands in the expression or the relationship between the operands.

Predefined values

Predefined values are IBM-defined fixed values that have predefined uses in control language (CL) and are considered to be reserved in the IBM i operating system.

Predefined values have an asterisk (*) as the first character in the value followed by a word or abbreviation, such as *ALL or *PGM. The purpose of the * in predefined values is to prevent possible conflicts with user-specified values, such as object names. Each predefined value has a specific use in one or more command parameters, and each is described in detail in the command description.

Some predefined values are used as operators in expressions, such as *EQ and *AND. The predefined value *N is used to specify a null value and can be used to represent any optional parameter. A *null value* (*N) indicates a parameter position for which no value is being specified; it allows other parameters that follow it to be entered in positional form. To specify the characters *N as a character value (not as a null), the string must be enclosed in single quotation marks ('*N') to be passed. Also, when the value *N appears in a CL program variable at the time it is run, it is always treated as a null value.

Expressions in CL commands

A character string expression can be used for any parameter, element, or qualifier that is defined with EXPR(*YES) in the command definition object.

Any expression can be used as a single parameter in the **Change Variable (CHGVAR)** and **If (IF)** commands. An expression in its simple form is a single constant, a variable, or a built-in function. An expression typically contains two operands and an operator that indicates how the expression is to be evaluated. Two or more expressions can be combined to make a complex expression.

The following types of expressions are supported in CL programs:

- Arithmetic (`&VAR + 15`)
- Character string (`SIX || TEEN`)
- Relational (`&VAR > 15`)
- Logical (`&VAR & &TEST`)

A complex expression contains multiple operands, operators that indicate what operation is performed on the operands, and parentheses to group them. Only one operator is allowed between operands, except for the + and - signs when they immediately precede a decimal value (as a signed value), and the *NOT operator when it is used in a logical expression.No complex expression can have more than five nested levels of parentheses, including the outermost (required) level.

Arithmetic and character string expressions can be used together in a complex expression if they are used with relational and logical operators; for example: `(A=B&(1+2)=3)`. A pair of arithmetic expressions or a pair of character string expressions can be compared within a relational expression. Also, relational expressions can be used within a logical expression

Related reference

Expressions

An *expression* is a group of constants, variables, or built-in functions, separated by operators, that produces a single value.

Arithmetic expressions

The operands in an arithmetic expression must be decimal constants, decimal CL variables, integer CL variables, or CL built-in functions that returns a numeric result. These CL built-in functions include %BINARY, %CHECK, %CHECKR, %SCAN, %DEC, %INT, %UINT, %LEN, %SIZE and %PARMS.

An arithmetic operator (only in symbolic form) must be between the operands. The results of all arithmetic expressions are decimal values, which may be stored in a CL variable.

Note: The division operator (/) must be preceded by a blank if the operand that precedes it is a variable name. (For example, &A /2, not &A/2.) All other arithmetic operators may optionally be preceded or followed by a blank.

Arithmetic operands can be signed or unsigned; that is, each operand (whether it is a numeric constant or a decimal or integer CL variable) can be immediately preceded by a plus (+) or minus (-) sign, but a sign is not required. When used as a sign, no blanks can occur between the + or - and its value. For example, a decimal constant of 23.7 can be expressed as +23.7 or -23.7 (signed) or as 23.7 (unsigned).

The following are examples of arithmetic expressions:

```
(&A + 1)    (&A + &B -15)
(&A - &F)   (&A+&B-15)
(&A + (-&B))
(%SCAN('/' &STRING) + 1)
(%DEC(&STRING) - 1)
(%SIZE(&CHAR1) + %SIZE(&CHAR2))
(%PARMS() - 1)
```

If the last nonblank character on a line is a plus (+) or minus (-) sign, it is treated as a continuation character and not as an arithmetic operator.

Character string expressions

The operands in a character string expression must be quoted or unquoted character strings, character CL variables, the substring (%SUBSTRING or %SST) built-in function, or a trim (%TRIM, %TRIML, or %TRIMR) built-in function, or a conversion (%CHAR, %LOWER, or %UPPER) built-in function.

The value that is associated with each variable or built-in function must be a character string. The result of concatenation is a character string.

There are three operators that can be used in character string expressions. These operators concatenate (or join) two character strings, but each has a slightly different function. They are:

***CAT (concatenation, symbol ||) operator**

The *CAT operator concatenates two character strings. For example: `ABC *CAT DEF` becomes `ABCDEF`

Blanks are included in the concatenation. For example: `'ABC ' *CAT 'DEF '` becomes `'ABC DEF '`

***BCAT (concatenation with blank insertion, symbol |>) operator**

The *BCAT operator truncates all trailing blanks in the first character string; one blank is inserted, then the two character strings are concatenated. Leading blanks on the second operand are not truncated. For example:

`ABC *BCAT DEF` becomes `ABC DEF`
`'ABC ' *BCAT DEF` becomes `'ABC DEF'`

***TCAT (concatenation with trailing blank truncation, symbol |<) operator**

The *TCAT operator truncates all trailing blanks in the first character string, then the two character strings are concatenated. All leading blanks on the second operand are not truncated. For example:

`ABC *TCAT DEF` becomes `ABCDEF`
`'ABC ' *TCAT DEF` becomes `'ABCDEF'`
`ABC *TCAT ' DEF'` becomes `'ABC DEF'`
`'ABC '*TCAT ' DEF'` becomes `'ABC DEF'`

All blanks that surround the concatenation operator are ignored, but at least one blank must be on each side of the reserved value operator (*CAT, *BCAT, or *TCAT). If multiple blanks are wanted in the expression, a quoted character string (a character string that is enclosed within single quotation marks) must be used.

Related reference

Character strings
A *character string* is a string of any EBCDIC characters (alphanumeric and special) that are used as a value.

Example: Character string expressions
This example shows the variables and their corresponding character string expressions.

Assume the following variables.

Variable	Value
&AA	`'GOOD '`
&BB	`'REPLACEMENT'`
&CC	`'ALSO GOOD'`
&DD	`'METHOD'`
&EE	`'nice'`

The following table shows character string expressions and the result.

Expression	Result
(&AA || &BB)	GOOD REPLACEMENT
(&AA||&BB)	GOOD REPLACEMENT
(&AA *CAT &BB)	GOOD REPLACEMENT
(&CC |> &DD)	ALSO GOOD METHOD
(&CC *BCAT &DD)	ALSO GOOD METHOD
(A *CAT MOUSE)	AMOUSE
('A ' *CAT MOUSE)	A MOUSE
(FAST *CAT MOUSE)	FASTMOUSE
('FAST ' *BCAT MOUSE)	FAST MOUSE
('FAST ' *TCAT MOUSE)	FASTMOUSE
('AB' *CAT 'CD')	ABCD
('AB' *BCAT 'CD')	AB CD
('AB' *TCAT 'CD')	ABCD
(%SST(&AA 1 5) *CAT (%SST(&BB 3 5))	GOOD PLACE
(%SST(&CC 1 9) *BCAT (%SST(&BB 3 5))	ALSO GOOD PLACE
(&AA *CAT ' TIME')	GOOD TIME
(&CC *BCAT TIME)	ALSO GOOD TIME
(%TRIM(&CC 'ALOS ') *BCAT %TRIML(&BB 'ER'))	GOOD PLACEMENT
(%UPPER(&EE) *BCAT %LOWER(&BB))	NICE replacement

Example: Using character strings and variables
This example shows how several character variables and character strings can be concatenated to produce a message for a workstation operator.

The example assumes that the variables &DAYS and &CUSNUM were declared as character variables, not decimal variables.

```
DCL     VAR(&MSG)TYPE(*CHAR)     LEN(100)
         *
         *
CHGVAR      &MSG   ('Customer' *BCAT &CUSNAMD  +
              *BCAT 'Account Number' *BCAT +
              &CUSNUM *BCAT 'is overdue by' +
              *BCAT &DAYS *BCAT 'days.')
```

After the appropriate variables have been substituted, the resulting message might be:

```
Customer ABC COMPANY Account Number 12345
is overdue by 4 days.
```

If the variables &DAYS and &CUSNUM had been declared as decimal variables, two other CHGVAR commands would have to change the decimal variables to character variables before the concatenation could be performed. If, for example, two character variables named &DAYSALPH and &CUSNUMALPH were also declared in the program, the CHGVAR commands would be:

```
CHGVAR  &DAYSALPH  &DAYS

CHGVAR  &CUSNUMALPH  &CUSNUM
```

Then instead of &DAYS and &CUSNUM, the new variables &DAYSALPH and &CUSNUMALPH would be specified in the CHGVAR command used to concatenate all the variables and character strings for &MSG.

Or the built-in function %CHAR can be used to convert &DAYS and &CUSNUM into character format in the concatenation expression as follows:

```
CHGVAR      &MSG   ('Customer' *BCAT &CUSNAMD  +
                     *BCAT 'Account Number' *BCAT +
                     %CHAR(&CUSNUM) *BCAT 'is overdue by' +
                     *BCAT %CHAR(&DAYS) *BCAT 'days.')
```

Relational expressions

The operands in a relational expression can be arithmetic or character string expressions. They can also be logical constants and logical variables.

Only two operands can be used with each relational operator. The data type (arithmetic, character string, or logical) must be the same for the pair of operands. The result of a relational expression is a logical value '0' or '1'.

Refer to the table in Operators in expressions for the meanings of the relational operators, which can be specified by symbols (=, >, <, >=, <=, ¬=, ¬>, ¬<) or their reserved values (*EQ, *GT, *LT, *GE, *LE, *NE, *NG, *NL).

If an operation involves character fields of unequal length, the shorter field is extended by blanks added to the right.

Arithmetic fields are compared algebraically; character fields are compared according to the EBCDIC collating sequence.

When logical fields are compared, a logical one ('1') is greater than logical zero ('0'). Symbolically, this is ('1' > '0').

The following are examples of relational expressions:

```
(&X *GT 25)
(&X > 25)
(&X>25)

(&NAME *EQ GSD)
(&NAME *EQ &GSD)
(&NAME *EQ 'GSD')
(&BLANK *EQ ' ')
```

Logical expressions

The operands in a logical expression consist of relational expressions, logical variables, or constants that are separated by logical operators.

Two or more of these types of operands can be used in combinations, making up two or more expressions within expressions, up to the maximum of five nested levels of parentheses. The result of a logical expression is a '0' or '1' that can be used as part of another expression or saved in logical variables.

The logical operators used to specify the relationship between the operands are *AND and *OR (as reserved values), and & and | (as symbols). The AND operator indicates that both operands (on either side of the operator) have to be a certain value to produce a particular result. The OR operator indicates that one or the other of its operands can determine the result.

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

The following are examples of logical expressions:

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

(&A *OR *NOT &B)
(&TOWN *EQ CHICAGO *AND &ZIP *EQ 60605)
```

Two examples of logical expressions used in the IF command are:

```
IF &A CALL PROG1
IF (&A *OR &B) CALL PROG1
```

Operators in expressions

Operators are used in expressions to indicate an action to be performed on the operands in the expression or the relationship between the operands.

There are four kinds of operators, one for each of the four types of expressions:

- Arithmetic operators (`+, -, *, /`)
- Character operator (`||, |>, |<`)
- Logical operators (`&, |, ¬`)
- Relational operators (`=, >, <, >=, <=, ¬=, ¬>, ¬<`)

Each operator must be between the operands of the expression in which it is used; for example, (`&A + 4`). Operators can be specified as a predefined value (for example, *EQ) or as a symbol (for example, =).

- All predefined value operators must have a blank on each side of the operator:

  ```
  (&VAR *EQ 7)
  ```

- Except for the division operator (/), symbolic operators need no blanks on either side. For example, either (`&VAR=7`) or (`&VAR = 7`) is valid.

  Where the division operator follows a variable name, the division operator must be preceded by a blank. For example, (`&VAR / 5`) or (`&VAR /5`) is valid; (`&VAR/5`) is not valid.

The following character combinations are the predefined values and symbols that represent the four kinds of operators; they should not be used in unquoted strings for any other purpose.

Table 16. Predefined values and symbols representing the four kinds of operators

Predefined value	Predefined symbol	Meaning	Type
	+	Addition	Arithmetic operator
	-	Subtraction	Arithmetic operator
	*	Multiplication	Arithmetic operator
	/	Division	Arithmetic operator
*CAT	|| [1]	Concatenation	Character string operator
*BCAT	|> [1]	Blank insertion with concatenation	Character string operator
*TCAT	|< [1]	Blank truncation with concatenation	Character string operator
*AND	&	AND	Logical operator
*OR	| [1]	OR	Logical operator
*NOT	¬ [2]	NOT	Logical operator
*EQ	=	Equal	Relational operator
*GT	>	Greater than	Relational operator
*LT	<	Less than	Relational operator
*GE	>=	Greater than or equal	Relational operator
*LE	<=	Less than or equal	Relational operator
*NE	¬= [2]	Not equal	Relational operator
*NG	¬> [2]	Not greater than	Relational operator
*NL	¬< [2]	Not less than	Relational operator

Notes:

1. In some national character sets and in the multinational character set, the character | (hexadecimal 4F) is replaced by the character ! (exclamation point). Either ! or *OR can be used as the OR operator and either || or *CAT, !> or *BCAT, and !< or *TCAT can be used for concatenation in those character sets.
2. In some national character sets and in the multinational character set, the character ¬ (hexadecimal 5F) is replaced by the character *. Either * or *NOT can be used as the NOT operator in those character sets.

Related concepts

Special character use
Special characters can be used only in these special ways or inside quoted character strings or comments.

Symbolic operators
A variety of characters can be used as symbolic operators in CL commands.

Related tasks

Adding conditional breakpoints

To add a conditional breakpoint to a program that is being debugged, use the **Add Breakpoint (ADDBKP)** command to specify the statement and condition.

Priority of operators when evaluating expressions

When multiple operators occur in an expression, the expression is evaluated in a specific order depending upon the operators in the expression.

Parentheses can be used to change the order of expression evaluation. The following table shows the priority of all the operators used in expressions, including signed decimal values.

Table 17. Priority of operators

Priority	Operators
1	signed (+ and -) decimal values, *NOT, ¬
2	* , /
3	+, - (when used between two operands)
4	*CAT, ||, *BCAT, |>, *TCAT, |<
5	*GT, *LT, *EQ, *GE, *LE, *NE, *NG, *NL, >, <, =, >=, <=, ¬=, ¬>, ¬<
6	*AND, &
7	*OR, |

A priority of 1 is the highest priority (signed values are evaluated first); a priority of 7 is the lowest priority (OR relationships are evaluated last). When operators with different priority levels appear in an expression, operations are performed according to priorities.

When operators of the same priority appear in an expression, operations are performed from left to right within the expression. Parentheses can always be used to control the order in which operations are performed. The value of a parenthetical expression is determined from the innermost level to the outermost level, following the previous priorities stated within matching sets of parentheses.

Built-in functions for CL

Control language (CL) provides several built-in functions. Built-in functions are used in arithmetic, character string, relational, or logical expressions. All of these built-in functions can only be used in compiled CL programs or procedures.

CL provides the following built-in functions:

- %ADDRESS
- %BINARY
- %CHAR
- %CHECK
- %CHECKR
- %DEC
- %INT
- %LEN
- %LOWER
- %OFFSET
- %PARMS
- %SCAN
- %SIZE
- %SUBSTRING

- %SWITCH
- %TRIM
- %TRIML
- %TRIMR
- %UINT
- %UPPER

Related concepts

Additional rules for unique names
Additional rules involve special characters (as an extra character) for object naming.

Related reference

%ADDRESS built-in function
The address built-in function (%ADDRESS or %ADDR) can be used to change or test the memory address stored in a CL pointer variable.

%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.

%CHAR built-in function
%CHAR converts logical, decimal, integer, or unsigned integer data to character format. The converted value can be assigned to a CL variable, passed as a character constant to another program or procedure, or specified as a value for a command parameter of a CL command run from compiled CL.

%CHECK built-in function
The check built-in function (%CHECK) returns the first position of a *base string* that contains a character that does not appear in the *comparator string*. If all of the characters in the base string also appear in the comparator string, the function returns 0.

%CHECKR built-in function
The reverse check built-in function (%CHECKR) returns the last position of a *base string* that contains a character that does not appear in the *comparator string*. If all of the characters in the base string also appear in the comparator string, the function returns 0.

%DEC built-in function
%DEC converts character, logical, decimal, integer, or unsigned integer data to packed decimal format. The converted value can be assigned to a CL variable, passed as a numeric constant to another program or procedure, or specified as a value for a command parameter of a CL command run from compiled CL.

%INT built-in function
%INT converts character, logical, decimal, or unsigned integer data to integer format. The converted value can be assigned to a CL variable, passed as a numeric constant to another program or procedure, or specified as a value for a command parameter of a CL command run from compiled CL.

%LEN built-in function
The %LEN built-in function returns the number of digits or characters of the CL numeric or character variable.

%LOWER built-in function
The %LOWER built-in function returns a character string that is the same length as the argument specified with each uppercase letter replaced by the corresponding lowercase letter.

%OFFSET built-in function
The offset built-in function (%OFFSET or %OFS) can be used to store or change the offset portion of a CL pointer variable.

%SCAN built-in function
The scan built-in function (%SCAN) returns the first position of a *search argument* in the *source string*, or 0 if it was not found.

%SIZE built-in function

The %SIZE built-in function returns the number of bytes occupied by the CL variable.

%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.

%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'.

%TRIM built-in function
The trim built-in function (%TRIM) with one parameter produces a character string with any leading and trailing blanks removed. The trim built-in function (%TRIM) with two parameters produces a character string with any leading and trailing characters that are in the *characters to trim* parameter removed.

%TRIML built-in function
The trim left built-in function (%TRIML) with one parameter produces a character string with any leading blanks removed. The trim left built-in function (%TRIML) with two parameters produces a character string with any leading characters that are in the *characters to trim* parameter removed.

%TRIMR built-in function
The trim right built-in function (%TRIMR) with one parameter produces a character string with any trailing blanks removed. The trim right built-in function (%TRIMR) with two parameters produces a character string with any trailing characters that are in the *characters to trim* parameter removed.

%UINT built-in function
%UINT converts character, logical, decimal, or integer data to unsigned integer format. The converted value can be assigned to a CL variable, passed as a numeric constant to another program or procedure, or specified as a value for a command parameter of a CL command run from compiled CL.

%UPPER built-in function
The %UPPER built-in function returns a character string that is the same length as the argument specified with each lowercase letter replaced by the corresponding uppercase letter.

Naming within commands

The type of name you specify in control language (CL) determines the characters you can use to specify a name.

For certain types of names, there are restrictions on the use of certain characters to represent the name. Those types of names are *NAME, *SNAME, and *CNAME.

Note: For a description of how to specify these names when you use command definitions to create commands, see the PARM (parameter) and ELEM (element) statements in “Defining a CL command” on page 305.

The characters allowed for the *NAME, *SNAME, and *CNAME names and the rules you use to specify them are shown in the following table.

*Table 18. Allowable characters for *NAME, *SNAME, and *CNAME*

Type of name	First character	Other characters	Min. length	Max. length
*NAME [1]	A-Z, $, #, @	A-Z, 0-9, $, #, @, _, .	1	256
*SNAME [1]	A-Z, $, #, @	A-Z, 0-9, $, #, @, _	1	256

*Table 18. Allowable characters for *NAME, *SNAME, and *CNAME (continued)*				
Type of name	**First character**	**Other characters**	**Min. length**	**Max. length**
*CNAME [1]	A-Z, $, #, @	A-Z, 0-9, $, #, @		
Quoted name [2]	"3	Any except blank, *, ?, ', ", X'00'-X'3F', and X'FF'	3	256

Notes:

1. The system converts lowercase letters to uppercase letters.
2. Quotation marks can only be used for basic names (*NAME).
3. Both the first and last characters must be a quotation mark (").

Related concepts

Object naming rules
These rules are used to name all IBM i objects used in control language (CL) commands. The parameter summary table for each CL command shows whether a simple object name, a qualified name, or a generic name can be specified.

Related reference

CL command definition statements
Command definition statements allows system users to create additional CL commands to meet specific application needs.

CL command label
Command labels identify particular commands for branching purposes in a CL program. Labels can also be used to identify statements in CL programs that are being debugged. They can identify statements used either as breakpoints or as starting and ending statements for tracing purposes.

Folder and document names

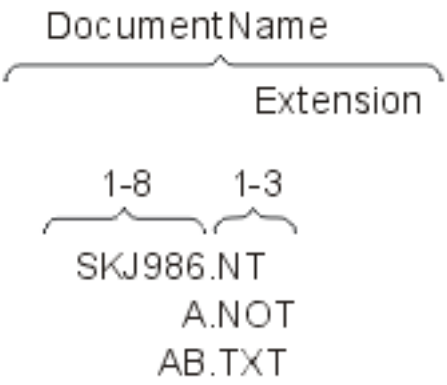
Folder and document names should describe the contents of the folder or document.

Folder names must be unique and should be easy to type, as well as descriptive to a user. To find a particular folder on the system and change a document stored in it, you must either supply the folder name or select it from a list of names.

Document names must be unique in the folder and should be easy to type, as well as descriptive. You should give careful consideration to the names you use to help you find the document later.

The name you use for a folder or a document must follow these rules:

- The name must be unique within a folder.
- A document or folder name can be 1 to 12 characters long, including an optional extension. If no extension is included, a document or folder name can have a maximum of eight characters. If an extension is included, the extension must start with a period and can have up to three additional characters. An extension in the document name allows you to identify the document by using specific information that can help you do a selective listing of documents on your system.



- A document or folder name can include any single-byte EBCDIC character except for the following special characters that the system uses for other purposes.

Character	Special uses
Asterisk (*)	Multiplication operator, indicates generic names, and indicates IBM i reserved values
Slash (/)	Division operator, delimiter within system values, and separates parts of qualified object names
Question mark (?)	Initiates requests for system help

- When a folder is stored in another folder, both folder names are used, separated by a slash (/). That combination of names is called a **folder path**. For example, if a folder named FOLDR2 is stored in FOLDR1, the path for FOLDR2 is FOLDR1/FOLDR2. FOLDR1 is the **first-level folder**. FOLDR2 is the **next-level folder**. The name of a single folder can be 1 to 12 characters long, including an optional extension. A folder path can contain a maximum of 63 characters.

  Folder names should not begin with Q because the system-supplied folder names begin with Q. The following are examples of permitted folder names and folder paths:

```
@LETTERS
FOLDER.PAY
PAYROLL/FOLDER.PAY
#TAX1/FOLD8.TAX/$1988/PAYROLL/FOLDER.PAY
```

Notes:

1. In CL commands, folder path names must be enclosed in single quotation marks to prevent the system from processing them as qualified (library/object) names. If a single quotation mark is to be part of the name, it must be specified as two consecutive single quotation marks.
2. A number of CL commands act on either documents or folders, and some act on both. The abbreviation DLO (document library object) is used when referring to either a document or folder.
3. In CL commands, folder and document names must be enclosed in single quotation marks if they contain characters that are CL delimiters.
4. The system does not recognize graphic characters; it recognizes only code points and uses the following assumptions:
   - All folder and document names are encoded using single-byte EBCDIC code pages. Since code points hex 41 through FE represent graphic characters in those code pages, they are the only code points that can be used in folder and document names.
   - Code point hexadecimal 5C represents the asterisk (*); 61 represents the slash (/); and 6F represents the question mark (?). They cannot be used in folder and document names.

- The code points for lowercase letters in English are converted to the code points for uppercase letters. Hexadecimal 81 through 89 are converted to C1 through C9; 91 through 99 are converted to D1 through D9; and A2 through A9 are converted to E2 through E9.

In addition to the folder and document names previously described, folders and documents are internally classified in the system by their system object names. These are 10-character names derived from date/time stamps, and, while they are generally not known to the user, they may be specified on some CL commands by specifying *SYSOBJNAM for the folder or document name and by specifying the system object name in a separate parameter.

Related concepts

Object naming rules

These rules are used to name all IBM i objects used in control language (CL) commands. The parameter summary table for each CL command shows whether a simple object name, a qualified name, or a generic name can be specified.

Related information

Local device configuration PDF

Create Device Desc (Display) (CRTDEVDSP) command

IBM i objects

An *object* is a named unit that exists (occupies space) in storage, and on which operations are performed by the operating system. IBM i objects provide the means through which all data processing information is stored and processed by the IBM i operating system.

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 system. You need only to 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.

CL commands perform operations on the IBM i objects. Several types of IBM i objects are created and used in the control language. IBM i objects have the following characteristics in common:

- Objects have a set of descriptive attributes that are defined when the object is created.
- Objects that have to be used by the system to perform a specific function must be specified in the CL command that performs that function.
- Objects have a set of attributes that describe it and give the specific values assigned for those attributes.
- Generally, objects are independent of all other objects. However, some objects must be created before other objects can be created; for example, a logical file cannot be created if the physical file it must be based on does not exist.
- Objects must be created before other operations that use the object are performed. Descriptions of the create commands (those commands that begin with the letters CRT) give more information about the object types that they create.
- Every IBM i object that is used by the control language has a name. The object name specified in a CL command identifies which object is used by the operating system to perform the function of the command.
- Objects have either a simple, qualified, or generic name.

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.

Related concepts

Objects and libraries

Tasks and concepts specific to objects and libraries include performing functions on objects, creating libraries, and specifying object authority.

Library objects

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.

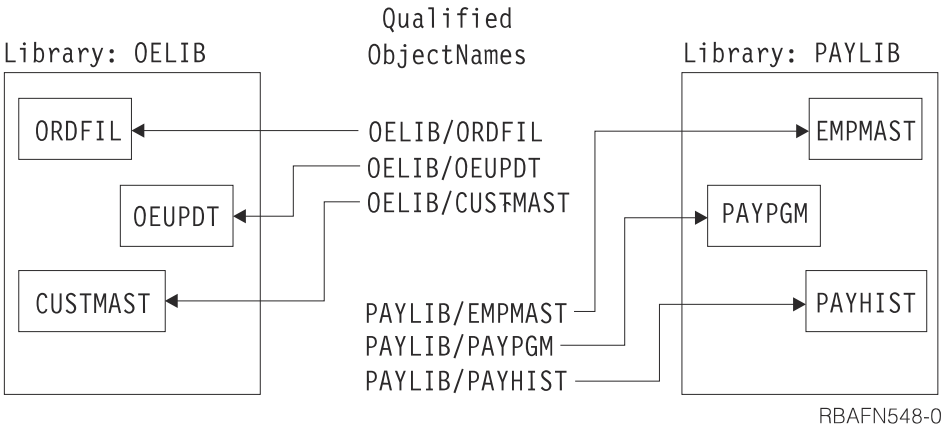
You can use libraries to group the objects into any meaningful collection. For example, you can group objects according to security requirements, backup requirements, or processing requirements. The amount of available disk storage limits the number of objects that a library can contain, and the number 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.

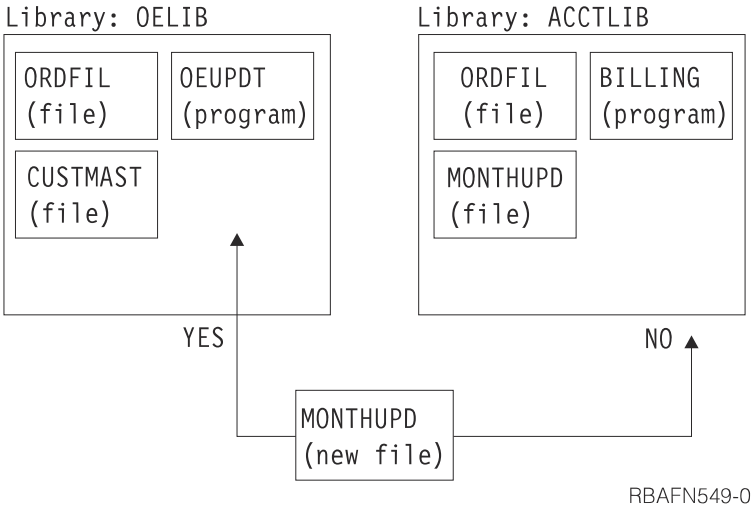
Most types of objects are placed in a library when they are created. The AUT parameter on **CRTLIB** defines the public authority of the library. The CRTAUT parameter specifies the default authority for objects that are created into the library. If the command creating the object specifies ***LIBCRTAUT** 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 figure 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 figure, 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.

Objects that use the integrated file system are located in directories and can be found by using path names or object name patterns instead of searching libraries. You can also use these directories to locate objects.

Related concepts

Integrated file system

Related tasks

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.

Related information

Independent disk pool examples

External object types

Many types of external objects are stored in libraries. Some types of external objects can only be stored in the integrated file system in directories.

Some types of objects can only be stored in a specific library. For those object types, the library where the objects are created is shown in the Default user library column in Table 19 on page 129.

Other types of objects allow a library name to be specified when the object is created. The default value for the library name qualifier on the CL commands that create those types of objects is the special value *CURLIB. The *CURLIB value causes the name of the current library to be used. The current library for a job or thread can be set as a user profile attribute or job attribute or by running the **Change Current Library (CHGCURLIB)** command. If there is no current library in effect when the object is created, library QGPL is used.

The other types of objects, identified by N/A (not applicable) in the Default user library column, cannot be stored in libraries or directories.

With the exception of the **Dump System Object (DMPSYSOBJ)** command, you cannot specify the object type in the format shown in the hexadecimal format column with commands.

Table 19. Predefined values and default library or directory locations for external IBM i object types

Value	Object type	Hexadecimal format	Default user library or directory
*ALRTBL	Alert table	0E09	*CURLIB
*AUTL	Authorization list	1B01	QSYS
*BLKSF	Block special file	1E05	Current directory
*BNDDIR	Binding directory	1937	*CURLIB
*CFGL	Configuration list	1918	QSYS
*CHRSF	Character special file	1E06	Current directory
*CHTFMT	Chart format	190D	*CURLIB
*CLD	C/400 locale description	190B	*CURLIB
*CLS	Class	1904	*CURLIB
*CMD	Command	1905	*CURLIB
*CNNL	Connection list	1701	QSYS
*COSD	Class-of-service description	1401	QSYS
*CRG	Cluster resource group	192C	QUSRSYS
*CRQD	Change request description	0E0F	*CURLIB
*CSI	Communications side information	1935	*CURLIB
*CSPMAP	Cross-system product map	1922	*CURLIB
*CSPTBL	Cross-system product table	1923	*CURLIB
*CTLD	Controller description	1201	QSYS
*DDIR	Distributed file directory	1F02	N/A
*DEVD	Device description	1001	QSYS
*DIR	Directory	0C01	Current directory

Table 19. Predefined values and default library or directory locations for external IBM i object types (continued)

Value	Object type	Hexadecimal format	Default user library or directory
*DOC	Document	190E	QDOC
*DSTMF	Distributed stream file	1F01	N/A
*DTAARA	Data area	190A	*CURLIB
*DTADCT	Data dictionary	1920	library with same name as data dictionary
*DTAQ	Data queue	0A01	*CURLIB
*EDTD	Edit description	1908	QSYS
*EXITRG	Exit registration	1913	QUSRSYS
*FCT	Forms control table	0E04	*CURLIB
*FIFO	First-in-first-out special file	1E07	Current directory
*FILE	File	1901	*CURLIB
*FLR	Folder	1912	QDOC
*FNTRSC	Font resources	1926	*CURLIB
*FNTTBL	Font mapping table	192B	*CURLIB
*FORMDF	Form definition	1928	*CURLIB
*FTR	Filter	0E0B	*CURLIB
*GSS	Graphics symbol set	190C	*CURLIB
*IGCDCT	Double-byte character set (DBCS) conversion dictionary	0E06	*CURLIB
*IGCSRT	Double-byte character set (DBCS) sort table	191A	*CURLIB
*IGCTBL	Double-byte character set (DBCS) font table	1910	QSYS
*IMGCLG	Image Catalog	192E	QUSRSYS
*IPXD	Internetwork packet exchange description	191E	QSYS
*JOBD	Job description	1903	*CURLIB
*JOBQ	Job queue	0E01	*CURLIB
*JOBSCD	Job schedule	0E0C	*CURLIB
*JRN	Journal	0901	*CURLIB
*JRNRCV	Journal receiver	0701	*CURLIB
*LIB	Library	0401	QSYS
*LIND	Line description	1101	QSYS
*LOCALE	Locale	1921	*CURLIB
*MEDDFN	Media definition	191C	*CURLIB

Table 19. Predefined values and default library or directory locations for external IBM i object types (continued)

Value	Object type	Hexadecimal format	Default user library or directory
*MENU	Menu description	1916	*CURLIB
*MGTCOL	Management collection	192D	NA, or QPFRDATA if library specified using QYPSCSCA API
*MODD	Mode description	1501	QSYS
*MODULE	Compiler unit	0301	*CURLIB
*MSGF	Message file	0E03	*CURLIB
*MSGQ	Message queue	1902	*CURLIB
*M36	AS/400 Advanced 36 machine	1E04	*CURLIB
*M36CFG	AS/400 Advanced 36 machine configuration	1924	*CURLIB
*NODGRP	Node group	192A	*CURLIB
*NODL	Node list	0E0E	*CURLIB
*NTBD	NetBIOS description	1914	QSYS
*NWID	Network interface description	1601	QSYS
*NWSCFG	Network server configuration	1939	QUSRSYS
*NWSD	Network server description	1D01	QSYS
*OUTQ	Output queue	0E02	*CURLIB
*OVL	Overlay	1929	*CURLIB
*PAGDFN	Page definition	1936	*CURLIB
*PAGSEG	Page segment	1927	*CURLIB
*PDFMAP	Portable Document Format map	0E11	*CURLIB
*PDG	Print Descriptor Group	1930	*CURLIB
*PGM	Program	0201	*CURLIB
*PNLGRP	Panel group definition	1915	*CURLIB
*PRDAVL	Product availability	1933	QSYS
*PRDDFN	Product definition	191B	QSYS
*PRDLOD	Product load	191D	QSYS
*PSFCFG	Print Services Facility configuration	1925	*CURLIB
*QMFORM	Query management form	1932	*CURLIB
*QMQRY	Query management query	1931	*CURLIB
*QRYDFN	Query definition	1911	QGPL

Table 19. Predefined values and default library or directory locations for external IBM i object types (continued)

Value	Object type	Hexadecimal format	Default user library or directory
*RCT	Reference code translate table	0E08	QGPL
*SBSD	Subsystem description	1909	*CURLIB
*SCHIDX	Search index	0E07	QGPL
*SOCKET	Local socket	1E03	N/A
*SPADCT	Spelling aid dictionary	1C01	QGPL
*SQLPKG	Structured Query Language package	0202	*CURLIB
*SQLUDT	User-defined SQL type	191F	*CURLIB
*SQLXSR	SQL XML schema repository	1949	*CURLIB
*SRVPGM	Service program	0203	*CURLIB
*SSND	Session description	0E05	QGPL
*STMF	Bytestream file	1E01	Current directory
*SVRSTG	Server storage space	1917	QUSRSYS
*SYMLNK	Symbolic link	1E02	Current directory
*S36	System/36 machine description	1919	QGPL
*TBL	Table	1906	*CURLIB
*TIMZON	Time zone description	192F	QSYS
*USRIDX	User index	0E0A	*CURLIB
*USRPRF	User profile	0801	QSYS
*USRQ	User queue	0A02	*CURLIB
*USRSPC	User space	1934	*CURLIB
*VLDL	Validation list	0E10	*CURLIB
*WSCST	Workstation user customization object	1938	*CURLIB

Related concepts

Object types and common attributes
Each type of object on the system has a unique purpose within the system, and each object type has a common set of attributes that describes the object.

Related reference

OBJTYPE parameter
The object type (OBJTYPE) parameter specifies the types of IBM i objects that can be operated on by the command in which they are specified.

Related information

Change System Collector Attributes (QYPSCSCA, QypsChgSysCollectorAttributes) API

Simple and qualified object names

The name of a specific object that is located in a library can be specified as a simple name or as a qualified name.

A *simple object name* is the name of the object only. A *qualified object name* is the name of the library where the object is stored followed by the name of the object. In a qualified object name, the library name is connected to the object name by a slash (/).

Either the simple name or the qualified name of an object can be specified if the object exists in one of the libraries named in the job's library list; the library qualifier is optional in this case. A qualified name must be specified if the named object is not in a library named in the library list.

Note: Although a job name also has a qualified form, it is not a qualified object name because a job is not an IBM i object. A job name is qualified by a user name and a job number, not by a library name.

The following table shows how simple and qualified object names are formed.

Table 20. Simple and qualified object names

Name type	Name syntax	Example
Simple object name	object-name	OBJA
Qualified object name	library-name/object-name	LIB1/OBJB

Related concepts

CL command delimiter characters
Command delimiter characters are special characters or spaces that identify the beginning or end of a group of characters in a command.

CL command definition
Command definition allows you to create additional control language commands to meet specific application needs. These commands are similar to the system commands.

Related reference

CL command names
The command name identifies the function that will be performed by the program that is called when the command is run. Most command names consist of a combination of a verb (or action) followed by a noun or phrase that identifies the receiver of the action (or object being acted on): (command = verb + object acted on).

CL command definition statements
Command definition statements allows system users to create additional CL commands to meet specific application needs.

JOB parameter
The JOB parameter specifies the name of the job to which the command is applied.

Generic object names

Generic object names can be used when referring to multiple objects with similar names.

A *generic object name* can refer to more than one object. That is, a generic name contains one or more characters that are the first group of characters in the names of several objects. The system then searches for all the objects that have those characters at the beginning of their names and are in the libraries named in the library list. A generic name is identified by an asterisk (*) as the last character in the name.

A quoted generic name consists of a generic name enclosed in quotation marks. Unlike quoted names, if there are no special characters between the quotation marks, the quotation marks are not removed. For example, the generic name "ABC*" causes the system to search for all objects whose name begins with a quotation mark followed by the three letters ABC.

A generic name can also be qualified by a library name. If the generic name is qualified, the system searches only the specified library for objects whose names begin with that generic name.

Note: A generic name also can be qualified by one or more directories if it is a path name. In a path name, letters can be specified before and after the asterisk (*).

When you specify a generic name, the system performs the required function on all objects whose names begin with the specified series of characters. You must have the authority required to perform that function on every object the generic name identifies. If you do not have the required authority for an object, the function is not performed and a diagnostic message is issued for each instance that the attempted generic function failed. A completion message is issued for each object the generic function operates on successfully. You must view the online low-level messages to see the completion messages. After the entire generic function is completed, a completion message is issued that states that all objects were operated on successfully. If one or more objects cannot be successfully operated on, an escape message is issued. If an override is in effect for a specific device file, the single object name specified on the override, rather than the generic name, is used to perform the operation.

You might not be able to use a generic name for delete, move, or rename commands if the library containing the objects is already locked. A search for generic object names requires a more restrictive lock on the library containing the objects than a search for full object names. The more restrictive lock is necessary to prevent another user from creating an object with the same name as the generic search string in the library while the delete, move, or rename command is running. You can circumvent this problem by using the full name of the objects instead of a generic name. Or you can end the job or subsystem that has a lock on the library.

Note: Use the **Work with Object Locks (WRKOBJLCK)** command to determine which jobs or subsystems have a lock on the library.

For some commands, a library qualifier can be specified with the generic name to limit the scope of the operation. For example, a **Change Print File (CHGPRTF)** command with `FILE(LIB1/PRT*)` performs the operation on printer files that begin with PRT in library `LIB1` only; printer files in other libraries are not affected.

The limitations associated with the various library qualifiers are as follows:

- *library-name:* The operation is performed on generic object names only in the specified library.
- *LIBL: The operation is performed on generic object names in the library list associated with the job that requested the generic operation.
- *CURLIB: The operation is performed on generic object names in the current library.
- *ALL: The operation is performed on generic object names in all libraries for which you are authorized.
- *USRLIBL: The operation is performed on generic object names in the user part of the library list for the job.
- *ALLUSR: The operation is performed on all nonsystem libraries (libraries that do not start with the letter Q), with some exceptions.

  Note: A different library name, of the form QUSRVxRxMx, can be created by the user for each release that IBM supports. VxRxMx is the version, release, and modification level of the library.

Table 21. Generic object name

Name type	Name syntax	Example
Simple generic name	generic-name*	OBJ*
Qualified generic name	library-name/generic-name*	LIB1/OBJ*
Quoted generic name	"generic-name*"	"ABC*"

Related concepts
Integrated file system

Generic library names

Object naming rules

These rules are used to name all IBM i objects used in control language (CL) commands. The parameter summary table for each CL command shows whether a simple object name, a qualified name, or a generic name can be specified.

Generic names (*GENERIC)

A *generic name* is one that contains at least one initial character that is common to a group of objects, followed by an asterisk.

Object naming rules

These rules are used to name all IBM i objects used in control language (CL) commands. The parameter summary table for each CL command shows whether a simple object name, a qualified name, or a generic name can be specified.

Naming a Single Object

In the name of a single object, each part (the simple name and the library qualifier name) can have a maximum of 10 characters.

Naming a User-Created Object

To distinguish a user-created object from an IBM-supplied object, you should not begin user-created object names with Q because the names of all IBM-supplied objects (except commands) begin with Q. Although you can use as many as 10 characters in CL object names, you might need to use fewer characters to be consistent with the naming rules of the particular high-level language that you are also using. Also, the high-level language might not allow underscores in the naming rules. For example, RPG limits file names to eight characters and does not allow underscores.

Naming a Generic Object

In a generic name, a maximum of nine alphanumeric characters can be used, not including the asterisk (*) that must immediately follow the last character.

INV and INV* are valid values where a generic name is accepted. When the name INV is specified, only the object INV is referenced. When the generic name INV* is specified, objects that begin with INV are referred to, such as INV, INVOICE, INVENTORY, and INVENPGM1. When the quoted generic name "INV*" is specified, objects that begin with "INV" are referred to, such as "INV%1" and "INV>."

Object Library Qualifier Limitations

No library qualifier can be specified with the object name if the object being created is a library, user profile, line description, controller description, device description, mode description, class-of-service description, or configuration list. A library name can never be qualified because a library cannot be placed in a library. The other object types (*USRPRF, *LIND, *CTLD, *DEVD, *MODD, *COSD, and *CFGL) appear to be types that exist only in the QSYS library. When only the name of an object of these object types is accepted, a library qualifier cannot be specified with the object name. On the **Display Object Description (DSPOBJD)** command, where any object name is accepted, QSYS can be specified.

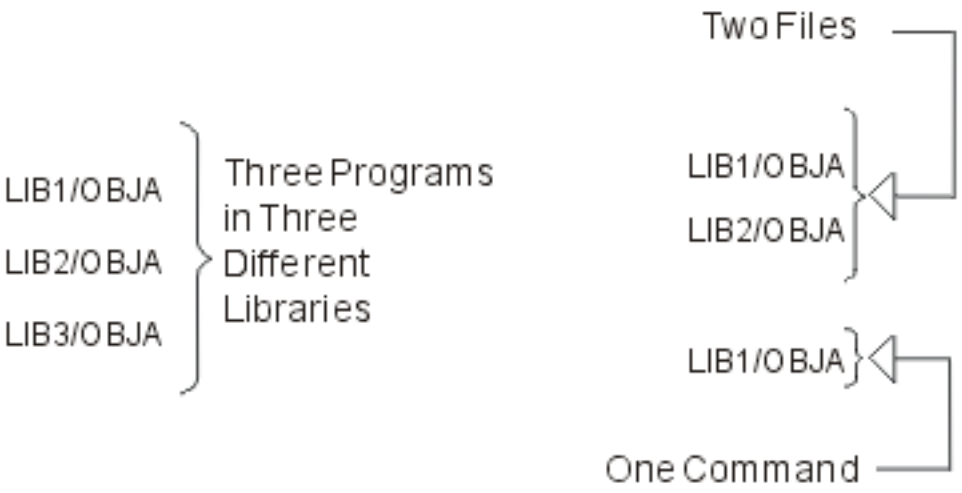
Library List Qualifiers

The predefined value *LIBL (and others, such as *CURLIB) can be used in place of a library name in most commands. *LIBL indicates that the libraries named in the job's library list are used to find the object named in the second part of the qualified name.

Duplicate Object Names

Duplicate names are not allowed for objects of the same type in the same library.

Two objects with the same name cannot be stored in the same library unless their object types are different. Two objects named OBJA can be stored in the library LIBx only if, for example, one of the objects is a program and the other is a file. The following combinations of names and object types could all exist on the system at the same time.



If more than one library contains an object with the same name (and both libraries are in the same library list) and a library qualifier is not specified with the object name, the first object found by that name is used. Therefore, when you have multiple objects with the same name, you should specify the library name with the object name or ensure that the appropriate library occurs first in the library list. For example, if you are testing and debugging and choose not to qualify the names, ensure that your test library precedes your production library in the library list.

Default libraries

In a qualified object name, the library name is typically optional. If an optional library qualifier is not specified, the default given in the command's description is used (typically either *CURLIB or *LIBL). If the named object is being created, the current library is the default; when the object is created, it is placed either in the current library or in the QGPL (the general purpose library) if no current library is defined. For objects that already exist, *LIBL is the default for most commands, and the job's library list is used to find the named object. The system searches all of the libraries currently in the library list until it finds the object name specified.

Related concepts

Naming within commands
The type of name you specify in control language (CL) determines the characters you can use to specify a name.

Generic object names
Generic object names can be used when referring to multiple objects with similar names.

Folder and document names
Folder and document names should describe the contents of the folder or document.

Related reference

CL command names
The command name identifies the function that will be performed by the program that is called when the command is run. Most command names consist of a combination of a verb (or action) followed by a noun

or phrase that identifies the receiver of the action (or object being acted on): (command = verb + object acted on).

*Communication names (*CNAME)*

A communications name has a restrictive name syntax. It is typically used to refer to a device configuration object whose name length and valid character set is limited by one or more communication architectures.

Communications names are the same as unquoted basic names with some exceptions:

1. Periods (.) and underscores (_) cannot be used.
2. For IBM commands, *CNAME is limited to 8 characters.

An example of a communications name is shown as follows:

```
APPN3@@
```

Because restricted character sets are sometimes used by other IBM systems, use caution when choosing names that use the special characters #, $, and @. These special characters might not be on the remote system's keyboard. The names that can be exchanged with the remote systems include the following names:

- Network IDS
- Location names
- Mode names
- Class-of-service names
- Control point names

Related concepts

Names (*NAME)

When you create basic names and basic names in quoted form, follow these rules.

Related reference

CL command label

Command labels identify particular commands for branching purposes in a CL program. Labels can also be used to identify statements in CL programs that are being debugged. They can identify statements used either as breakpoints or as starting and ending statements for tracing purposes.

*Generic names (*GENERIC)*

A *generic name* is one that contains at least one initial character that is common to a group of objects, followed by an asterisk.

The asterisk identifies the series of common characters as a generic name; otherwise, the system interprets the series of characters as the name of a specific object.

Related concepts

Generic object names

Generic object names can be used when referring to multiple objects with similar names.

*Names (*NAME)*

When you create basic names and basic names in quoted form, follow these rules.

*NAME (basic name in unquoted form)

Every basic name can begin with the characters A-Z, $, #, or @ and can be followed by up to nine characters. The remaining characters can include the same characters as the first but can also include numbers 0-9, underscores (_), and periods (.). Lowercase letters are changed to uppercase letters by the system. Basic names used in IBM-supplied commands can be no longer than 10 characters. However, in

your own commands, you can define parameters of type *NAME (specified on the TYPE parameter of the PARM or ELEM statements) with up to 256 characters.

Examples of basic names are shown as follows:

```
A987@.442#    ONE_NAME    LIB_0690    $LIBX
```

Names can be entered in quoted or unquoted form. If you use the quoted form, the following rules and considerations also apply:

*NAME (basic name in quoted form)

Every quoted name must begin and end with a quotation mark ("). The middle characters of a quoted name can contain any character except , *, ?, ', ", hex 00 through 3F, or hex FF, and is delimited by a slash. Quoted names allow you to use graphic characters in the name. The quoted form of basic names used in IBM-supplied commands can be no longer than 8 characters (not including the quotation marks). In your own commands, you can define parameters of type *NAME in quoted form with up to 254 characters (not including the quotation marks).

Note: Only basic names can be used in quoted form.

Examples of quoted names are shown as follows:

```
"A"         "AA%abc"         "ABC%%abc"
```

When you use quoted names, you should be aware of certain restrictions:

- Code points in a name might not be addressable from all keyboards.
- Characters in a quoted name might not be valid in a high-level language.
- The System/38 environment supports only simple (*SNAME) names. If other characters are used, the objects cannot be accessed as System/38 environment objects.
- Names that are longer than eight characters cannot be accessed by the System/36 environment unless control language overrides are used.
- A Structured Query Language (SQL) name that contains a period must be specified in an SQL statement in quotation marks.

If a name enclosed in quotation marks is a valid unquoted basic name, the quotation marks are removed. Thus, "ABC" is equivalent to ABC. Because the quotation marks are removed, they are not included in the length of the name. "ABCDEFGHIJ" is, therefore, a valid name on IBM* commands even though it is longer than 10 characters.

Related concepts

Communication names (*CNAME)
A communications name has a restrictive name syntax. It is typically used to refer to a device configuration object whose name length and valid character set is limited by one or more communication architectures.

Simple names (*SNAME)
Simple names are used for control language (CL) variables, labels, and keywords to simplify the syntax of CL. Simple names are the same as unquoted basic names but with one exception: periods (.) cannot be used.

Related reference

CL command label

Command labels identify particular commands for branching purposes in a CL program. Labels can also be used to identify statements in CL programs that are being debugged. They can identify statements used either as breakpoints or as starting and ending statements for tracing purposes.

*Path names (*PNAME)*

A *path name* is a character string that can be used to locate objects in the integrated file system.

The string can consist of one or more elements, each separated by a slash (/). Each element is typically a directory or equivalent, except for the last element, which can be a directory, another object such as a file, or a generic presentation of an object or objects to be located.

Note: Some CL commands also allow the backslash (\) to be used as a separator by automatically converting the backslash (\) to a slash (/). Some other CL commands, however, treat the backslash (\) no differently than any other character. Therefore, the backslash (\) separator should be used with caution.

The / and \ characters and nulls cannot be used in the individual components of the path name when the / and \ characters are used as separators. The name can or cannot be changed to uppercase, depending on whether the file system containing the object is case-sensitive and whether the object is being created or searched for. If the parameter is defined as CASE(*MONO) (the default), any values that are not enclosed in single quotation marks will be changed to uppercase by the command analyzer.

A separator character (for example, /) at the beginning of a path name means that the path begins at the top most directory, the root (/) directory. If the path name does not begin with a separator character, the path is assumed to begin at the current directory of the user entering the command.

The path name must be represented in the CCSID currently in effect for the job. If the CCSID of the job is 65535, the path name must be represented in the default CCSID of the job. Hard-coded path names in programs are encoded in CCSID 37. Therefore, the path name should be converted to the job CCSID before being passed to the command. The maximum length of the path name character string on the CL commands is 5000 characters.

When operating on objects in the QSYS.LIB file system, the component names must be of the form name.object-type; for example:

```
'/QSYS.LIB/PAY.LIB/TAX.FILE'
```

If the objects are in an independent ASP QSYS.LIB file system, the name must be of the form:

```
'/asp-name/QSYS.LIB/PAY.LIB/TAX.FILE'
```

where `asp-name` is the name of the independent ASP.

Path names must be enclosed in single quotation marks (') when entered on a command line if they contain special characters. These marks are optional when path names are entered on displays. If the path name includes any quoted strings or special characters; however, the enclosing '' marks must be included. The following are rules for using special characters:

- A tilde (~) character followed by a separator character (for example, /) at the beginning of a path name means that the path begins at the home directory of the user entering the command.
- A tilde (~) character followed by a user name and then a separator character (for example, /) at the beginning of a path name means that the path begins at the home directory of the user identified by the user name.
- In some commands, an asterisk (*) or a question mark (?) can be used in the last component of a path name to search for patterns of names. The * tells the system to search for names that have any number of characters in the position of the * character. The ? tells the system to search for names that have a single character in the position of the ? character.
- To avoid confusion with IBM i special values, path names cannot start with a single asterisk (*) character. To perform a pattern match at the beginning of a path name, use two asterisks (**).

  Note: This only applies to relative path names where there are no other characters before the asterisk.

- The path name must be enclosed in single quotation marks (') and quotation marks (") if any of the following characters are used in a component name:
  - Asterisk (*)

    Note: To avoid confusion with IBM i special values, do not start path names with a single asterisk (*) character.
  - Question mark (?)
  - Single quotation mark (')
  - Quotation mark (")
  - Tilde (~), if used as the first character in the first component name of the path name (if used in any other position, the tilde is interpreted as just another character)

  Note: Using the previous characters as the first character in the first component name of the path name is not recommended because the meaning of the character in a command string could be confused and it is more likely that the command string will be entered incorrectly.
- Do not use a colon (:) in path names. It has a special meaning within the system.
- The processing support for commands and associated user displays does not recognize code points below hexadecimal 40 as characters that can be used in command strings or on displays. If these code points are used, they must be entered as a hexadecimal representation, such as the following example:

  ```
  crtdir dir(X'02')
  ```

  Therefore, use of code points below hexadecimal 40 in path names is not recommended. This restriction applies only to commands and associated displays, not to APIs. In addition, a value of hexadecimal 00 is not allowed in path names.

Related tasks

Specifying the device name

Accessing the integrated file system

Related reference

OBJ parameter

The object (OBJ) parameter specifies the names of one or more objects affected by the command in which this parameter is used.

CL command label

Command labels identify particular commands for branching purposes in a CL program. Labels can also be used to identify statements in CL programs that are being debugged. They can identify statements used either as breakpoints or as starting and ending statements for tracing purposes.

*Simple names (*SNAME)*

Simple names are used for control language (CL) variables, labels, and keywords to simplify the syntax of CL. Simple names are the same as unquoted basic names but with one exception: periods (.) cannot be used.

Some examples of simple names are as follows:

```
NEWCMD    LIB_2
```

Related concepts

Names (*NAME)

When you create basic names and basic names in quoted form, follow these rules.

Related reference

CL command label

Command labels identify particular commands for branching purposes in a CL program. Labels can also be used to identify statements in CL programs that are being debugged. They can identify statements used either as breakpoints or as starting and ending statements for tracing purposes.

Additional rules for unique names

Additional rules involve special characters (as an extra character) for object naming.

- A command label must be immediately followed by a colon (:). Blanks can follow the colon, but none can precede it. A command label name cannot be a quoted name.
- A CL variable name must be preceded by an ampersand (&) to indicate that it is a CL variable used in a CL program.
- The built-in functions for CL must be preceded by a percent sign (%) to indicate that it is an IBM-supplied built-in function that can be used in an expression. A built-in function name cannot be a quoted name.

These special characters are not part of the name; each is an additional character attached to a name (making a maximum of 11 characters) indicating to the system what the name identifies.

The names of IBM i objects, CL program variables, system values, and built-in functions can be specified in the parameters of individual commands. Instead of specifying a constant value, a CL variable name can be used on most parameters in CL programs to specify a value that can change during the running of programs. It is the contents of the variable that identify the objects and variables that are used when the command is run.

Related reference

CL command label

Command labels identify particular commands for branching purposes in a CL program. Labels can also be used to identify statements in CL programs that are being debugged. They can identify statements used either as breakpoints or as starting and ending statements for tracing purposes.

Variable name

A variable contains a data value that can be changed when a program is run. A *variable name* is the name of the variable that contains the value.

Built-in functions for CL

Control language (CL) provides several built-in functions. Built-in functions are used in arithmetic, character string, relational, or logical expressions. All of these built-in functions can only be used in compiled CL programs or procedures.

Database files and device files used by CL commands

Many of the IBM-supplied CL commands use database files and device files during processing.

All of the commands and files for all licensed programs that meet the criteria are included:

- The types of files included are:
  - Database files: physical (PF) and logical (LF), including files with data and files used as model files (no data)
  - Device files: tape (TAPF) and printer (PRTF)
- The files are included only if a user might have a reason to use them, such as declaring them in a program so they can be overridden with another file. Two examples:
  - You want to change some attributes for a printer device file, such as the font to be used or the printed lines per inch (the FONT or LPI parameter).
  - You want to override the IBM-supplied file with your own output file (when valid).

The types of files not included in this section are:

- IBM-supplied display (DSPF) device files, because they should not be changed or overridden.
- Most of the IBM-supplied database files used by *directory* commands, *document library object* commands, and *optical index database* files, because they cannot be overridden.

As mentioned, IBM-supplied physical (PF) or logical (LF) files that are used as **model files** for certain commands are included in the following table. Some examples are the model files listed under the DSPFD, DSPJRN, and STRPFRMON commands. In most cases, these model files do not contain data; instead, they contain the *definitions* (or record formats) of the files to be created for storing the actual output data resulting from use of these commands. For the record formats of these files, you can display the file's description online or you can refer to the topic collection that documents that command.

Additionally, some files considered to be IBM-supplied files do not actually exist on the system until some function is used that requires the file and so creates it at that time.

The following notes describe how the table of commands and files are sorted and explain the meanings of the superscripts used in Table 22 on page 142.

Notes:

1. The first column of the table lists the CL commands that use the IBM-supplied files shown in the third column. The table entries are in alphabetic order by *command name* first. If there is more than one library for a command, they are further ordered by the file *library name* and then by *file name* within each library.
2. **A superscript 1 ([1]) ** following the description of a file indicates that the file is used only when the output from the command is directed to that *form* of output-by an output-related parameter on the command.
   - The superscript [1] is used at the end of *printer file* (PRTF) descriptions to show that use of the printer file is dependent on the job environment and the print-related value specified (or assumed) on the command. For commands with an OUTPUT parameter (primarily DSPxxx and WRKxxx commands), the output is printed either when OUTPUT(*) is specified in a batch job, or when OUTPUT(*PRINT) is specified in a batch or interactive job. For other commands, the output is printed if a *PRINT, *LIST, or *SRC value is specified on a different parameter.
   - The superscript [1] is used at the end of *database file* (PF or LF) descriptions to show that use of the database file is dependent on the print-related value specified (or assumed) on the command. For commands with an OUTPUT parameter (primarily DSPxxx and SAVxxx commands), the output is directed to the database file when OUTPUT(*OUTFILE) is specified. The same is true for other commands that specify a value similar to *OUTFILE on one of its parameters.
3. **A superscript 2 ([2]) ** following a file's description indicates that the file is a *model file* and not an output file. As a model file, it defines the record format of the file created to contain the actual output.
4. Those files showing "user-lib" as the file library do not exist on the system until the user creates them. When the command is used, the file is created in the user's specified library with the file name shown.

Table 22. Files used by CL commands (part 1)

Command name	File library	File name	File type	File usage
ADDDSTQ	QUSRSYS	QASNADSQ	PF	SNADS distribution queues table.
ADDDSTRTE	QUSRSYS	QASNADSQ	PF	SNADS distribution queues table.
	QUSRSYS	QASNADSR	PF	SNADS routing table.
ADDDSTSYSN	QUSRSYS	QASNADSA	PF	SNADS secondary node ID table.
ADDNETJOBE	QUSRSYS	QANFNJE	PF	Network job entry database file.
ADDSOCE	QUSRSYS	QAALSOC	PF	Sphere of control file.
ADDTAPCTG	QUSRSYS	QATAMID	PF	Cartridge ID db file.
	QUSRSYS	QLTAMID	LF	Cartridge ID logical file.
	QUSRSYS	QATACGY	PF	Category db file.
	QUSRSYS	QLTACGY	LF	Category logical file.

Table 22. Files used by CL commands (part 1) (continued)

Command name	File library	File name	File type	File usage
	QSYS	QSYSTAP	TAPF	Tape device file used for input.
ANSQST	QSYS	QPQAPRT	PRTF	Q & A printer file.
ANZDBF	QPFR	QAPTAZDR	PF	Performance data collection file: analyze application database files data.
	QPFR	QAPTDDS	PF	Performance data DDS source file.
	QPFR	QPPTANZD	PRTF	Performance printer file showing physical-to-logical and logical-to-physical database file relationships.
ANZDBFKEY	QPFR	QAPTAZDR	PF	Performance input file of analyze application database files data showing logical file key structures.
	QPFR	QAPTDDS	PF	Performance data DDS source file.
	QPFR	QPPTANKM	PRTF	Performance printer file containing logical file key structure data.
	QPFR	QPPTANZK	PRTF	Performance printer file containing access path and record selection data.
ANZPFRDTA	QSYS	QAPMxxxx	PF	QAPMxxxx performance data collection model files. See the Collection Services data files topic for the names and descriptions of the model files. [2]
	QPFR	QPAVPRT	PRTF	Performance printer file containing the advisor report. [1]
ANZPGM	QPFR	QAPTAZPD	PF	Performance data collection file: analyze application programs data.
	QPFR	QPPTANZP	PRTF	Performance printer file showing program-to-file and file-to-program relationships.
APYJRNCHG	QSYS	QAJRNCHG	PF	Model output file for apply journaled changes.
ASKQST	QSYS	QPQAPRT	PRTF	Q & A printer file.
CFGDSTSRV	QUSRSYS	QASNADSA	PF	SNADS secondary node ID table.
	QUSRSYS	QASNADSQ	PF	SNADS destination queues table.
	QUSRSYS	QASNADSR	PF	SNADS destination systems routing table.
	QUSRSYS	QATOCIFC	PF	TCP/IP interface file.
	QUSRSYS	QATOCPORT	PF	TCP/IP port restrictions file.
	QUSRSYS	QATOCPS	PF	TCP/IP services file.
	QUSRSYS	QATOCRSI	PF	TCP/IP RSI file.
	QUSRSYS	QATOCRTE	PF	TCP/IP route file.

Table 22. Files used by CL commands (part 1) (continued)

Command name	File library	File name	File type	File usage
	QUSRSYS	QATOCTCPIP	PF	TCP/IP attributes file.
CFGTCPSMTP	QUSRSYS	QATMSMTP	PF	TCP/IP SMTP file.
	QUSRSYS	QATMSMTPA	PF	TCP/IP SMTP file.
CHGDTA	QIDU	QDTALOG	PRTF	Audit control log printer file.
	QIDU	QDTAPRT	PRTF	Record and accumulator total printer file.
	QSYS	QPDZDTALOG	PRTF	DFU runtime audit log.
	QSYS	QPDZDTAPRT	PRTF	DFU runtime printer data file.
CHGDSTQ	QUSRSYS	QASNADSQ	PF	SNADS distribution queues table.
CHGDSTRTE	QUSRSYS	QASNADSQ	PF	SNADS distribution queues table.
	QUSRSYS	QASNADSR	PF	SNADS routing table.
CHGFTPA	QUSRSYS	QATMFTP	PF	TCP/IP FTP configuration file.
CHGHTTPA	QUSRSYS	QATMHTTP	PF	TCP/IP HTTP file.
CHGLPDA	QUSRSYS	QATMLPD	PF	TCP/IP LPD configuration file.
CHGPOPA	QUSRSYS	QATMPOPA	PF	POP server configuration file.
CHGPRB	QUSRSYS	QASXNOTE	PF	Problem log user notes file.
	QUSRSYS	QASXPROB	PF	Problem log problem file.
CHGQSTDB	QSYS	QPQAPRT	PRTF	Q & A printer file.
CHGSMTPA	QUSRSYS	QATMSMTP	PF	TCP/IP SMTP configuration file.
CHGTAPCTG	QUSRSYS	QATAMID	PF	Cartridge ID database file.
	QUSRSYS	QLTAMID	LF	Cartridge ID logical file.
	QSYS	QSYSTAP	TAPF	Tape device file used for input.
CHGTELNA	QUSRSYS	QATMTELN	PF	TCP/IP TELNET configuration file.
CHKTAP	QSYS	QSYSTAP	TAPF	Tape device file used for input.
CMPJRNIMG	QSYS	QPCMPIMG	PRTF	Compare journal images printer file.
CPYF	QSYS	QSYSPRT	PRTF	Copy file printer file. [1]
CPYFRMQRYF	QSYS	QSYSPRT	PRTF	Copy file printer file. [1]
CPYFRMTAP	QSYS	QSYSPRT	PRTF	Copy file printer file. [1]
	QSYS	QSYSTAP	TAPF	Tape device file used for input and output.
CPYPFRCOL	QPFR	QAPGSUMD	PF	Performance data collection file for graphics data.
	QSYS	QAPMxxxx	PF	QAPMxxxx performance data collection model files. See the Collection Services data files topic for the names and descriptions of the model files. [2]

Table 22. Files used by CL commands (part 1) (continued)

Command name	File library	File name	File type	File usage
	QSYS	QAPYDWxxxx	PF	Disk Watcher performance data model files. See the Disk Watcher data files topic for the names and descriptions of the model files. [2]
	QSYS	QAPYJWxxxx	PF	Job Watcher performance data model files. [2]
	QSYS	QAYPExxxx	PF	Performance Explorer (PEX) data model files. [2]
	QPFR	QAPTLCKD	PF	Performance data collection file: lock and seizure conflict data.
CPYSRCF	QSYS	QSYSPRT	PRTF	Copy file printer file. [1]
CRTBNDC	QGPL	QCSRC	PF	ILE C source default input file.
CRTBNDCBL	QGPL	QCBLLESRC	PF	ILE COBOL source default input file.
	QSYS	QSYSPRT	PF	ILE COBOL source listing printer file.
CRTBNDCL	QGPL	QCLSRC	PF	CL source default input file.
	QSYS	QSYSPRT	PRTF	CL source listing printer file. [1]
CRTBNDRPG	QGPL	QRPGLESRC	PF	ILE RPG source default input file
	QSYS	QSYSPRT	PRTF	ILE RPG source listing printer file.
CRTCBLMOD	QGPL	QCBLLESRC	PF	ILE COBOL source default input file.
	QSYS	QSYSPRT	PF	ILE COBOL source listing printer file.
CRTCBLPGM	QGPL	QLBLSRC	PF	OPM COBOL source default input file.
	QSYS	QSYSPRT	PRTF	OPM COBOL source listing printer file. [1]
CRTCLD	QGPL	QCLDSRC	PF	C locale description default source input file.
	QSYS	QSYSPRT	PRTF	C locale description source listing printer file. [1]
CRTCLMOD	QGPL	QCLSRC	PF	CL source default input file.
	QSYS	QSYSPRT	PRTF	CL source listing printer file. [1]
CRTCLPGM	QGPL	QCLSRC	PF	CL source default input file.
	QSYS	QSYSPRT	PRTF	CL source listing printer file. [1]
CRTCMD	QGPL	QCMDSRC	PF	Command definition source default input file.
	QSYS	QSYSPRT	PRTF	Command definition source listing printer file.
CRTCMOD	QGPL	QCSRC	PF	ILE C source default input file.
	QSYS	QSYSPRT	PRTF	ILE C source listing printer file. [1]
CRTDSPF	QGPL	QDDSSRC	PF	DDS source default input file.

Table 22. Files used by CL commands (part 1) (continued)

Command name	File library	File name	File type	File usage
	QSYS	QPDDSSRC	PRTF	DDS source listing printer file. [1]
CRTICFF	QGPL	QDDSSRC	PF	DDS source default input file.
	QSYS	QPDDSSRC	PRTF	DDS source listing printer file. [1]
CRTLF	QGPL	QDDSSRC	PF	DDS source default input file.
	QSYS	QPDDSSRC	PRTF	DDS source listing printer file. [1]
CRTMNU	QGPL	QMNUSRC	PF	Default menu source input file.
	QSYS	QSYSPRT	PRTF	Menu source listing printer file. [1]
CRTMSGFMNU	QGPL	QDDSSRC	PF	DDS source created for menu by $BMENU.
	QGPL	QS36DDSSRC	PF	DDS source created for menu by $BMENU.
	QSSP	QPUTMENU	PRTF	$BMENU source listing printer file.
	QSYS	QSYSPRT	PRTF	Pascal source listing printer file. [1]
CRTPF	QGPL	QDDSSRC	PF	DDS source default input file.
	QSYS	QPDDSSRC	PRTF	DDS source listing printer file. [1]
CRTPFRDTA	QSYS	QAPMxxxx	PF	QAPMxxxx performance data collection model files. See the Collection Services data files topic for the names and descriptions of the model files. [2]
CRTPGM	QSYS	QSYSPRT	PRTF	Program source listing printer file. [1]
CRTPNLGRP	QGPL	QPNLSRC	PF	Default panel group source input file.
	QSYS	QSYSPRT	PRTF	Panel group source listing printer file. [1]
CRTPRTF	QGPL	QDDSSRC	PF	DDS source default input file.
	QSYS	QPDDSSRC	PRTF	DDS source listing printer file. [1]
CRTQSTDB	QSYS	QAQA00xxxx	LF	Q & A database model files. [2]
	QSYS	QAQA00xxxx	PF	Q & A database model files. [2]
CRTQSTLOD	QSYS	QPQAPRT	PRTF	Q & A printer file.
CRTRJEBSCF	QRJE	QRJESRC	PF	DDS source file used to create RJE BSC file.
CRTRJECFG	QRJE	QRJESRC	PF	DDS source file used to create RJE BSC or RJE communications file.
CRTRJECMNF	QRJE	QRJESRC	PF	DDS source file used to create RJE communications file.
CRTRPGMOD	QGPL	QRPGLESRC	PF	ILE RPG source default input file.
	QSYS	QSYSPRT	PRTF	ILE RPG source listing printer file. [1]
CRTRPGPGM	QGPL	QRPGSRC	PF	RPG source default input file.

Table 22. Files used by CL commands (part 1) (continued)

Command name	File library	File name	File type	File usage
	QSYS	QSYSPRT	PRTF	RPG source listing printer file. [1]
CRTRPTPGM	QGPL	QRPGSRC	PF	RPG source default input file.
	QSYS	QSYSPRT	PRTF	RPG source listing printer file. [1]
CRTSQLC	QGPL	QCSRC	PF	SQL C source file.
	QSYS	QSYSPRT	PRTF	SQL C printer file. [1]
CRTSQLCI	QGPL	QCSRC	PF	SQL C source file.
	QSYS	QSYSPRT	PRTF	SQL C printer file. [1]
CRTSQLCBL	QGPL	QLBLSRC	PF	SQL COBOL source file.
	QSYS	QSYSPRT	PRTF	SQL COBOL printer file. [1]
CRTSQLCBLI	QGPL	QCBLLESRC	PF	SQL COBOL source file.
	QSYS	QSYSPRT	PRTF	SQL COBOL printer file. [1]
CRTSQLPLI	QGPL	QPLISRC	PF	SQL PLI source file.
	QSYS	QSYSPRT	PRTF	SQL PLI printer file. [1]
CRTSQLRPG	QGPL	QRPGSRC	PF	SQL RPG source file.
	QSYS	QSYSPRT	PRTF	SQL RPG printer file. [1]
CRTSQLRPGI	QGPL	QRPGLESRC	PF	SQL RPG source file.
	QSYS	QSYSPRT	PRTF	SQL RPG printer file. [1]
CRTSRVPGM	QSYS	QSYSPRT	PRTF	Service program source listing printer file. [1]
CRTS36CBL	#LIBRARY	QS36SRC	PF	S/36-compatible COBOL source default input file.
	QSYS	QSYSPRT	PRTF	S/36-compatible COBOL source listing printer file. [1]
CRTS36DSPF	QGPL	QDDSSRC	PF	DDS source created for display file by $SFGR.
	QGPL	QS36DDSSRC	PF	DDS source created for display file by $SFGR.
	QSSP	QPUTSFGR	PRTF	$SFGR source listing printer file.
CRTS36MNU	QGPL	QDDSSRC	PF	DDS source created for menu by $BMENU.
	QGPL	QS36DDSSRC	PF	DDS source created for menu by $BMENU.
	QSSP	QPUTMENU	PRTF	$BMENU source listing printer file.
CRTS36RPG	#LIBRARY	QS36SRC	PF	System/36 RPG II source default input file.
	QSYS	QSYSPRT	PRTF	System/36 RPG II source listing printer file. [1]

Table 22. Files used by CL commands (part 1) (continued)

Command name	File library	File name	File type	File usage
CRTS36RPGR	#LIBRARY	QS36SRC	PF	System/36 RPG II source default input file.
CRTS36RPT	#LIBRARY	QS36SRC	PF	System/36 RPG II Auto Report source default input file.
	QSYS	QSYSPRT	PRTF	System/36 RPG II Auto Report source listing printer file. [1]
CRTTAPCGY	QUSRSYS	QATACGY	PF	Library device database file.
	QUSRSYS	QLTACGY	LF	Library device logical database file.
	QSYS	QSYSTAP	TAPF	Tape device file used for input.
CRTTBL	QGPL	QTBLSRC	PF	Table source default input file.
CVTPFRCOL	QPFR	QAPGSUMD	PF	Performance data collection file for graphics data.
	QSYS	QAPMxxxx	PF	QAPMxxxx performance data collection model files. See the Collection Services data files topic for the names and descriptions of the model files. [2]
	QSYS	QAPYDWxxxx	PF	Disk Watcher performance data model files. See the Disk Watcher data files topic for the names and descriptions of the model files. [2]
	QSYS	QAPYJWxxxx	PF	Job Watcher performance data model files. [2]
	QSYS	QAYPExxxx	PF	Performance Explorer (PEX) data model files. [2]
	QPFR	QAPTLCKD	PF	Performance data collection file: lock and seizure conflict data.
CVTRPGSRC	QSYS	QSYSPRT	PRTF	ILE RPG listing printer file.
	user-lib	QRNCVTLG	PF	ILE RPG conversion log file.
	user-lib	QRPGLESRC	PF	ILE RPG source file (to file).
	user-lib	QRPGSRC	PF	RPG/400® source file (from file).
DLTALR	QUSRSYS	QAALERT	PF	Alert database file.
DLTPFRCOL	QPFR	QAPGSUMD	PF	Performance data collection file for graphics data.
	QSYS	QAPMxxxx	PF	QAPMxxxx performance data collection model files. See the Collection Services data files topic for the names and descriptions of the model files. [2]
	QSYS	QAPYDWxxxx	PF	Disk Watcher performance data model files. See the Disk Watcher data files topic for the names and descriptions of the model files. [2]

Table 22. Files used by CL commands (part 1) (continued)

Command name	File library	File name	File type	File usage
	QSYS	QAPYJWxxxx	PF	Job Watcher performance data model files. [2]
	QSYS	QAYPExxxx	PF	Performance Explorer (PEX) data model files. [2]
	QPFR	QAPTLCKD	PF	Performance data collection file: lock and seizure conflict data.
DLTPRB	QUSRSYS	QASXxxxx	PF	All 8 of the QASXxxxx files for the DLTPRB command are the same subset of files that are shown in the QUSRSYS library for the DSPPRB command.
DLTQST	QSYS	QPQAPRT	PRTF	Q & A printer file.
DLTQSTDB	QSYS	QPQAPRT	PRTF	Q & A printer file.
DLTTAPCGY	QUSRSYS	QATACGY	PF	Library device database file.
	QUSRSYS	QLTACGY	LF	Library device logical database file.
	QSYS	QSYSTAP	TAPF	Tape device file used for input.
DMPCLPGM	QSYS	QPPGMDMP	PRTF	CL program dump printer file.
DMPJOB	QSYS	QPSRVDMP	PRTF	Service dump printer file.
DMPOBJ	QSYS	QPSRVDMP	PRTF	Service dump printer file.
DMPSYSOBJ	QSYS	QPSRVDMP	PRTF	Service dump printer file.
DMPTAP	QSYS	QPTAPDMP	PRTF	Tape dump printer file.
	QSYS	QSYSTAP	TAPF	Tape device file used for input and output.
DMPTRC	QSYS	QAPMDMPT	PF	Performance trace file.
	QSYS	QSYSPRT	PRTF	SDA source printer file.
DSPACC	QSYS	QSYSPRT	PRTF	Display access codes printer file. [1]
DSPACCAUT	QSYS	QSYSPRT	PRTF	Display access code authority printer file. [1]
DSPACTPJ	QSYS	QSYSPRT	PRTF	Display active prestart jobs printer file. [1]
DSPAPPNINF	QUSRSYS	QALSxxx	PF	Set of 4 QALSxxx model database files that contain the record formats used for storing APPN information, where xxx = DIR, END, INM, and TDB. [2]
	QSYS	QSYSPRT	PRTF	Display APPN information printer file. [1]
DSPAUTHLR	QSYS	QADSHLR	PF	Model database file that contains the record format for the authority holder object entries. [2]
	QSYS	QPSYDSHL	PRTF	Display authority holders printer file. [1]

Table 22. Files used by CL commands (part 1) (continued)

Command name	File library	File name	File type	File usage
DSPAUTL	QSYS	QAOBJAUT	PF	Model database file that contains the record format for the authorization list entries. [2]
	QSYS	QPOBJAUT	PRTF	Authorization list entries printer file. [1]
DSPAUTLDLO	QSYS	QSYSPRT	PRTF	Authorization list printer file. [1]
DSPAUTLOBJ	QSYS	QADALO	PF	Model database file that contains the record format for the authorization list object entries. [2]
	QSYS	QPSYDALO	PRTF	Display authorization list objects printer file. [1]
DSPAUTUSR	QSYS	QPAUTUSR	PRTF	Authorized users printer file. [1]
DSPBCKSTS	QSYS	QSYSPRT	PRTF	Display backup status printer file. [1]
DSPBCKUP	QSYS	QSYSPRT	PRTF	Display backup options printer file. [1]
DSPBCKUPL	QSYS	QSYSPRT	PRTF	Display backup list printer file. [1]
DSPBKP	QSYS	QPDBGDSP	PRTF	Breakpoint (debug mode) printer file. [1]
DSPBNDDIR	QSYS	QABNDBND	PF	Model database file that contains the record format for the binding directory entries. [2]
	QSYS	QSYSPRT	PRTF	Displays the contents of the binding directory printer file. [1]
DSPCFGL	QSYS	QPDCCFGL	PRTF	Configuration list printer file. [1]
DSPCHT	QSYS	QPGDDM	PRTF	BGU-defined chart output printer file. [1]
DSPCLS	QSYS	QPDSPCLS	PRTF	Class printer file. [1]
DSPCMD	QSYS	QPCMD	PRTF	Command values printer file. [1]
DSPCNNL	QSYS	QPDCCNNL	PRTF	Connection list printer file. [1]
DSPCNNSTS	QSYS	QSYSPRT	PRTF	Connection status printer file. [1]
DSPCOSD	QSYS	QPDCCOS	PRTF	Class-of-service description printer file. [1]
DSPCSI	QSYS	QSYSPRT	PRTF	Communications side information printer file. [1]
DSPCTLD	QSYS	QPDCCTL	PRTF	Controller description printer file. [1]
DSPDBG	QSYS	QPDBGDSP	PRTF	Debug display (debug mode) printer file. [1]
DSPDBR	QSYS	QADSPDBR	PF	Model database file that defines the record format of the file created to store information about database file relationships. [2]
	QSYS	QPDSPDBR	PRTF	Printer file that contains information about database file relationships. [1]

Table 22. Files used by CL commands (part 1) (continued)

Command name	File library	File name	File type	File usage
DSPDDMF	QSYS	QPDSPDDM	PRTF	Distributed data management (DDM) file listing printer file. [1]
DSPDEVD	QSYS	QPDCDEV	PRTF	Device description printer file. [1]
DSPDIRE	QSYS	QAOSDIRO	PF	Display directory output file: OUTFILFMT(*TYPE1)
	QSYS	QAOSDIRB	PF	Display directory output file: OUTFILFMT(*TYPE2) DETAIL(*BASIC)
	QSYS	QAOSDIRF	PF	Display directory output file: OUTFILFMT(*TYPE2) DETAIL(*FULL)
	QSYS	QAOSDIRX	PF	Display directory output file: OUTFILFMT(*TYPE3) DETAIL(*FULL)
	QSYS	QPDSPDDL	PRTF	Printer file for *full* details of displayed directory entries. [1]
	QSYS	QPDSPDSM	PRTF	Printer file for *basic* details of displayed directory entries. [1]
DSPDLOAUT	QSYS	QSYSPRT	PRTF	Display document library object authority printer file. [1]
DSPDLONAM	QSYS	QSYSPRT	PRTF	Display document library object printer file. [1]
DSPDSTL	QSYS	QAOSDSTO	PF	Distribution list output file.
	QSYS	QPDSPLDL	PRTF	Distribution list *details* printer file. [1]
	QSYS	QPDSPLSM	PRTF	Distribution list *summary* printer file. [1]
DSPDSTCLGE	QSVMSS	QACQFVOF	PF	Output file model for MSS/400 Display Distribution Catalog Entries command. [2]
DSPDSTLOG	QSYS	QPDSTDLG	PRTF	Display distribution log printer file. [1]
DSPDSTSRV	QSYS	QPDSTSRV	PRTF	Distribution services printer file. [1]
	QUSRSYS	QASNADSA	PF	SNADS secondary node ID table.
	QUSRSYS	QASNADSQ	PF	SNADS destination queues table.
	QUSRSYS	QASNADSR	PF	Routing table database file.
DSPDTA	QIDU	QDTAPRT	PRTF	DFU audit control printer file.
	QSYS	QPDZDTALOG	PRTF	DFU runtime audit log.
	QSYS	QPDZDTAPRT	PRTF	DFU runtime printer data file.
DSPDTAARA	QSYS	QPDSPDTA	PRTF	Data area printer file. [1]
DSPDTADCT	QSYS	QPDSPFFD	PRTF	Data dictionary printer file. [1]
DSPEDTD	QSYS	QPDCEDSP	PRTF	Edit description printer file. [1]

Table 22. Files used by CL commands (part 1) (continued)				
Command name	**File library**	**File name**	**File type**	**File usage**
DSPFD	For the DSPFD command, all of the following entries having a file type of PF are physical files (model files that are not actual output files) that define the record formats of created files used to store a specific type of information about a type (or group) of files. Therefore, the common part of each model file's description begins here and the unique part of each description continues under the **File usage** column:			

Table 22. Files used by CL commands (part 1)				
	QSYS	QAFDACCP	PF	...access path file information. [2]
	QSYS	QAFDBASI	PF	...basic file information common to all files. [2]
	QSYS	QAFDBSC	PF	...BSC file and mixed file device attribute information. [2]
	QSYS	QAFDCMN	PF	...communications file and mixed file device attribute information. [2]
	QSYS	QAFDCSEQ	PF	... collating sequence information. [2]
	QSYS	QAFDCST	PF	... constraint relationship information. [2]
	QSYS	QAFDDDM	PF	...distributed data management (DDM) file attribute information. [2]
	QSYS	QAFDDSP	PF	...display file and mixed file display device attribute information. [2]
	QSYS	QAFDICF	PF	...ICF file attribute information. [2]
	QSYS	QAFDJOIN	PF	...join logical file information. [2]
	QSYS	QAFDLGL	PF	...logical file attribute information. [2]
	QSYS	QAFDMBR	PF	...database member information. [2]
	QSYS	QAFDMBRL	PF	...database member list information. [2]
	QSYS	QAFDNGP	PF	... node group information. [2]
	QSYS	QAFDPHY	PF	...physical file attribute information. [2]
	QSYS	QAFDPRT	PF	...printer file attribute information. [2]
	QSYS	QAFDRFMT	PF	...record format information. [2]
	QSYS	QAFDSAV	PF	...save file information. [2]
	QSYS	QAFDSELO	PF	...select/omit information. [2]
	QSYS	QAFDSPOL	PF	...device file spooled information. [2]
	QSYS	QAFDTAP	PF	...tape file attribute information. [2]
	QSYS	QAFDTRG	PF	... trigger information. [2]
	QSYS	QPDSPFD	PRTF	File description printer file. [1]
DSPFFD	QSYS	QADSPFFD	PF	Model database file that defines the record format of the file created to store file field descriptions. [2]

Table 22. Files used by CL commands (part 1) (continued)				
	QSYS	QPDSPFFD	PRTF	File field description printer file. [1]
DSPFLR	QSYS	QADSPDOC	PF	Document list output database file.
	QSYS	QADSPFLR	PF	Folder list output database file.
	QSYS	QPDSPFLR	PRTF	Display folder printer file. [1]
DSPFNTRSCA	QSYS	QPDSPFNT	PRTF	Font resource attributes printer file. [1]
DSPGDF	QSYS	QPGDDM	PRTF	BGU-defined graphics data printer file. [1]
DSPHDWRSC	For the DSPHDWRSC command, all of the following entries having a file type of PF are physical files (model files that are not actual output files) that define the record formats of created files used to store a specific type of hardware resource information. Therefore, the common part of each model file's description begins here and the unique part of each description continues under the **File usage** column:			
	QSYS	QARZDCMN	PF	...communications resources. [2]
	QSYS	QARZDLWS	PF	...local workstation resources. [2]
	QSYS	QARZDPRC	PF	...processor resources. [2]
	QSYS	QARZDTRA	PF	...token-ring local area network (TRLAN) adapter resources. [2]
	QSYS	QARZDSTG	PF	...storage device resources. [2]
	QSYS	QSYSPRT	PRTF	Hardware resources printer file. [1]
DSPHFS	QSYS	QSYSPRT	PRTF	Display hierarchical file systems printer file. [1]
DSPHSTGPH	QPFR	QPPGGPH	PRTF	Display historical graph printer file. [1]
DSPIGCDCT	QSYS	QPDSPDCT	PRTF	DBCS printer file. [1]
DSPJOB	QSYS	QPDSPJOB	PRTF	Display job printer file. [1]
DSPJOBD	QSYS	QPRTJOBD	PRTF	Job description printer file. [1]
DSPJOBLOG	QSYS	QPJOBLOG	PRTF	Job log printer file. [1]
	QSYS	QPJOBLOGO	PRTF	Job log printer file for jobs before Version 2 Release 3 of the AS/400 . [1]
	QSYS	QAMHJLPR	PF	Primary job log model file. [2]
	QSYS	QAMHJLSC	PF	Secondary job log model file. [2]

Table 22. Files used by CL commands (part 1) (continued)				
DSPJRN	All of the following files for the DSPJRN command having a file type of PF are physical files (model files, not actual output files) that define the record formats of files created to store a group of journal entries retrieved and converted from journal receivers. The retrieved group of entries can be a specific type of information or all types of information that was journaled. Each created file stores the retrieved journal entries after they are converted into one of five basic formats (*TYPE1, *TYPE2, *TYPE3, *TYPE4, or *TYPE5) or into the format defined for the specific type of data being retrieved. • *TYPE1: the basic file format is described in the Journal Management topic collection. • *TYPE2: all of *TYPE1 plus the user profile field. • *TYPE3: all of *TYPE2 plus the null value indicators. • *TYPE4: all of *TYPE3 plus the JID, referential integrity, and trigger information. • *TYPE5: all of *TYPE4 plus more information. • Type-dependent format - the format associated with the specific type (as described as follows for the fourth and following files) of information being retrieved. For example, the model file QASY **AF**JE has a unique format for storing all retrieved journal entries related to authority failures (**AF**) on the system. For the DSPJRN PF files listed as follows, the common part of all the model files' descriptions begins here and the unique part of each file's description continues under the **File usage** column:			
	QSYS	QADSPJRN	PF	...a specific type (or all types) of information; stored in the *TYPE1 format. [2]
	QSYS	QADSPJR2	PF	...a specific type (or all types) of information; stored in the *TYPE2 format. [2]
	QSYS	QADSPJR3	PF	...a specific type (or all types) of information; stored in the *TYPE3 format. [2]
	QSYS	QADSPJR4	PF	...a specific type (or all types) of information; stored in the *TYPE4 format. [2]
	QSYS	QADSPJR5	PF	...a specific type (or all types) of information; stored in the *TYPE5 format. [2]
	QSYS	QADXERLG	PF	...DSNX logged errors. [2]
	QSYS	QADXERL4	PF	...DSNX logged errors; stored in the *TYPE4 format. [2]
	QSYS	QADXJRNL	PF	...DSNX logged data. [2]
	QSYS	QADXJRN4	PF	...DSNX logged data; stored in the *TYPE4 format. [2]
	QSYS	QAJBACG	PF	...job accounting. [2]
	QSYS	QAJBACG4	PF	...job accounting; stored in the *TYPE4 format. [2]
	QSYS	QALZALK	PF	...invalid license keys. [2]

Table 22. Files used by CL commands (part 1) (continued)				
	QSYS	QALZALK4	PF	...invalid license keys; stored in the *TYPE4 format. [2]
	QSYS	QALZALL	PF	...usage limit increases. [2]
	QSYS	QALZALL4	PF	...usage limit increases; stored in the *TYPE4 format. [2]
	QSYS	QALZALU	PF	...licensed users that exceed usage limits. [2]
	QSYS	QALZALU4	PF	...licensed users that exceed usage limits; stored in the *TYPE4 format. [2]
	QSYS	QAPTACG	PF	...print job accounting. [2]
	QSYS	QAPTACG4	PF	...print job accounting; stored in the *TYPE4 format. [2]
	QSYS	QAPTACG5	PF	...print job accounting; stored in the *TYPE5 format. [2]
	QSYS	QASYADJE	PF	...changes to auditing attributes. [2]
	QSYS	QASYADJ4	PF	...changes to auditing attributes; stored in the *TYPE4 format. [2]
	QSYS	QASYADJ5	PF	...changes to auditing attributes; stored in the *TYPE5 format. [2]
	QSYS	QASYAFJE	PF	...authority failures. [2]
	QSYS	QASYAFJ4	PF	...authority failures; stored in the *TYPE4 format. [2]
	QSYS	QASYAFJ5	PF	...authority failures; stored in the *TYPE5 format. [2]
	QSYS	QASYAPJE	PF	...use of adopted authority. [2]
	QSYS	QASYAPJ4	PF	...use of adopted authority; stored in the *TYPE4 format. [2]
	QSYS	QASYAPJ5	PF	...use of adopted authority; stored in the *TYPE5 format. [2]
	QSYS	QASYAUJ5	PF	...security attribute changed; stored in the *TYPE5 format. [2]
	QSYS	QASYCAJE	PF	...changes to object authority (authorization list or object). [2]
	QSYS	QASYCAJ4	PF	...changes to object authority (authorization list or object); stored in the *TYPE4 format. [2]
	QSYS	QASYCAJ5	PF	...changes to object authority (authorization list or object); stored in the *TYPE5 format. [2]
	QSYS	QASYCDJE	PF	...command strings. [2]
	QSYS	QASYCDJ4	PF	...command strings; stored in the *TYPE4 format. [2]

Table 22. Files used by CL commands (part 1) (continued)				
	QSYS	QASYCDJ5	PF	...command strings; stored in the *TYPE5 format. [2]
	QSYS	QASYCOJE	PF	...objects created on the system. [2]
	QSYS	QASYCOJ4	PF	...objects created on the system; stored in the *TYPE4 format. [2]
	QSYS	QASYCOJ5	PF	...objects created on the system; stored in the *TYPE5 format. [2]
	QSYS	QASYCPJE	PF	...create, change, and restore user profile operations. [2]
	QSYS	QASYCPJ4	PF	...create, change, and restore user profile operations; stored in the *TYPE4 format. [2]
	QSYS	QASYCPJ5	PF	...create, change, and restore user profile operations; stored in the *TYPE5 format. [2]
	QSYS	QASYCQJE	PF	...changes to *CRQD object. [2]
	QSYS	QASYCQJ4	PF	...changes to *CRQD object; stored in the *TYPE4 format. [2]
	QSYS	QASYCQJ5	PF	...changes to *CRQD object; stored in the *TYPE5 format. [2]
	QSYS	QASYCUJ4	PF	...cluster operation; stored in the *TYPE4 format. [2]
	QSYS	QASYCUJ5	PF	...cluster operation; stored in the *TYPE5 format. [2]
	QSYS	QASYCVJ4	PF	...connection verification; stored in the *TYPE4 format. [2]
	QSYS	QASYCVJ5	PF	...connection verification; stored in the *TYPE5 format. [2]
	QSYS	QASYCYJ4	PF	...Cryptographic configuration; stored in the *TYPE4 format. [2]
	QSYS	QASYCYJ5	PF	...Cryptographic configuration; stored in the *TYPE5 format. [2]
	QSYS	QASYDIJ4	PF	...Directory services; stored in the *TYPE4 format. [2]
	QSYS	QASYDIJ5	PF	...Directory services; stored in the *TYPE5 format. [2]
	QSYS	QASYDOJE	PF	...objects deleted from the system. [2]
	QSYS	QASYDOJ4	PF	...objects deleted from the system; stored in the *TYPE4 format. [2]
	QSYS	QASYDOJ5	PF	...objects deleted from the system; stored in the *TYPE5 format. [2]
	QSYS	QASYDSJE	PF	...resetting the DST security officer password. [2]

Table 22. Files used by CL commands (part 1) (continued)				
	QSYS	QASYDSJ4	PF	...resetting the DST security officer password; stored in the *TYPE4 format. [2]
	QSYS	QASYDSJ5	PF	...resetting the DST security officer password; stored in the *TYPE5 format. [2]
	QSYS	QASYEVJ4	PF	...Environment variable; stored in the *TYPE4 format. [2]
	QSYS	QASYEVJ5	PF	...Environment variable; stored in the *TYPE5 format. [2]
	QSYS	QASYGRJ4	PF	...General purpose audit record; stored in the *TYPE4 format. [2]
	QSYS	QASYGRJ5	PF	...General purpose audit record; stored in the *TYPE5 format. [2]
	QSYS	QASYGSJE	PF	...gives of descriptors. [2]
	QSYS	QASYGSJ4	PF	...gives of descriptors; stored in the *TYPE4 format. [2]
	QSYS	QASYGSJ5	PF	...gives of descriptors; stored in the *TYPE5 format. [2]
	QSYS	QASYIMJ5	PF	...intrusion monitor; stored in the *TYPE5 format. [2]
	QSYS	QASYIPJE	PF	...interprocess communications. [2]
	QSYS	QASYIPJ4	PF	...interprocess communications; stored in the *TYPE4 format. [2]
	QSYS	QASYIPJ5	PF	...interprocess communications; stored in the *TYPE5 format. [2]
	QSYS	QASYIRJ4	PF	...IP rules actions; stored in the *TYPE4 format. [2]
	QSYS	QASYIRJ5	PF	...IP rules actions; stored in the *TYPE5 format. [2]
	QSYS	QASYISJ4	PF	...Internet security management; stored in the *TYPE4 format. [2]
	QSYS	QASYISJ5	PF	...Internet security management; stored in the *TYPE5 format. [2]
	QSYS	QASYJDJE	PF	...changes to the USER parameter of job descriptions. [2]
	QSYS	QASYJDJ4	PF	...changes to the USER parameter of job descriptions; stored in the *TYPE4 format. [2]
	QSYS	QASYJDJ5	PF	...changes to the USER parameter of job descriptions; stored in the *TYPE5 format. [2]
	QSYS	QASYJSJE	PF	...job changes. [2]

Table 22. Files used by CL commands (part 1) (continued)				
	QSYS	QASYJSJ4	PF	...job changes; stored in the *TYPE4 format. [2]
	QSYS	QASYJSJ5	PF	...job changes; stored in the *TYPE5 format. [2]
	QSYS	QASYKFJ4	PF	...Key ring file; stored in the *TYPE4 format. [2]
	QSYS	QASYKFJ5	PF	...Key ring file; stored in the *TYPE5 format. [2]
	QSYS	QASYLDJE	PF	...link/unlink/lookup directory. [2]
	QSYS	QASYLDJ4	PF	...link/unlink/lookup directory; stored in the *TYPE4 format. [2]
	QSYS	QASYLDJ5	PF	...link/unlink/lookup directory; stored in the *TYPE5 format. [2]
	QSYS	QASYMLJE	PF	...mail actions. [2]
	QSYS	QASYMLJ4	PF	...mail actions; stored in the *TYPE4 format. [2]
	QSYS	QASYMLJ5	PF	...mail actions; stored in the *TYPE5 format. [2]
	QSYS	QASYNAJE	PF	...changes to network attributes. [2]
	QSYS	QASYNAJ4	PF	...changes to network attributes; stored in the *TYPE4 format. [2]
	QSYS	QASYNAJ5	PF	...changes to network attributes; stored in the *TYPE5 format. [2]
	QSYS	QASYNDJE	PF	...directory search violations. [2]
	QSYS	QASYNDJ4	PF	...directory search violations; stored in the *TYPE4 format. [2]
	QSYS	QASYNDJ5	PF	...directory search violations; stored in the *TYPE5 format. [2]
	QSYS	QASYNEJE	PF	...end point violations. [2]
	QSYS	QASYNEJ4	PF	...end point violations; stored in the *TYPE4 format. [2]
	QSYS	QASYNEJ5	PF	...end point violations; stored in the *TYPE5 format. [2]
	QSYS	QASYOMJE	PF	...object move and rename operations. [2]
	QSYS	QASYOMJ4	PF	...object move and rename operations; stored in the *TYPE4 format. [2]
	QSYS	QASYOMJ5	PF	...object move and rename operations; stored in the *TYPE5 format. [2]
	QSYS	QASYORJE	PF	...object restore operations. [2]
	QSYS	QASYORJ4	PF	...object restore operations; stored in the *TYPE4 format. [2]

Table 22. Files used by CL commands (part 1) (continued)				
	QSYS	QASYORJ5	PF	...object restore operations; stored in the *TYPE5 format. [2]
	QSYS	QASYOWJE	PF	...changes to object ownership. [2]
	QSYS	QASYOWJ4	PF	...changes to object ownership; stored in the *TYPE4 format. [2]
	QSYS	QASYOWJ5	PF	...changes to object ownership; stored in the *TYPE5 format. [2]
	QSYS	QASYO1JE	PF	...single optical object accesses. [2]
	QSYS	QASYO1J4	PF	...single optical object accesses; stored in the *TYPE4 format. [2]
	QSYS	QASYO1J5	PF	...single optical object accesses; stored in the *TYPE5 format. [2]
	QSYS	QASYO2JE	PF	...dual optical object accesses. [2]
	QSYS	QASYO2J4	PF	...dual optical object accesses; stored in the *TYPE4 format. [2]
	QSYS	QASYO2J5	PF	...dual optical object accesses; stored in the *TYPE5 format. [2]
	QSYS	QASYO3JE	PF	...optical volume accesses. [2]
	QSYS	QASYO3J4	PF	...optical volume accesses; stored in the *TYPE4 format. [2]
	QSYS	QASYO3J5	PF	...optical volume accesses; stored in the *TYPE5 format. [2]
	QSYS	QASYPAJE	PF	...changes to programs (CHGPGM) that will now adopt the owner's authority. [2]
	QSYS	QASYPAJ4	PF	...changes to programs (CHGPGM) that will now adopt the owner's authority; stored in the *TYPE4 format. [2]
	QSYS	QASYPAJ5	PF	...changes to programs (CHGPGM) that will now adopt the owner's authority; stored in the *TYPE5 format. [2]
	QSYS	QASYPGJE	PF	...changes to object primary group. [2]
	QSYS	QASYPGJ4	PF	...changes to object primary group; stored in the *TYPE4 format. [2]
	QSYS	QASYPGJ5	PF	...changes to object primary group; stored in the *TYPE5 format. [2]
	QSYS	QASYPOJE	PF	...printer output actions. [2]
	QSYS	QASYPOJ4	PF	...printer output actions; stored in the *TYPE4 format. [2]
	QSYS	QASYPOJ5	PF	...printer output actions; stored in the *TYPE5 format. [2]
	QSYS	QASYPSJE	PF	...profile swapping. [2]

Table 22. Files used by CL commands (part 1) (continued)				
	QSYS	QASYPSJ4	PF	...profile swapping; stored in the *TYPE4 format. [2]
	QSYS	QASYPSJ5	PF	...profile swapping; stored in the *TYPE5 format. [2]
	QSYS	QASYPWJE	PF	...attempted usage of invalid passwords or user profile names. [2]
	QSYS	QASYPWJ4	PF	...attempted usage of invalid passwords or user profile names; stored in the *TYPE4 format. [2]
	QSYS	QASYPWJ5	PF	...attempted usage of invalid passwords or user profile names; stored in the *TYPE5 format. [2]
	QSYS	QASYRAJE	PF	...restore of objects when authority changes. [2]
	QSYS	QASYRAJ4	PF	...restore of objects when authority changes; stored in the *TYPE4 format. [2]
	QSYS	QASYRAJ5	PF	...restore of objects when authority changes; stored in the *TYPE5 format. [2]
	QSYS	QASYRJJE	PF	...restore of job descriptions that contain user profile names. [2]
	QSYS	QASYRJJ4	PF	...restore of job descriptions that contain user profile names; stored in the *TYPE4 format. [2]
	QSYS	QASYRJJ5	PF	...restore of job descriptions that contain user profile names; stored in the *TYPE5 format. [2]
	QSYS	QASYROJE	PF	...restore of objects when ownership was changed to QDFTOWN. [2]
	QSYS	QASYROJ4	PF	...restore of objects when ownership was changed to QDFTOWN; stored in the *TYPE4 format. [2]
	QSYS	QASYROJ5	PF	...restore of objects when ownership was changed to QDFTOWN; stored in the *TYPE5 format. [2]
	QSYS	QASYRPJE	PF	...restore of programs that adopt their owner's authority. [2]
	QSYS	QASYRPJ4	PF	...restore of programs that adopt their owner's authority; stored in the *TYPE4 format. [2]
	QSYS	QASYRPJ5	PF	...restore of programs that adopt their owner's authority; stored in the *TYPE5 format. [2]
	QSYS	QASYRQJE	PF	...restores of *CRQD object. [2]

Table 22. Files used by CL commands (part 1) (continued)				
	QSYS	QASYRQJ4	PF	...restores of *CRQD object; stored in the *TYPE4 format. [2]
	QSYS	QASYRQJ5	PF	...restores of *CRQD object; stored in the *TYPE5 format. [2]
	QSYS	QASYRUJE	PF	...operations restoring authority for user profiles, using the RSTAUT command. [2]
	QSYS	QASYRUJ4	PF	...operations restoring authority for user profiles, using the RSTAUT command; stored in the *TYPE4 format. [2]
	QSYS	QASYRUJ5	PF	...operations restoring authority for user profiles, using the RSTAUT command; stored in the *TYPE5 format. [2]
	QSYS	QASYRZJE	PF	...changes on primary group restores. [2]
	QSYS	QASYRZJ4	PF	...changes on primary group restores; stored in the *TYPE4 format. [2]
	QSYS	QASYRZJ5	PF	...changes on primary group restores; stored in the *TYPE5 format. [2]
	QSYS	QASYSDJE	PF	...changes to the system distribution directory. [2]
	QSYS	QASYSDJ4	PF	...changes to the system distribution directory; stored in the *TYPE4 format. [2]
	QSYS	QASYSDJ5	PF	...changes to the system distribution directory; stored in the *TYPE5 format. [2]
	QSYS	QASYSEJE	PF	...changes to subsystem routings. [2]
	QSYS	QASYSEJ4	PF	...changes to subsystem routings; stored in the *TYPE4 format. [2]
	QSYS	QASYSEJ5	PF	...changes to subsystem routings; stored in the *TYPE5 format. [2]
	QSYS	QASYSFJE	PF	...actions with spooled files. [2]
	QSYS	QASYSFJ4	PF	...actions with spooled files; stored in the *TYPE4 format. [2]
	QSYS	QASYSFJ5	PF	...actions with spooled files; stored in the *TYPE5 format. [2]
	QSYS	QASYSGJ4	PF	...Asynchronous signals; stored in the *TYPE4 format. [2]
	QSYS	QASYSGJ5	PF	...Asynchronous signals; stored in the *TYPE5 format. [2]
	QSYS	QASYSKJ4	PF	...secure sockets connections; stored in the *TYPE4 format. [2]
	QSYS	QASYSKJ5	PF	...secure sockets connections; stored in the *TYPE5 format. [2]

Table 22. Files used by CL commands (part 1) (continued)				
	QSYS	QASYSMJE	PF	...system management changes. [2]
	QSYS	QASYSMJ4	PF	...system management changes; stored in the *TYPE4 format. [2]
	QSYS	QASYSMJ5	PF	...system management changes; stored in the *TYPE5 format. [2]
	QSYS	QASYSOJE	PF	...server security changes. [2]
	QSYS	QASYSOJ4	PF	...server security changes; stored in the *TYPE4 format. [2]
	QSYS	QASYSOJ5	PF	...server security changes; stored in the *TYPE5 format. [2]
	QSYS	QASYSTJE	PF	...use of system service tools. [2]
	QSYS	QASYSTJ4	PF	...use of system service tools; stored in the *TYPE4 format. [2]
	QSYS	QASYSTJ5	PF	...use of system service tools; stored in the *TYPE5 format. [2]
	QSYS	QASYSVJE	PF	...changes to system values. [2]
	QSYS	QASYSVJ4	PF	...changes to system values; stored in the *TYPE4 format. [2]
	QSYS	QASYSVJ5	PF	...changes to system values; stored in the *TYPE5 format. [2]
	QSYS	QASYVAJE	PF	...changes to access control list. [2]
	QSYS	QASYVAJ4	PF	...changes to access control list; stored in the *TYPE4 format. [2]
	QSYS	QASYVAJ5	PF	...changes to access control list; stored in the *TYPE5 format. [2]
	QSYS	QASYVCJE	PF	...connection starts and ends. [2]
	QSYS	QASYVCJ4	PF	...connection starts and ends; stored in the *TYPE4 format. [2]
	QSYS	QASYVCJ5	PF	...connection starts and ends; stored in the *TYPE5 format. [2]
	QSYS	QASYVFJE	PF	...closes of server files. [2]
	QSYS	QASYVFJ4	PF	...closes of server files; stored in the *TYPE4 format. [2]
	QSYS	QASYVFJ5	PF	...closes of server files; stored in the *TYPE5 format. [2]
	QSYS	QASYVLJE	PF	...exceeding account limit. [2]
	QSYS	QASYVLJ4	PF	...exceeding account limit; stored in the *TYPE4 format. [2]
	QSYS	QASYVLJ5	PF	...exceeding account limit; stored in the *TYPE5 format. [2]
	QSYS	QASYVNJE	PF	...network logons and logoffs. [2]

Table 22. Files used by CL commands (part 1) (continued)				
	QSYS	QASYVNJ4	PF	...network logons and logoffs; stored in the *TYPE4 format. [2]
	QSYS	QASYVNJ5	PF	...network logons and logoffs; stored in the *TYPE5 format. [2]
	QSYS	QASYVOJ4	PF	...actions on validation lists; stored in the *TYPE4 format. [2]
	QSYS	QASYVOJ5	PF	...actions on validation lists; stored in the *TYPE5 format. [2]
	QSYS	QASYVPJE	PF	...network password errors. [2]
	QSYS	QASYVPJ4	PF	...network password errors; stored in the *TYPE4 format. [2]
	QSYS	QASYVPJ5	PF	...network password errors; stored in the *TYPE5 format. [2]
	QSYS	QASYVRJE	PF	...accesses to network resources. [2]
	QSYS	QASYVRJ4	PF	...accesses to network resources; stored in the *TYPE4 format. [2]
	QSYS	QASYVRJ5	PF	...accesses to network resources; stored in the *TYPE5 format. [2]
	QSYS	QASYVSJE	PF	...server session starts and ends. [2]
	QSYS	QASYVSJ4	PF	...server session starts and ends; stored in the *TYPE4 format. [2]
	QSYS	QASYVSJ5	PF	...server session starts and ends; stored in the *TYPE5 format. [2]
	QSYS	QASYVUJE	PF	...changes to network profiles. [2]
	QSYS	QASYVUJ4	PF	...changes to network profiles; stored in the *TYPE4 format. [2]
	QSYS	QASYVUJ5	PF	...changes to network profiles; stored in the *TYPE5 format. [2]
	QSYS	QASYVVJE	PF	...changes to service status. [2]
	QSYS	QASYVVJ4	PF	...changes to service status; stored in the *TYPE4 format. [2]
	QSYS	QASYVVJ5	PF	...changes to service status; stored in the *TYPE5 format. [2]
	QSYS	QASYXDJ5	PF	...directory services extension; stored in the *TYPE5 format. [2]
	QSYS	QASYX0J4	PF	...network authentication; stored in the *TYPE4 format. [2]
	QSYS	QASYX0J5	PF	...network authentication; stored in the *TYPE5 format. [2]
	QSYS	QASYX1J5	PF	...identity token; stored in the *TYPE5 format. [2]
	QSYS	QASYYCJE	PF	...changes to document library objects. [2]

Table 22. Files used by CL commands (part 1) (continued)				
	QSYS	QASYYCJ4	PF	...changes to document library objects; stored in the *TYPE4 format. [2]
	QSYS	QASYYCJ5	PF	...changes to document library objects; stored in the *TYPE5 format. [2]
	QSYS	QASYYRJE	PF	...read operations of document library objects. [2]
	QSYS	QASYYRJ4	PF	...read operations of document library objects; stored in the *TYPE4 format. [2]
	QSYS	QASYYRJ5	PF	...read operations of document library objects; stored in the *TYPE5 format. [2]
	QSYS	QASYZCJE	PF	...changes to objects. [2]
	QSYS	QASYZCJ4	PF	...changes to objects; stored in the *TYPE4 format. [2]
	QSYS	QASYZCJ5	PF	...changes to objects; stored in the *TYPE5 format. [2]
	QSYS	QASYZMJE	PF	...object method access. [2]
	QSYS	QASYZMJ4	PF	...object method access; stored in the *TYPE4 format. [2]
	QSYS	QASYZRJE	PF	...read operations of objects. [2]
	QSYS	QASYZRJ4	PF	...read operations of objects; stored in the *TYPE4 format. [2]
	QSYS	QASYZRJ5	PF	...read operations of objects; stored in the *TYPE5 format. [2]
	QSYS	QATOFIPF	PF	...IP filter rule actions. [2]
	QSYS	QATOFNAT	PF	...IP NAT rule actions. [2]
	QSYS	QATOQQOS	PF	...modification of QoS policies. [2]
	QSYS	QATOSLOG	PF	...SNMP log entries. [2]
	QSYS	QATOSLO4	PF	...SNMP log entries; stored in the *TYPE4 format. [2]
	QSYS	QATOVSOF	PF	...VPN information. [2]
	QSYS	QAWCTPJE	PF	...performance tuning. [2]
	QSYS	QAWCTPJ4	PF	...performance tuning; stored in the *TYPE4 format. [2]
	QSYS	QAZDALLG	PF	...SNADS alert logging. [2]
	QSYS	QAZDCFLG	PF	...configuration changes to SNADS distribution queues table. [2]
	QSYS	QAZDCFL4	PF	...configuration changes to SNADS distribution queues table; stored in the *TYPE4 format. [2]
	QSYS	QAZDERLG	PF	...SNADS logged errors. [2]

Table 22. Files used by CL commands (part 1) (continued)				
	QSYS	QAZDERL4	PF	...SNADS logged errors; stored in the *TYPE4 format. [2]
	QSYS	QAZDJRNL	PF	...SNADS logged data. [2]
	QSYS	QAZDJRN4	PF	...SNADS logged data; stored in the *TYPE4 format. [2]
	QSYS	QAZDRTLG	PF	...changes to SNADS routing and secondary system name tables. [2]
	QSYS	QAZDRTL4	PF	...changes to SNADS routing and secondary system name tables; stored in the *TYPE4 format. [2]
	QSYS	QAZDSYLG	PF	...SNADS miscellaneous logged system-level occurrences. [2]
	QSYS	QAZDSYL4	PF	...SNADS miscellaneous logged system-level occurrences; stored in the *TYPE4 format. [2]
	QSYS	QAZMFCF	PF	...Mail server framework (MSF) configuration changes logging. [2]
	QSYS	QAZMFCF4	PF	...Mail server framework (MSF) configuration changes logging; stored in the *TYPE4 format. [2]
	QSYS	QAZMFER	PF	...Mail server framework (MSF) error logging. [2]
	QSYS	QAZMFER4	PF	...Mail server framework (MSF) error logging; stored in the *TYPE4 format. [2]
	QSYS	QAZMFLG	PF	...Mail server framework (MSF) data logging. [2]
	QSYS	QAZMFLG4	PF	...Mail server framework (MSF) data logging; stored in the *TYPE4 format. [2]
	QSYS	QAZMFSY	PF	...Mail server framework (MSF) system information logging. [2]
	QSYS	QAZMFSY4	PF	...Mail server framework (MSF) system information logging; stored in the *TYPE4 format. [2]
	QSYS	QPDSPJRN	PRTF	Display journal printer file. [1]

Table 22. Files used by CL commands (part 1)				
DSPJRNRCVA	QSYS	QPDSPRCV	PRTF	Journal receiver attributes printer file. [1]
DSPLIB	QSYS	QPDSPLIB	PRTF	Library printer file. [1]
DSPLIBD	QSYS	QPRTLIBD	PRTF	Library description printer file. [1]
DSPLIBL	QSYS	QPRTLIBL	PRTF	Library list printer file. [1]
DSPLIND	QSYS	QPDCLINE	PRTF	Line description printer file. [1]
DSPLOG	QSYS	QPDSPLOG	PRTF	Log display printer file. [1]
DSPMNUA	QSYS	QPDSPMNU	PRTF	Menu attributes printer file. [1]

Table 22. Files used by CL commands (part 1) (continued)				
DSPMOD	For the DSPMOD command, all of the following entries having a file type of files used to store a specific type of information about a type (or group) of files. Therefore, the common part of each model file's description begins here and the unique part of each description continues under the **File usage** column:			
	QSVMSS	QACQSRC	PRTF	...contains sources of example security exit programs. [1]
	QSYS	QABNDMBA	PF	...basic information and compatibility sections. [1]
	QSYS	QABNDMSI	PF	...decompressed size and size limits. [1]
	QSYS	QABNDMEX	PF	...symbols defined in this module and exported to others. [1]
	QSYS	QABNDMIM	PF	...defined symbols that are external to this module. [1]
	QSYS	QABNDMPR	PF	...a list of procedure names and types. [1]
	QSYS	QABNDMRE	PF	...a list of system objects referred to by the module at the time the module is bound into a program or service program. [1]
	QSYS	QABNDMCO	PF	...module copyright information.
	QSYS	QSYSPRT	PRTF	...module printer file. [1]
DSPMODD	QSYS	QPDCMOD	PRTF	Mode description printer file. [1]
DSPMODSTS	QSYS	QPDCMOD	PRTF	Mode status printer file. [1]
DSPMSG	QSYS	QPDSPMSG	PRTF	Message display printer file. [1]
DSPMSGD	QSYS	QPMSGD	PRTF	Message description printer file. [1]
DSPNETA	QSYS	QANFDNTF	PF	Model database file used to define the record format for network file entries. [2]
	QSYS	QPDSPNET	PRTF	Display network attributes printer file. [1]
	QSYS	QPNFNJE	PRTF	Display network job entries printer file. [1]
DSPNWID	QSYS	QPDCNWID	PRTF	Network interface description printer file. [1]
DSPOBJAUT	QSYS	QAOBJAUT	PF	Model database file that defines the record format for the object authority entries. [2]
	QSYS	QPOBJAUT	PRTF	Object authority printer file. [1]
DSPOBJD	QSYS	QADSPOBJ	PF	Model database file that defines the record format for the object description entries. [2]
	QSYS	QPRTOBJD	PRTF	Object description printer file. [1]
DSPOPT	QSYS	QAMODFA	PF	Model output file for file attributes. [2]
	QSYS	QAMODPA	PF	Model output file for directory attributes. [2]

Table 22. Files used by CL commands (part 1) (continued)				
	QSYS	QAMODVA	PF	Model output file for volume attributes. [2]
	QSYS	QPSRODSP	PRTF	Printer file for save/restore information. [1]
	QSYS	QSYSPRT	PRTF	Printer file for optical information. [1]
DSPOVR	QSYS	QPDSPOVR	PRTF	Display overrides printer file. [1]
DSPPDGPRF	QGPL	QPCJPDGPRF	PRTF	Printer file for print descriptor group profile. [1]
DSPPFRGPH	QPFRDATA	QAPGGPHF	PF	Performance database file: graph format data.
	QPFRDATA	QAPGPKGF	PF	Performance database file: graph package data.
	QPFR	QPPGGPH	PRTF	Performance graphs printer file. [1]
DSPPGM	QSYS	QPDPGM	PRTF	Display program printer file. [1]
DSPPGMADP	QSYS	QADPGMAD	PF	Model database file that defines the record format of the file created to store the names of programs that adopt the specified profile. [2]
	QSYS	QPPGMADP	PRTF	Printer file that lists the programs that adopt the specified profile. [1]
DSPPGMREF	QSYS	QADSPPGM	PF	Model database file that defines the record format of the file created to store program references. [2]
	QSYS	QPDSPPGM	PRTF	Printer file that contains program references. [1]
DSPPGMVAR	QSYS	QPDBGDSP	PRTF	Program variable (debug mode) printer file. [1]
DSPPRB	All 8 of the QASXxxxx files shown as follows in the QUSRSYS library are also used by the DLTPRB and WRKPRB commands. The following other files (in QSYS) are not used by those commands.			
	QSYS	QASXxxxx	PF	Set of 5 QASXxxxx model database files that contain the layouts for problem output files, where xxxx = CAOF, FXOF, PBOF, SDOF, and TXOF. [2]
	QSYS	QSXPRTD	PRTF	Problem log *detail* printer file. [1]
	QSYS	QSXPRTL	PRTF	Problem log *summary* printer file. [1]
[3] The following 8 files are also used by the DLTPRB and WRKPRB commands.				
	QUSRSYS	QASXCALL [3]	PF	Problem log call override file.
	QUSRSYS	QASXDTA [3]	PF	Problem log data identifier file.
	QUSRSYS	QASXEVT [3]	PF	Problem log event file.
	QUSRSYS	QASXFRU [3]	PF	Problem log possible cause file.
	QUSRSYS	QASXNOTE [3]	PF	Problem log user notes file.

Table 22. Files used by CL commands (part 1) (continued)				
	QUSRSYS	QASXPROB [3]	PF	Problem log problem file.
	QUSRSYS	QASXPTF [3]	PF	Problem log PTF file.
	QUSRSYS	QASXSYMP [3]	PF	Problem log symptom string file.
DSPPTF	QSYS	QADSPPTF	PF	Model database file that defines the record format of the file created to store program temporary fix (PTF) information. [2]
	QSYS	QSYSPRT	PRTF	Display programming temporary fix (PTF) printer file. [1]
DSPPWRSCD	QSYS	QSYSPRT	PRTF	Display power schedule printer file. [1]
DSPRCDLCK	QSYS	QPDSPRLK	PRTF	Record locks display printer file. [1]
DSPRCYAP	QSYS	QSYSPRT	PRTF	Display recovery for access paths printer file.
DSPRDBDIRE	QSYS	QSYSPRT	PRTF	Distributed relational database directory printer file. [1]
	QSYS	QADSPDE	PF	Model database file that defines the record format for the RDB directory entries.
DSPRJECFG	QRJE	QPRTCFGC	PRTF	RJE configuration printer file. [1]
DSPSAVF	QSYS	QPSRODSP	PRTF	Printer file for save file save/restore information. [1]
DSPSBSD	QSYS	QPRTSBSD	PRTF	Subsystem description printer file. [1]
DSPSFWRSC	QSYS	QARZLCOF	PF	Model database file of the file created to store information about IBM licensed programs and SystemView packaged applications. [2]
	QSYS	QSYSPRT	PRTF	Software resources printer file. [1]
DSPSOCSTS	QSYS	QSYSPRT	PRTF	Sphere of control status printer file. [1]
	QUSRSYS	QAALSOC	PF	Sphere of control database file.
DSPSRVPGM	QSYS	QSYSPRT	PRTF	Service program printer file. [1]
DSPSYSSTS	QSYS	QPDSPSTS	PRTF	Display system status printer file. [1]
DSPSYSVAL	QSYS	QPDSPSVL	PRTF	System value printer file. [1]
DSPTAP	QSYS	QATADOF	PF	Model output file for tape output. [2]
	QSYS	QPTAPDSP	PRTF	Printer file for tape output. [1]
	QSYS	QPSRODSP	PRTF	Printer file for tapes in save/restore format. [1]
	QSYS	QSYSTAP	TAPF	Tape device file for input.
DSPTAPCGY	QSYS	QTAPCGY	PRTF	Printer file for tape categories 1.
	QSYS	QATACOF	PF	Model output file for tape categories 2.
	QUSRSYS	QATACGY	PF	Library device database file.

Table 22. Files used by CL commands (part 1) (continued)				
	QUSRSYS	QLTACGY	LF	Library device logical database file.
DSPTAPCTG	QSYS	QPTACTG	PRTF	Printer file for tape cartridge identifiers 1.
	QSYS	QATAVOF	PF	Model output file for tape cartridge identifiers 2.
	QUSRSYS	QATAMID	PF	Library device database file.
	QUSRSYS	QLTAMID	LF	Library device logical database file.
DSPTAPSTS	QSYS	QPTAPSTS	PRTF	Printer file for tape library
	QSYS	QATAIOF	PF	Model output file for tape
	QSYS	QSYSTAP	TAPF	Tape device file for input
DSPTRC	QSYS	QPDBGDSP	PRTF	Trace (debug mode) printer file. [1]
DSPTRCDTA	QSYS	QPDBGDSP	PRTF	Trace data (debug mode) printer file. [1]
DSPUSRPMN	QSYS	QSYSPRT	PRTF	Display document authority printer file. [1]
DSPUSRPRF	QSYS	QADSPUPA	PF	Model database file that defines the record format of user profiles for TYPE(*OBJAUT). [2]
	QSYS	QADSPUPB	PF	Model database file that defines the record format of user profiles for TYPE(*BASIC). [2]
	QSYS	QADSPUPO	PF	Model database file that defines the record format of user profiles for TYPE(*OBJOWN). [2]
	QSYS	QPUSRPRF	PRTF	User profile printer file. [1]
DSPWSUSR	QSYS	QSYSPRT	PRTF	Workstation user printer file. [1]
DUPTAP	QSYS	QSYSTAP	TAPF	Tape device file used for input and output.
EDTIGCDCT	QSYS	QPDSPDCT	PRTF	DBCS printer file.
EDTQST	QSYS	QPQAPRT	PRTF	Q & A printer file.
EJTEMLOUT	QSYS	QPEMPRTF	PRTF	Emulation printer file.
ENDJOBTRC	QPFR	QAPTDDS	PF	Performance data DDS source file.
	QPFR	QAPTTRCJ	PF	Performance data collection file: job trace data.
	QPFR	QPPTTRCD	PRTF	Performance printer file containing job trace analysis *detail* data.
	QPFR	QPPTTRC1	PRTF	Performance printer file containing job trace analysis *summary* data of physical disk activity.
	QPFR	QPPTTRC2	PRTF	Performance printer file containing job trace analysis I/O *summary* data.
ENDPRTEML	QSYS	QPEMPRTF	PRTF	Emulation printer file.

Table 22. Files used by CL commands (part 1) (continued)				
	QPFR	QAPTSAMH	PF	Performance data collection file: high-level sampled address monitor (SAM) data.
	QPFR	QAPTSAMV	PF	Performance data collection file: low-level sampled address monitor (SAM) data.
FMTDTA	QSYS	QSYSPRT	PRTF	Data format printer file.
	QGPL	QFMTSRC	PF	Sort source default input file.
FNDSTRPDM	QPDA	QPUOPRTF	PRTF	PDM printer output file for user's find string requests.
HLDDSTQ	QUSRSYS	QASNADSQ	PF	SNADS distribution queues table.
INZTAP	QSYS	QSYSTAP	TAPF	Tape device file used for input and output.
LODQSTDB	QQALIB	QAQAxxxx00	PF	Q & A supplied model database files. [2]
	QSYS	QAQA00xxxx	LF	Q & A database model files. [2]
	QSYS	QAQA00xxxx	PF	Q & A database model files. [2]
	QSYS	QPQAPRT	PRTF	Q & A printer file.
MRGFORMD	QPDA	QPAPFPRT	PRTF	Merge form description printer file.
PRTACTRPT	user-lib	QAITMON	PF	Performance data collection file: job and Licensed Internal Code task data.
	QPFR	QAPTDDS	PF	Performance data DDS source file.
	QPFR	QPITACTR	PRTF	Performance printer file containing job and Licensed Internal Code task data.
PRTAFPDTA	QSYS	QSYSPRT	PRTF	Advanced Function Printing data printer file.
PRTCADMRE	QHASM	QAHAPMRSC	PF	Model database file that contains the record format for the monitored resource entries. [2]
	QHASM	QAHAPMATTR	PF	Model database file that contains the record format for the monitored resource attributes entries. [2]
	QHASM	QPHAPRTMRE	PRTF	Printer file of monitored resources in a cluster administrative domain.
PRTCMDUSG	QSYS	QSYSPRT	PRTF	Command usage printer file.
PRTCMNTRC	QSYS	QASCCMNT	PF	Model database file that defines the record format of the file created to store communications trace records. [2]
	QSYS	QPCSMPRT	PRTF	Communications trace printer file (concurrent service monitor). [1]
PRTCPTRPT	QPFR	QAPTDDS	PF	Performance data DDS source file.
	QPFR	QPPTCPTR	PRTF	Performance printer file containing component-level activity data.

Table 22. Files used by CL commands (part 1) (continued)				
	QSYS	QAPMxxxx	PF	QAPMxxxx performance data collection model files. See the Collection Services data files topic for the names and descriptions of the model files. [2]
PRTDEVADR	QSYS	QPDDEVA	PRTF	Print device address printer file.
PRTDOC	QSYS	QAOPOUFL	PF	Document output database file. [1]
	QSYS	QSYSPRT	PRTF	Print document printer file. [1]
PRTDSKINF	QSYS	QAEZDISK	PF	Model outfile for disk space information.
	QUSRSYS	QAEZDISK	PF	Database input file of disk space information.
	QSYS	QPEZDISK	PRTF	Disk space report printer file.
	QSYS	QSYSPRT	PRTF	Disk space report printer file. This file must be specified if using OVRPRTF.
PRTERRLOG	QSYS	QAPRTELG	PF	Model database file that defines the record format of the file created to store error log records. [2]
	QSYS	QAVOLSTA	PF	Model database file that defines the record format of the file created to store volume statistics. [2]
	QSYS	QPCSMPRT	PRTF	Error log (for concurrent service monitor) printer file. [1]
PRTINTDTA	QSYS	QPCSMPRT	PRTF	Internal data (for concurrent service monitor) printer file.
	QSYS	QSYSTAP	TAPF	Tape device file used for output.
PRTJOBRPT	QPFR	QAPTDDS	PF	Performance data DDS source file.
	QPFR	QPPTITVJ	PRTF	Performance printer file containing job interval collection data.
	QSYS	QAPMxxxx	PF	QAPMxxxx performance data collection model files. See the Collection Services data files topic for the names and descriptions of the model files. [2]
PRTJOBTRC	QPFR	QAJOBTRC	PF	Performance data collection file: job trace data.
	QPFR	QAPTDDS	PF	Performance data DDS source file.
	QPFR	QAPTTRCJ	PF	Performance data collection file: job trace data.
	QPFR	QPPTTRCD	PRTF	Performance printer file containing job trace analysis *detail* data.
	QPFR	QPPTTRC1	PRTF	Performance printer file containing job trace analysis *summary* data of physical disk activity.
	QPFR	QPPTTRC2	PRTF	Performance printer file containing job trace analysis I/O *summary* data.

Table 22. Files used by CL commands (part 1) (continued)				
PRTLCKRPT	user-lib	QAPTLCKD	PF	Performance data collection *output* file containing lock and seizure conflict data.
	QPFR	QAPTDDS	PF	Performance data DDS source file.
	QPFR	QPPTLCK	PRTF	Performance data collection *printer* file containing lock and seizure conflict data.
	QSYS	QAPMDMPT	PF	Performance data collection *input* file containing system lock and seizure conflict trace data. [2]
PRTOPCACT	QSOC	QAOPCACT	PF	Model output file for OptiConnect activity. [2]
	QSYS	QSYSPRT	PRTF	Printer file for OptiConnect activity. [1]
PRTOPCJOB	QSOC	QAOPCJOB PF	PF	Model output file for OptiConnect jobs. [2]
	QSYS	QSYSPRT	PRTF	Printer file for OptiConnect jobs. [1]
PRTPOLRPT	QPFR	QAPTDDS	PF	Performance data DDS source file.
	QPFR	QPPTITVP	PRTF	Performance printer file containing subsystem and pool activity interval data.
	QSYS	QAPMxxxx	PF	QAPMxxxx performance data collection model files. See the Collection Services data files topic for the names and descriptions of the model files. [2]
PRTRSCRPT	QPFR	QAPTDDS	PF	Performance data DDS source file.
	QPFR	QPPTITVR	PRTF	Performance printer file containing disk and communications line activity interval data.
	QSYS	QAPMxxxx	PF	QAPMxxxx performance data collection model files. See the Collection Services data files topic for the names and descriptions of the model files. [2]
	QPFR	QAPTSAMH	PF	Performance data collection file: high-level sampled address monitor (SAM) data.
	QPFR	QAPTSAMV	PF	Performance data collection file: low-level sampled address monitor (SAM) data.
	QPFR	QPPTSAM	PRTF	Printer file of SAM performance data.
PRTSYSRPT	QPFR	QAPTDDS	PF	Performance data DDS source file.
	QPFR	QPPTSYSR	PRTF	Performance printer file containing system workload and resource utilization data.

Table 22. Files used by CL commands (part 1) (continued)				
	QSYS	QAPMxxxx	PF	QAPMxxxx performance data collection model files. See the Collection Services data files topic for the names and descriptions of the model files. [2]
PRTTNSRPT	QPFR	QAPTDDS	PF	Performance data DDS source file.
	QPFR	QPSPDJS	PRTF	Performance printer file containing job *summary* data.
	QPFR	QPSPDTD	PRTF	Performance printer file containing job state *transition* data.
	QPFR	QPSPDTS	PRTF	Performance printer file containing job *transaction* data.
	QSYS	QAPMxxxx	PF	QAPMxxxx performance data collection model files. See the Collection Services data files topic for the names and descriptions of the model files. [2]
PRTTRC	QSYS	QPSRVTRC	PRTF	Job trace printer output file. [1]
PRTTRCRPT	user-lib	QTRTJOBT	PF	Performance data collection *input* file containing batch job trace data.
	QPFR	QAPTDDS	PF	Performance data DDS source file.
	QSYS	QAPMDMPT	PF	Performance data collection input file containing system trace data. [2]
QRYDOCLIB	QSYS	QAOSIQDL	PF	Query document library output file.
QRYDST	QSYS	QAOSILIN	PF	Incoming distribution output file.
	QSYS	QAOSILOT	PF	Outgoing distribution output file.
RCLSTG	QSYS	QPRCLDMP	PRTF	Reclaim dump output listing.
RCVDST	QSYS	QAOSIRCV	PF	Receive incoming mail distribution model database file. [2]
RCVTIEF	QSYS	QPTIRCV	PRTF	Received files summary printer file. [1]
RLSDSTQ	QUSRSYS	QASNADSQ	PF	SNADS distribution queues table.
RMVDSTQ	QUSRSYS	QASNADSQ	PF	SNADS distribution queues table.
	QUSRSYS	QASNADSR	PF	SNADS routing table.
RMVDSTRTE	QUSRSYS	QASNADSQ	PF	SNADS distribution queues table.
	QUSRSYS	QASNADSR	PF	SNADS routing table.
RMVDSTSYSN	QUSRSYS	QASNADSA	PF	SNADS secondary node ID table.
RMVJRNCHG	QSYS	QAJRNCHG	PF	Model output file for remove journaled changes.
RMVNETJOBE	QUSRSYS	QANFNJE	PF	Network job entry database file.
RMVSOCE	QUSRSYS	QAALSOC	PF	Sphere of control file.
RMVTAPCTG	QUSRSYS	QATAMID	PF	Cartridge ID database file.
	QUSRSYS	QLTAMID	LF	Cartridge ID logical database file.

Table 22. Files used by CL commands (part 1) (continued)				
	QUSRSYS	QATACGY	PF	Category database file.
	QUSRSYS	QLTACGY	LF	Category logical file.
RNMTCPHTE	QUSRSYS	QATOCHOST	PF	TCP/IP host file.
RST	QSYS	QSYSTAP	TAPF	Tape device file used for input.
RSTCFG	QSYS	QASRRSTO	PF	Model output file for configuration. [2]
	QSYS	QPSRLDSP	PRTF	Restored objects status printer file. [1]
	QSYS	QSYSTAP	TAPF	Tape device file used for input.
RSTDFROBJ	QSYS	QASRRSTO	PF	Model output file for restored objects. [2]
	QSYS	QPSRLDSP	PRTF	Restored objects status printer file. [1]
RSTDLO	QSYS	QAOJRSTO	PF	Model output file for restored document library objects. [2]
	QSYS	QPRSTDLO	PRTF	Printer file for restored documents and folders. [1]
	QSYS	QSYSTAP	TAPF	Tape device file used for input.
RSTLIB	QSYS	QASRRSTO	PF	Model output file for libraries. [2]
	QSYS	QPSRLDSP	PRTF	Restored objects status printer file. [1]
	QSYS	QSYSTAP	TAPF	Tape device file used for input.
RSTLICPGM	QSYS	QPSRLDSP	PRTF	Restored objects status printer file. [1]
	QSYS	QSYSTAP	TAPF	Tape device file used for input.
RSTOBJ	QSYS	QASRRSTO	PF	Model output file for restored objects. [2]
	QSYS	QPSRLDSP	PRTF	Restored objects status printer file. [1]
	QSYS	QSYSTAP	TAPF	Tape device file used for input.
RSTPFRCOL	QPFR	QAPGSUMD	PF	Performance data collection file for graphics data.
	QSYS	QAPMxxxx	PF	QAPMxxxx performance data collection model files. See the Collection Services data files topic for the names and descriptions of the model files. [2]
	QSYS	QAPYDWxxxx	PF	Disk Watcher performance data model files. See the Disk Watcher data files topic for the names and descriptions of the model files. [2]
	QSYS	QAPYJWxxxx	PF	Job Watcher performance data model files. [2]
	QSYS	QAYPExxxx	PF	Performance Explorer (PEX) data model files. [2]
	QPFR	QAPTLCKD	PF	Performance data collection file: lock and seizure conflict data.
RSTSYSINF	QSYS	QSYSTAP	TAPF	Tape device file used for input.
RSTUSRPRF	QSYS	QASRRSTO	PF	Model output file for user profiles. [2]

Table 22. Files used by CL commands (part 1) (continued)				
	QSYS	QSYSTAP	TAPF	Tape device file used for input.
RTVDOC	QSYS	QAOSIRTV	PF	Retrieve document from document library output file.
RTVDSKINF	QSYS	QAEZDISK	PF	Model file for disk information.
	QUSRSYS	QAEZDISK	PF	Database output file for disk information.
RUNQRY	QSYS	QPQUPRFIL	PRTF	Printer file used for query output. [1]
SAV	QSYS	QSYSTAP	TAPF	Tape device file used for output.
SAVCFG	QSYS	QASAVOBJ	PF	Model output file for saved objects. [2]
	QSYS	QPSAVOBJ	PRTF	Printer file for saved objects. [1]
	QSYS	QSYSTAP	TAPF	Tape device file used for output.
SAVCHGOBJ	QSYS	QASAVOBJ	PF	Model output file for saved objects. [2]
	QSYS	QPSAVOBJ	PRTF	Printer file for saved objects. [1]
	QSYS	QSYSTAP	TAPF	Tape device file used for output.
SAVDLO	QSYS	QAOJSAVO	PF	Model output file for saved documents and folders. [2]
	QSYS	QPSAVDLO	PRTF	Printer file for saved documents and folders. [1]
	QSYS	QSYSTAP	TAPF	Tape device file used for output.
SAVLIB	QSYS	QASAVOBJ	PF	Model output file for saved objects. [2]
	QSYS	QPSAVOBJ	PRTF	Printer file for saved objects. [1]
	QUSRSYS	QSRPNTWK	PRTF	Printer file for saved database file networks.
	QSYS	QSYSTAP	TAPF	Tape device file used for output.
SAVOBJ	QSYS	QASAVOBJ	PF	Model output file for saved objects. [2]
	QSYS	QPSAVOBJ	PRTF	Printer file for saved objects. [1]
	QSYS	QSYSTAP	TAPF	Tape device file used for output.
SAVPFRCOL	QPFR	QAPGSUMD	PF	Performance data collection file for graphics data.
	QSYS	QAPMxxxx	PF	QAPMxxxx performance data collection model files. See the Collection Services data files topic for the names and descriptions of the model files. [2]
	QSYS	QAPYDWxxxx	PF	Disk Watcher performance data model files. See the Disk Watcher data files topic for the names and descriptions of the model files. [2]
	QSYS	QAPYJWxxxx	PF	Job Watcher performance data model files. [2]

Table 22. Files used by CL commands (part 1) (continued)				
	QSYS	QAYPExxxx	PF	Performance Explorer (PEX) data model files. [2]
	QPFR	QAPTLCKD	PF	Performance data collection file: lock and seizure conflict data.
SAVSAVFDTA	QSYS	QASAVOBJ	PF	Model output file for saved objects. [2]
	QSYS	QPSAVOBJ	PRTF	Printer file for saved objects. [1]
	QSYS	QSYSTAP	TAPF	Tape device file used for output.
SAVSECDTA	QSYS	QASAVOBJ	PF	Model output file for saved objects. [2]
	QSYS	QPSAVOBJ	PRTF	Printer file for saved objects. [1]
	QSYS	QSYSTAP	TAPF	Tape device file used for output.
SAVSYS	QSYS	QASAVOBJ	PF	Model output file for saved objects. [2]
	QSYS	QPSAVOBJ	PRTF	Printer file for saved objects. [1]
	QSYS	QSYSTAP	TAPF	Tape device file used for output.
SAVSYSINF	QSYS	QSYSTAP	TAPF	Tape device file used for output.
SBMFNCJOB	QSYS	QDFNDATA	DSPF	Non-ICF finance display file.
	QUSRSYS	QFNDEVTBL	PF	File containing data for device tables.
	QUSRSYS	QFNPGMTBL	PF	File containing data for program tables.
	QUSRSYS	QFNUSRTBL	PF	File containing data for user tables.
SETTAPCGY	QUSRSYS	QATACGY	PF	Category database file.
	QUSRSYS	QLTACGY	LF	Category logical file.
	QSYS	QSYSTAP	TAPF	Tape device file used for input.
SNDDSTQ	QUSRSYS	QASNADSQ	PF	SNADS distribution queues table.
SNDFNCIMG	QSYS	QCRFDWNL	ICFF	ICF file used for communication with 4701 controller.
SNDPTFORD	QGPL	Qnnnnnnn	SAVF	PTF save file, where nnnnnnn is the PTF number.
	QSYS	QESPRTF	PRTF	Printer file for PTF cover letters.
SNDSRVRQS	QGPL	Qnnnnnnn	SAVF	PTF save file, where nnnnnnn is the PTF number.
	QSYS	QESPRTF	PRTF	Printer file for PTF cover letters.
	QUSRSYS	QAEDCDBPF	PF	File containing service contact data.
STRCODE	user-lib	EVFCICFF	ICFF	ICF file used for communication with workstation.
STRCPYSCN	QSYS	QASCCPY	PF	Pattern for copy screen output file.
STRDFU	QSYS	QDFUPRT	PRTF	DFU printer file.
	QSYS	QPDZDTALOG	PRTF	DFU runtime audit log.
	QSYS	QPDZDTAPRT	PRTF	DFU runtime printer data file.

Table 22. Files used by CL commands (part 1) (continued)				
	QUSRSYS	QABBxxxxx	PF	See Note E.
STRPDM	QPDA	QPUOPRTF	PRTF	Printer file for displayed lists in PDM.
STRPRTEML	QSYS	QPEMPRTF	PRTF	Emulation printer file.
STRPRTWTR	QSYS	QPSPLPRT	PRTF	Printer device file used for all printer writing.
STRQMQRY	QSYS	QPQXPRTF	PRTF	Printer file used by Query CPI. [1]
STRQST	QQALIB	QAQAxxxx00	PF	Q & A supplied model database files. [2]
	QSYS	QAQA00xxxx	LF	Q & A database model files. [2]
	QSYS	QAQA00xxxx	PF	Q & A database model files. [2]
	QSYS	QPQAPRT	PRTF	Q & A printer file.
	QUSRSYS	QABBxxxxx	PF	See Note F.
STRSDA	QGPL	QDDSSRC	PF	DDS source default input file.
STRSEU	QGPL	QTXTSRC	PF	SEU source default input file.
	QPDA	QPSUPRTF	PRTF	SEU source member printer file.
STRSST	QSYS	QPCSMPRT	PRTF	Service tools printer file. [1]
	QTY	QATYALMF	PF	Telephony database model file of alarm record formats.
	QUSRSYS	QATYSWTE	PF	Telephony database file of user-created switch entries.
	QTY	QATYCDRF	PF	Telephony database model file of alarm record formats.
	QUSRSYS	QATYSWTE	PF	Telephony database file of user-created switch entries.
	QUSRSYS	QABBADMTB	PF	...administration table.
	QUSRSYS	QABBCAN	PF	...candidates file.
	QUSRSYS	QABBCOX	PF	...context index.
	QUSRSYS	QABBDEX	PF	...external document index identifiers.
	QUSRSYS	QABBDIC	PF	...dictionary.
	QUSRSYS	QABBDIX	PF	...internal document index identifiers.
	QUSRSYS	QABBDOX	PF	...document index table.
	QUSRSYS	QABBFIX	PF	...fragment index.
	QUSRSYS	QABBIQTB	PF	...scheduling queue.
	QUSRSYS	QABBLADN [4]	PF	...list of indexed LADNs (library-assigned document names).
TRCCNN	QSYS	QSYSPRT	PRTF	Connection trace printer file.
[4] This QABBLADN file is not used by the STRRGZIDX and STRIDXMON commands.				
TRCCPIC	QSYS	QACM0TRC	PF	Trace CPI-Communications database model output file. [2]

Table 22. Files used by CL commands (part 1) (continued)				
	QSYS	QSYSPRT	PRTF	Trace CPI-Communications printer file. [1]
TRCICF	QSYS	QAIFTRCF	PF	Trace ICF database output file.
	QSYS	QPIFTRCF	PRTF	Trace ICF printer file. [1]
TRCINT	QSYS	QPCSMPRT	PRTF	Internal trace (for concurrent service monitor) printer file.
	QSYS	QSYSTAP	TAPF	Tape device file used for output.
TRCJOB	QSYS	QATRCJOB	PF	Database file that defines the record format of the file created to store trace records. [2]
	QSYS	QPSRVTRC	PRTF	Job trace printer output file. [1]
UPDDTA	QSYS	QPDZDTALOG	PRTF	DFU runtime audit log.
	QSYS	QPDZDTAPRT	PRTF	DFU runtime printer data file.
VFYLNKLPDA	QSYS	QSYSPRT	PRTF	Verify link supporting LPDA -2 printer file. [1]
WRKACTJOB	QSYS	QPDSPAJB	PRTF	Active jobs display printer file. [1]
WRKALR	QUSRSYS	QAALERT	PF	Alert database file.
	QSYS	QSYSPRT	PRTF	Alert printer file. [1]
WRKCFGSTS	QSYS	QSYSPRT	PRTF	Configuration status printer file. [1]
WRKCMTDFN	QSYS	QPDSPJOB	PRTF	Commitment definition list printer file. [1] The name of the spooled file is QPCMTCTL.
WRKCNTINF	QUSRSYS	QAEDCDBPF	PF	Database file containing contact data.
WRKDDMF	QSYS	QPWRKDDM	PRTF	Distributed data management (DDM) file attributes printer file. [1]
WRKDEVTBL	QUSRSYS	QFNDEVTBL	PF	File containing data for device tables.
WRKDIR	QSYS	QPDSPDDL	PRTF	Directory entry *details* printer file.
	QSYS	QPDSPDSM	PRTF	Directory entry *summary* printer file.
WRKDOCCVN	QUSRSYS	QAO1CRL	LF	Document conversion *logical* file for input or output.
	QUSRSYS	QAO1CVNP	PF	Document conversion *physical* file for input or output.
	QUSRSYS	QAO1DCVN	PRTF	Document conversion printer file.
WRKDPCQ	QSYS	QPDXWRKD	PRTF	Printer file for DSNX/PC queued distribution requests. [1]
WRKDSKSTS	QSYS	QPWCDSKS	PRTF	Disk status printer file. [1]
WRKDSTL	QSYS	QPDSPLDL	PRTF	Distribution list *details* printer file.
	QSYS	QPDSPLSM	PRTF	Distribution list *summary* printer file.
WRKDSTQ	QSYS	QPDSTSTS	PRTF	Distribution status printer file. [1]
	QUSRSYS	QASNADSQ	PF	SNADS destination queues table.

Table 22. Files used by CL commands (part 1) (continued)				
WRKFCT	QRJE	QPDSPFCT	PRTF	Forms control table printer file. [1]
WRKHDWRSC	QSYS	QASUPTEL	PF	Hardware resources locking database file.
WRKHTTPCFG	QUSRSYS	QATMHTTPC	PF	TCP/IP HTTP file.
WRKJOB	QSYS	QPDSPJOB	PRTF	Job display printer file. [1]
WRKJOBQ	QSYS	QPRTSPLQ	PRTF	Job queue printer file (spooling queue). [1]
WRKJOBSCDE	QSYS	QSYSPRT	PRTF	Job schedule entries printer file. [1]
WRKJRNA	QSYS	QPDSPJNA	PRTF	Journal attributes printer file. [1]
	QSYS	QAWRKJRNA	PF	Journal attributes model output file. [2]
WRKLIBPDM	QPDA	QPUOPRTF	PRTF	Printer file for displayed lists in PDM. [1]
WRKMBRPDM	QPDA	QPUOPRTF	PRTF	Printer file for displayed lists in PDM. [1]
WRKMSG	QSYS	QPDSPMSG	PRTF	Printer file for message queue messages. [1]
WRKMSGD	QSYS	QPMSGD	PRTF	Message description printer file.
WRKNAMSMTP	QSYS	QATMSMTP	PF	TCP/IP SMTP personal alias table.
	QSYS	QATMSMTPA	PF	TCP/IP SMTP system alias table.
WRKNETF	QSYS	QANFDNTF	PF	Database file for display network files. [1]
	QSYS	QPNFDNTF	PRTF	Printer file for display network files. [1]
WRKNETJOBE	QUSRSYS	QANFNJE	PF	Database file for network job entries. [1]
	QSYS	QPNFNJE	PRTF	Printer file for network job entries. [1]
WRKOBJLCK	QSYS	QPDSPOLK	PRTF	Object locks display printer file. [1]
WRKOBJPDM	QPDA	QPUOPRTF	PRTF	Printer file for displayed lists in PDM. [1]
WRKOUTQ	QSYS	QPRTSPLQ	PRTF	Output spooling queue printer file. [1]
WRKOUTQD	QSYS	QPDSPSQD	PRTF	Output queue description. [1]
WRKPGMTBL	QUSRSYS	QFNPGMTBL	PF	File containing data for program tables.
WRKPRB	All 8 of the QASXxxxx files shown in the QUSRSYS library for the WRKPRB command are the same subset of files that are shown in the QUSRSYS library for the DSPPRB command.			
	QSYS	QSXPRTD	PRTF	Problem log *detail* printer file.
	QSYS	QSXPRTL	PRTF	Problem log *summary* printer file.
	QUSRSYS	QASXxxxx	PF	See the DSPPRB command for these 8 files in the QUSRSYS library.
WRKQRY	QSYS	QPQUPRFIL	PRTF	Printer file used for query output.
WRKQST	QSYS	QPQAPRT	PRTF	Q & A printer file.
WRKRDBDIRE	QSYS	QSYSPRT	PRTF	Distributed relational database directory printer file. [1]

Table 22. Files used by CL commands (part 1) (continued)				
WRKRDR	QSYS	QPRTRDWT	PRTF	Reader display printer file. [1]
WRKRJESSN	QRJE	QPRJESTS	PRTF	Active status printer file for RJE session. [1]
WRKRPYLE	QSYS	QPRTRPYL	PRTF	System reply list printer file. [1]
WRKSBMJOB	QSYS	QPDSPSBJ	PRTF	Submitted jobs printer file. [1]
WRKSBS	QSYS	QPDSPSBS	PRTF	Subsystem display printer file. [1]
WRKSBSJOB	QSYS	QPDSPSBJ	PRTF	Subsystem jobs display printer file. [1]
WRKSHRPOOL	QSYS	QSYSPRT	PRTF	Shared storage pools printer file. [1]
WRKSOC	QUSRSYS	QAALSOC	PF	Sphere of control database file.
WRKSPLF	QSYS	QPRTSPLF	PRTF	Spooled file printer file. [1]
WRKSPLFA	QSYS	QPDSPSFA	PRTF	Spooled file attributes printer file. [1]
WRKSPTPRD	QSYS	QSYSPRT	PRTF	Supported products printer file. [1]
WRKSRVPVD	QUSRSYS	QAEDSPI	PF	Service provider information file.
WRKSRVRQS	QUSRSYS	QANSSRI	PF	Service requester file.
WRKSSND	QRJE	QPRTSSND	PRTF	Session description printer file. [1]
WRKSYSACT	user-lib	QAITMON	PF	Performance data collection file: job and Licensed Internal Code task data. [1]
	QPFR	QAPTDDS	PF	Performance data DDS source file.
WRKSYSSTS	QSYS	QPDSPSTS	PRTF	System status printer file. [1]
WRKSYSVAL	QSYS	QSYSPRT	PRTF	System values printer file. [1]
WRKTAPCTG	QUSRSYS	QATAMID	PF	Cartridge ID database file.
	QUSRSYS	QLTAMID	LF	Cartridge ID logical database file.
	QUSRSYS	QATACGY	PF	Category database file.
	QUSRSYS	QLTACGY	LF	Category logical file.
	QSYS	QSYSTAP	TAPF	Tape device file for input.
WRKTCPPTP	QUSRSYS	QATOCPTP	PF	TCP/IP point-to-point profile configuration.
	QUSRSYS	QATOCMODEM	PF	TCP/IP point-to-point modem configuration.
WRKTIE	QSYS	QPTIRCV	PRTF	Printer file summary of files received.
WRKTIMZON	QSYS	QSYSPRT	PRTF	Information for selected time zone description.
WRKTRA	QSYS	QSYSPRT	PRTF	Printer file containing list of token-ring network adapters. [1]
WRKUSRJOB	QSYS	QPDSPSBJ	PRTF	User jobs display printer file. [1]
WRKUSRTBL	QUSRSYS	QFNUSRTBL	PF	File containing data for user tables.
WRKWTR	QSYS	QPRTRDWT	PRTF	Writer display printer file. [1]

Related information

Display Problems (DSPPRB) command

CL programming

To program using CL, you must understand the procedures and concepts specific to CL programming.

A *CL source program* is a set of CL source statements that can be compiled into either an original program model (OPM) program or an Integrated Language Environment (ILE) module.

A *CL program* or *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 program or procedure is assigned a name by which it can be called by other procedures or bound into a program and run. As with other kinds of procedures, you must enter, compile, and bind CL source statements before you can run the program or 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 program or procedure, the source remains for later modification if you choose; the commands are compiled into a module if you use the ILE compiler. 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 program or procedure ensure consistent processing of groups of commands. You can perform functions with a CL program or 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 programs and 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 original program model (OPM) programs. CL source statements can be entered in a database source member or an IFS stream file 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 or an IFS stream file. You can then create an Integrated Language Environment (ILE) program by compiling the source member into a module and binding the module into a program object.

CL programs and 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.

These are some of the advantages in using CL programs and procedures for an application:

- Because the commands are stored in a form that can be processed when the program or procedure is created, using programs and procedures is faster than entering and running the commands individually.
- CL programs and procedures are flexible. Parameters can be passed to CL programs and procedures to adapt the operations performed by the program or procedure to the requirements of a particular use.
- CL programs and procedures can be tested and debugged like other high-level language programs and procedures.
- CL programs and procedures 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 programs and procedures for the following purposes:

- To add or update records in database files.
- To use printer or ICF files.
- To use subfiles within display files.
- To use program-described display files.

Related tasks

Using CL or other HLLs for mixed list CL command parameters
When a CL command is run, the elements in a mixed list are passed to the command processing program in this format.

Process for creating a CL program or CL procedure

All programs are created in steps: source creation, module creation, and program creation.

1. **Source creation**. CL source statements consist of CL commands. Source statements are entered into a database file or into an IFS stream 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 high-level language (HLL) languages can 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 original program model (OPM) CL program from the CL source statements, you can use the **Create CL Program (CRTCLPGM)** command.

Related information

Create Program (CRTPGM) command
Create CL Program (CRTCLPGM) command
Create Bound CL Program (CRTCLPGM) command

Interactive entry

Interactive entry allows you to enter commands individually.

The IBM i operating system provides many menus and displays to allow interactive entry, including the Programmer Menu, the Command Entry display, command prompt displays, and the Programming Development Manager (PDM) Menu. If you use the IBM i 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.

Related information

Edit File (EDTF) command

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 Data Base Jobs (SBMDBJOB)** 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 code in a file, 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.

Related information

Submit Data Base Jobs (SBMDBJOB) command

Copy File (CPYF) command

Create CL Module (CRTCLMOD) command

Parts of a CL source program

Although each source statement that is entered as part of a CL source program is actually a CL command, the source can be divided into the basic parts that are used in many typical CL source programs.

PGM command

PGM PARM(&A)

Optional PGM command beginning the source program and identifying any parameters received.

Declare commands

(DCL, DCLF, COPYRIGHT, DCLPRCOPT)

Mandatory declaration of program or procedure variables when variables are used, and optional definition of the size of the subroutine stack. DCLPRCOPT also provides the ability to override compiler processing options specified on the CL command used to invoke the CL compiler. The declare commands must precede all other commands except the PGM command.

INCLUDE command
CL command to embed additional CL source commands at compile time.

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, CALLSUBR, SUBR, RTNSUBR, ENDSUBR`

Commands used to control processing within the CL program or procedure.

Built-in functions
`%SUBSTRING (%SST), %SWITCH, %BINARY (%BIN), %ADDRESS (%ADDR), %OFFSET (%OFS), %CHECK, %CHECKR, %SCAN, %TRIM, %TRIML, %TRIMR, %CHAR, %DEC, %INT, %UINT (%UNS), %LEN, %SIZE, %LOWER, %UPPER, %PARMS`

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

Program control commands
`CALL, RETURN, TFRCTL`

CL commands used to pass control to other programs.

Procedure control commands
`CALLPRC`

CL command to pass control to another procedure.

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 program or procedure can refer to other objects that must exist when the program or procedure is created, when the command is processed, or both. In some circumstances, for your program or procedure to run successfully, you might need the following objects:

- 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.
- A database file. Records in a database file can 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.
- 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 CL program or CL procedure.
- Other procedures. If you use the **CALLPRC** command, the called procedure must exist at the time the Create Bound CL Program (CRTBNDCL) or Create Program (CRTPGM) command is run.

Related tasks
Accessing objects in CL programs
To access an object from a CL program, the object must be in the specified library when the command that refers to it runs.

Working with files in CL programs or procedures

Two types of files are supported in CL procedures and programs: display files and database files.

Controlling flow and communicating between programs and procedures
The **Call Program (CALL)**, **Call Bound Procedure (CALLPRC)**, and **Return (RETURN)** commands pass control back and forth between programs and procedures.

Related information
Create Physical File (CRTPF) command
Create Logical File (CRTLF) command
Create Display File (CRTDSPF) command
Declare File (DCLF) command
Call Program (CALL) command
Call Bound Procedure (CALLPRC) command

Example: Simple CL program

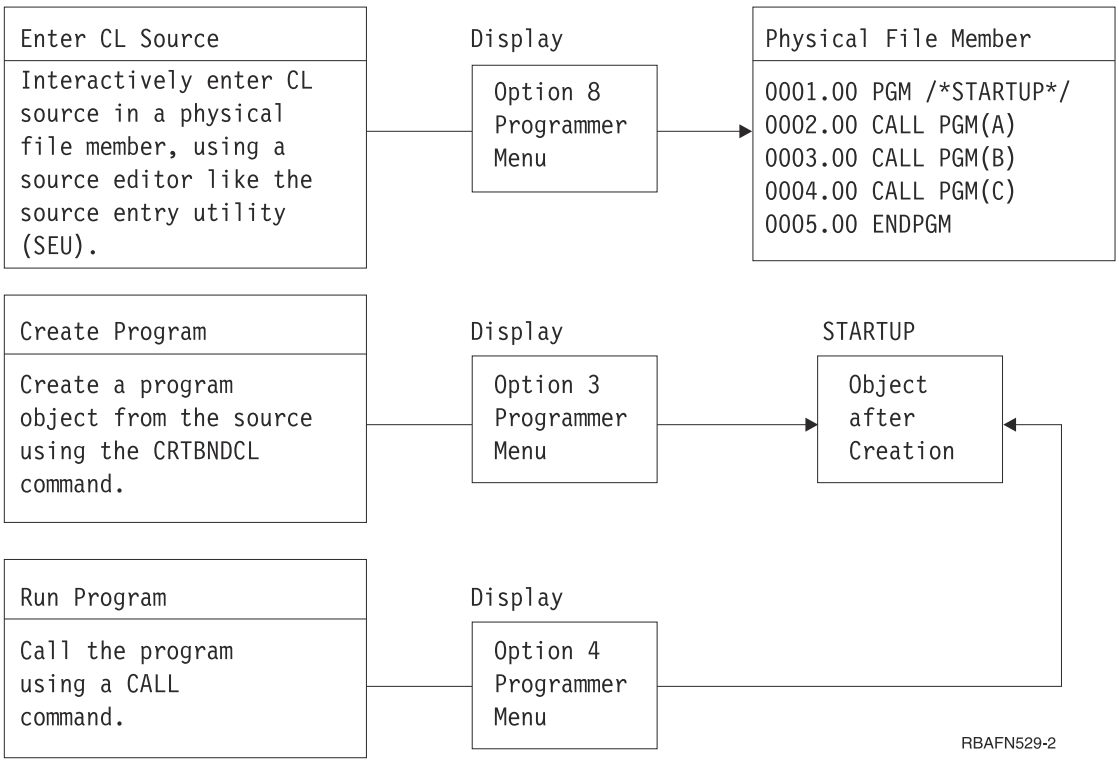
This example is a CL program that calls several programs.

A CL program or procedure 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 called 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 can also use the Programming Development Manager (PDM) or Remote System Explorer (RSE), which are functions of the WebSphere Development Studio for System i licensed program.

To enter, create, and use this program or procedure, follow these steps.



To enter CL source, follow these steps:

1. 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.)
2. Specify CLLE in the Type field and press the Enter key.
3. 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, complete these steps:

1. Press F3 to exit from SEU.
2. Accept the default on the exit display (option 2, Exit and update member) and press the Enter key to return to the Programmer Menu.
3. 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 want to run this example program, however, the referenced programs must exist by the time the CALL commands are run.

Related information

Call Program (CALL) command

Commands used in CL programs or procedures

A CL program or procedure contains only CL commands. These can be IBM-supplied commands or commands defined by you.

You cannot use some IBM-supplied commands in CL programs or procedures. Use the CL command finder or the online help to find CL command descriptions and their applicability in CL programs or procedures.

Commands entered on the RQSDTA and CMD parameters

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

Related information

Transfer Job (TFRJOB) command

Submit Job (SBMJOB) command

Common commands used in CL programs and procedures

The list contains the commands frequently used in CL programs and procedures. You can use this list to select the appropriate command for the function you want.

System function	Command	Command function
Change program or procedure control	**Call (CALL)**	Calling a program.
	Call Procedure (CALLPRC)[1]	Calling a procedure.
	Return (RETURN)	Returning to the command following the command that causes a program or procedure to run.
CL program or procedure limits	**End Program (ENDPGM)**[1]	Indicating the end of a CL source program.
	Program (PGM)[1]	Indicating the start of a CL source program.

System function	Command	Command function
CL program or procedure logic	**Call Subroutine (CALLSUBR)**[1]	Passing control to a subroutine that is defined within the same program or procedure.
	Do (DO)[1]	Indicating the start of a Do group.
	Do For (DOFOR)[1]	Indicating the start of a Do group that processes commands zero or more times based on specified values.
	Do Until (DOUNTIL)[1]	Indicating the start of a Do group that processes a set of commands until the value of a logical expression is true.
	Do While (DOWHILE)[1]	Indicating the start of a Do group that processes a set of commands while the value of a logical expression remains true.
	Else (ELSE)[1]	Defining the action to be taken for the else (false) condition of an IF command.
	End Do (ENDDO)[1]	Indicating the end of a Do group.
	End Select (ENDSELECT)[1]	Indicating the end of a Select group.
	End Subroutine (ENDSUBR)[1]	Ending a subroutine.
	Go To (GOTO)[1]	Branching to another command.
	If (IF)[1]	Processing commands based on the value of a logical expression.
	Iterate (ITERATE)[1]	Ending processing of commands in a Do While, Do Until, or Do For group, and evaluates the group conditions again.
	Leave (LEAVE)[1]	Ending processing of commands in a Do While, Do Until, or Do For group.
	Otherwise (OTHERWISE)[1]	Defining the commands to be processed if no conditions on a When command in a Select group are true.
	Return Subroutine (RTNSUBR)[1]	Exiting a subroutine.
	Subroutine (SUBR)[1]	Delimiting the group of commands which define a subroutine.
	Select (SELECT)[1]	Indicating the start of a Select group, which allows conditional processing of command groups.
	When (WHEN)[1]	Processing commands in a Select group when the value of a logical expression is true.
CL program or procedure variables	**Change Variable (CHGVAR)**[1]	Changing the value of a CL variable.
	Declare (DCL) [1]	Declaring a variable.
Conversion	**Change Variable (CHGVAR)**[1]	Changing the value of a CL variable.
	Convert Date (CVTDAT)[1]	Changing the format of a date.

System function	Command	Command function
Data areas	**Change Data Area (CHGDTAARA)**	Changing a data area.
	Create Data Area (CRTDTAARA)	Creating a data area.
	Delete Data Area (DLTDTAARA)	Deleting a data area.
	Display Data Area (DSPDTAARA)	Displaying a data area.
	Retrieve Data Area (RTVDTAARA)[1]	Copying the content of a data area to a CL variable.
Files	**End Receive (ENDRCV)**[1]	Canceling a request for input previously issued by a RCVF, SNDF, or SNDRCVF command to a display file.
	Declare File (DCLF)[1]	Declaring a display or database file.
	Receive File (RCVF)[1]	Reading a record from a display or database file.
	Retrieve Member Description (RTVMBRD)[1]	Retrieving a description of a specific member of a database file.
	Send File (SNDF)[1]	Writing a record to a display file.
	Send/Receive File (SNDRCVF)[1]	Writing a record to a display file and reads that record after the user has replied.
	Wait (WAIT)[1]	Waiting for data to be received from an SNDF, RCVF, or SNDRCVF command issued to a display file.
Messages	**Monitor Message (MONMSG)**[1]	Monitoring for escape, status, and notify messages sent to a program's message queue.
	Receive Message (RCVMSG)[1]	Copying a message from a message queue into CL variables in a CL program or procedure.
	Remove Message (RMVMSG)[1]	Removing a specified message from a specified message queue.
	Retrieve Message (RTVMSG)[1]	Copying a predefined message from a message file into CL program or procedure variables.
	Send Program Message (SNDPGMMSG)[1]	Sending a program message to a message queue.
	Send Reply (SNDRPY)[1]	Sending a reply message to the sender of an inquiry message.
	Send User Message (SNDUSRMSG)	Sending an informational or inquiry message to a display station or system operator.

System function	Command	Command function
Miscellaneous commands	**Check Object (CHKOBJ)**	Checking for the existence of an object and, optionally, the necessary authority to use the object.
	Include CL Source (INCLUDE)[1]	Embedding CL source commands at compile time.
	Print Command Usage (PRTCMDUSG)	Producing a cross-reference listing for a specified group of commands that are used in a specified group of CL programs or procedures.
	Retrieve Configuration Source (RTVCFGSRC)	Generating CL command source for creating existing configuration objects and placing the source in a source file member.
	Retrieve Configuration Status (RTVCFGSTS)[1]	Giving applications the capability to retrieve configuration status from three configuration objects: line, controller, and device.
	Retrieve Job Attributes (RTVJOBA)[1]	Retrieving the value of one or more job attributes and placing the values in a CL variable.
	Retrieve System Value (RTVSYSVAL)[1]	Retrieving a system value and placing it into a CL variable.
	Retrieve User Profile (RTVUSRPRF)[1]	Retrieving user profile attributes and placing them into CL variables.
Program creation commands	**Create CL Module (CRTCLMOD)**	Creating an integrated language environment (ILE) CL module.
	Delete Module (DLTMOD)	Deleting a module.
	Delete Program (DLTPGM)	Deleting a program.
	Create Bound Control Language Program (CRTBNDCL)	Creating an ILE CL program.
	Create CL Program (CRTCLPGM)	Creating an original program model (OPM) CL program.
	Create Program (CRTPGM)	Creating an ILE program from one or more modules.
	Create Service Program (CRTSRVPGM)	Creating an ILE service program from one or more modules.
[1] Indicates the commands that you can use only in CL programs and procedures.		

Related information

CL command finder

Operations performed by CL programs or procedures

This is an overview of the types of operations that can be performed by using CL programs or procedures.

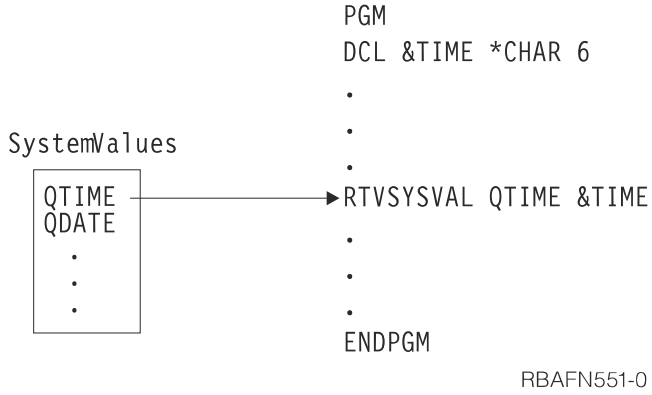
In general, you can perform the following operations:

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

```
PGM
DCL &C *LGL
DCL &A *DEC VALUE(22)
```

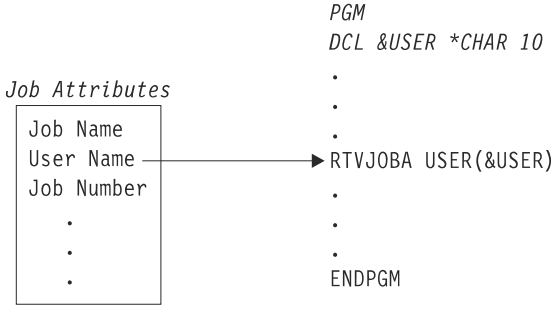
```
DCL &B *CHAR VALUE(ABCDE)
•
•
•
CHGVAR &A (&A + 30)
•
•
•
IF (&A < 50) THEN(CHGVAR &C '1')
•
DSPLIB ('Q' vv &B)
•
IF (%SST(&B 5 1)=E) THEN(CHGVAR &A 12)
•
•
•
ENDPGM
```

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



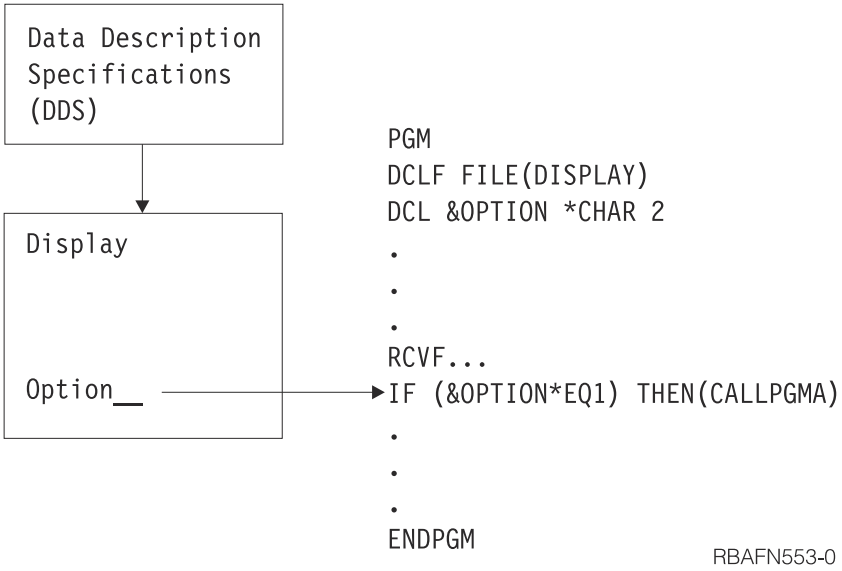
RBAFN551-0

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



RBAFN552-0

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



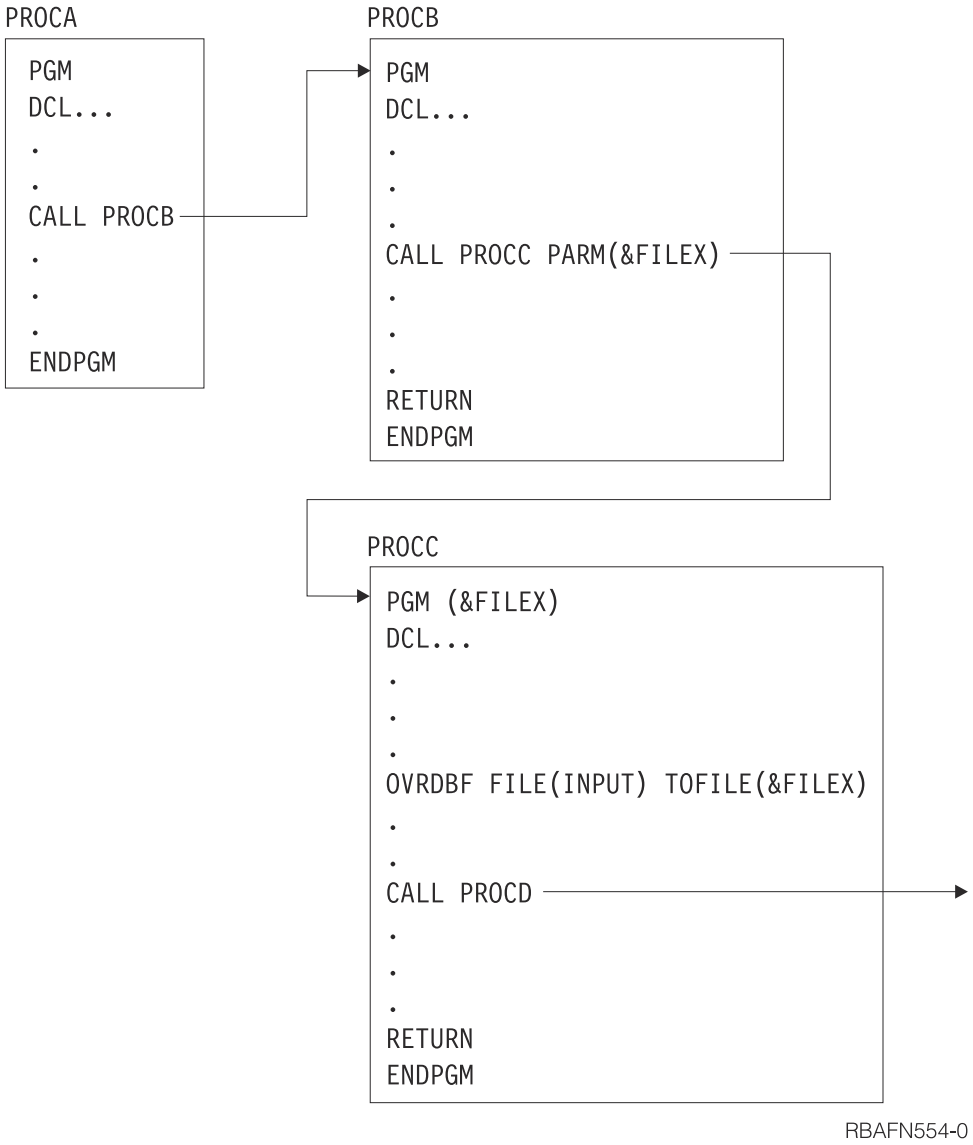
RBAFN553-0

- Create a CL program or procedure to monitor error messages for a job and to 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 program or procedure to other programs or procedures to override files.

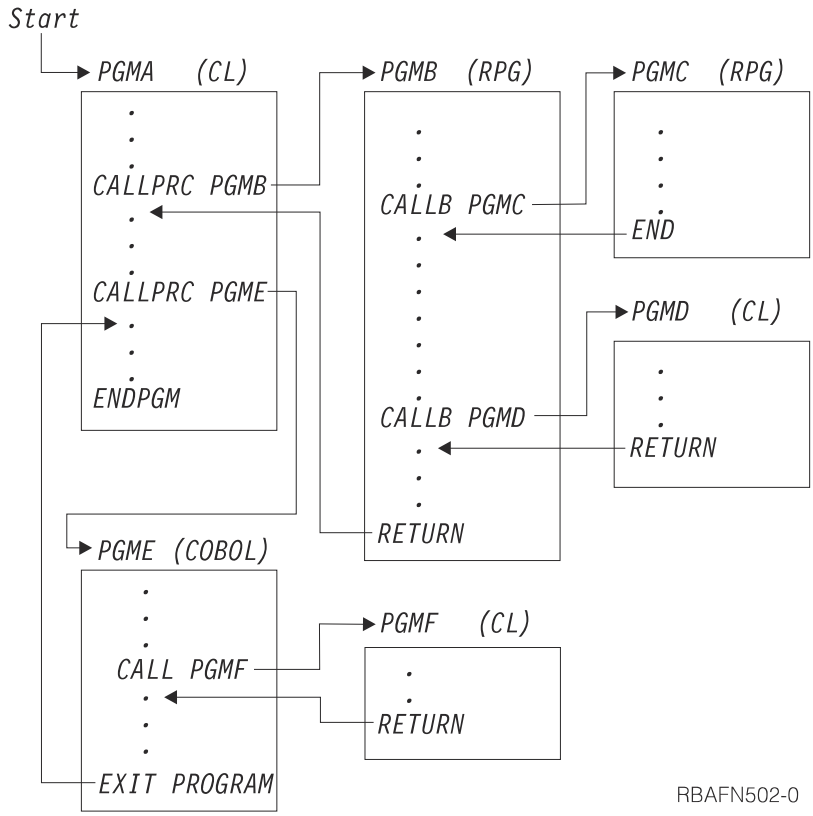


RBAFN554-0

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

- 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)
```

Variables in CL commands

A *variable* is a named changeable value that can be accessed or changed by referring to its name.

A CL source program consists of CL commands. Each command consists of a command name, parameter keyword names, and parameter values. Parameter values can be expressed as variables, constants, or expressions. 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, and they are not objects. Their values are destroyed when the program or 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 can be changed by specific applications. You might, for instance, write a CL source program 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 program or procedure. You can define (specify) the value of the variables when you run the CL program or procedure.

In addition to the uses discussed in this topic, variables can be used to complete the following tasks:

- Passing information between procedures and jobs.
- Passing information between procedures and device displays.
- Conditionally processing commands.
- Creating 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.

It is also possible to assemble the keywords and parameters for a command and process it using the QCAPCMD API or QCMDEXC API.

Related tasks

Controlling flow and communicating between programs and procedures
The **Call Program (CALL)**, **Call Bound Procedure (CALLPRC)**, and **Return (RETURN)** commands pass control back and forth between programs and procedures.

Working with multiple device display files
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 program or procedure, the display file, or different record formats within it, can be sent to several device displays.

Prompting for user input at run time
With most CL programs and procedures, the workstation user provides input by specifying command parameter values that are passed to the program or by typing into input-capable fields on a display prompt.

QCAPCMD program
The Process Commands (QCAPCMD) API performs command analyzer processing on command strings.

QCMDEXC program
The Execute Command (QCMDEXC) API is an IBM-supplied program that runs a single command.

Data type errors using the CALL command
When you use the **Call Program (CALL)** command, data type errors might occur.

Related reference

Controlling processing within a CL program or CL procedure
You can use commands to change the flow of logic within your CL procedure.

Declaring variables to a CL program or procedure

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

There are two ways of declaring variables:

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

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

- **Declare file**. If your CL program or 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.

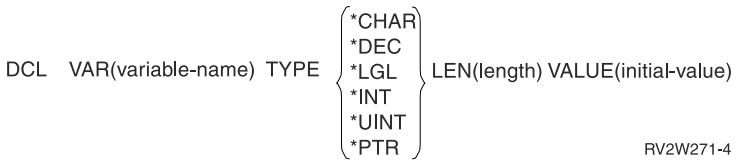
  ```
  DCLF  FILE(MCGANN/GUIDE)
  ```

  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 program or procedure (except the PGM command), but they can be intermixed in any order.

Rules for using the Declare CL Variable command

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, four, or eight bytes. The value can be negative if *INT is specified for the TYPE parameter. The value must be positive or zero if *UINT is specified for the TYPE parameter. LEN(8) can be specified only if the CL source is compiled with the **Create CL Module (CRTCLMOD)** command or the **Create Bound CL Program (CRTBNDCL)** command.
  - A pointer value which can hold the location of data in storage.
- 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.
  - Null for pointer 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.

Related information

Declare CL Variable (DCL) command

Uses for based variables

Based variables can be used to map variables passed to the program or manipulate arrays of values.

The basing pointer must be set using the ADDRESS keyword on the Declare (DCL Command) or with the %ADDRESS built-in function before being used. After the basing pointer is set, the variables will work like local variables.

In the following example, the basing pointer &PTR is declared to be equal to the address of &AUTO. The variable &BASED then has the value of the first 10 bytes addressed by the pointer variable &PTR. Later in the procedure, the value of variable &BASED is checked for equality against the first 10 bytes of variable &AUTO. If the values are the same, meaning pointer &PTR addresses the first byte of &AUTO, the pointer offset is changed to address byte 11 of variable &AUTO. Now variable &BASED has a value equal to bytes 11-20 of variable &AUTO.

Note: By using the code example, you agree to the terms of the "Code license and disclaimer information" on page 610.

```
PGM
DCL &AUTO  *CHAR 20
DCL &PTR   *PTR ADDRESS(&AUTO)
DCL &BASED *CHAR 10 STG(*BASED) BASPTR(&PTR)
:
IF COND(%SST(&AUTO 1 10) *EQ &BASED) +
   THEN(CHGVAR %OFS(&PTR) (%OFS(&PTR) + 10))
:
ENDPGM
```

Related information

Declare CL Variable (DCL) command

Uses for defined variables

Defined variables make it easy to manage complex data structures in control language (CL) by eliminating the need to substring values out of a large variable.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 610.

Defined variables can be used to map different parts of the defined on variable or the same part of a given variable in different ways.

In the following example, the variable &OBJNAME is equal to the first 10 bytes of &OBJECT and the variable &LIBNAME is equal to the last 10 bytes of &OBJECT. Using the defined variables &OBJNAME and &LIBNAME improves the readability of the code and makes it easier to work with. The variable &OBJECT provides the storage for both the &LIBNAME and &OBJNAME variables.

```
PGM
DCL &OBJECT  *CHAR 20
DCL &OBJNAME *CHAR 10 STG(*DEFINED) DEFVAR(&OBJECT)
DCL &LIBNAME *CHAR 10 STG(*DEFINED) DEFVAR(&OBJECT 11)
:
IF COND(&LIBNAME *EQ '*LIBL     ') +
   THEN(...))
:
ENDPGM
```

You can also make the same storage with multiple definitions. In this example, the variables &BINLEN and &CHARLEN both refer to the same 4 bytes of variable &STRUCT. The program can then use the definition that best suits its requirements.

```
PGM
DCL &STRUCT  *CHAR 50
DCL &BINLEN  *INT   4 STG(*DEFINED) DEFVAR(&STRUCT)
```

```
DCL &CHARLEN *CHAR  4 STG(*DEFINED) DEFVAR(&STRUCT)
:
ENDPGM
```

This example shows how a defined variable can be used to change values in a variable. This example also uses the %OFFSET built-in function and a based variable to navigate the library list. This is not the optimal way to do message substitution but illustrates some of the capabilities of defined variables.

```
PGM
DCL &MESSAGE  *CHAR  25 VALUE('LIBRARY NNN IS XXXXXXXXXX')
DCL &SEQUENCE *CHAR   3 STG(*DEFINED) DEFVAR(&MESSAGE  9)
DCL &MSGLIBN  *CHAR  10 STG(*DEFINED) DEFVAR(&MESSAGE 16)
DCL &COUNTER  *INT    2
DCL &LIBL     *CHAR 165
DCL &PTR      *PTR      ADDRESS(&LIBL)
DCL &LIBLNAME *CHAR  10 STG(*BASED) BASPTR(&PTR)
:
RTVJOBA SYSLIBL(&LIBL)
CHGVAR &COUNTER 0
DOFOR &COUNTER FROM(1) TO(15)
   IF (&LIBLNAME *EQ '          ') THEN(LEAVE)
   CHGVAR &SEQUENCE &COUNTER
   CHGVAR &MSGLIBN &LIBLNAME
   SNDPGMMSG  MSGID(CPF9898) MSGF(QSYS/QCPFMSG) MSGDTA(&MESSAGE)
   CHGVAR %OFS(&PTR) (%OFS(&PTR) + 11)
ENDDO
:
ENDPGM
```

Variables to use for specifying a list or qualified name

Variables can be used to specify a list or qualified name.

The value on a parameter can 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.

Related tasks

Defining CL commands

CL commands enable you to request a broad range of functions. You can use IBM-supplied commands, change the default values for command parameters, and define your own commands.

Related reference

%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.

Cases of characters in variables

There are restrictions on the cases used for characters in CL 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 single quotation marks. 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 single quotation marks, it would have been correct, because without the single quotation marks 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 previous paragraph does not take into account the fact that conversion to uppercase is language dependent. Remember to rely on the system to convert values to uppercase can produce unexpected results.

Variables that replace reserved or numeric parameter values

Character variables can be used for some commands to represent a value on the command parameter.

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 can 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 source program 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 required for a specified output queue. The variable &NRESP is a character variable manipulating numeric and predefined values (note the use of single quotation marks). The JOBSEP parameter on the **Change Output Queue (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.

Another alternative for this function is to use the prompter within a CL source program.

Related tasks

Defining CL commands

CL commands enable you to request a broad range of functions. You can use IBM-supplied commands, change the default values for command parameters, and define your own commands.

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, shown in this example:

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

  &INVCMPLT is set to 0.

  You could also use this notation:

  ```
  CHGVAR   &INVCMPLT  0
  ```

- To the value of another variable, shown in this example:

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

&A is set to the value of the variable &B

You could also use this notation:

```
CHGVAR   &A  &B
```

- To the value of an expression after it is evaluated, shown in the following example:

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

The value of &A is increased by 1.

You could also use this notation:

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

- To the value produced by the built-in function %SST, shown in the following example:

```
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, shown in the following example:

```
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:

```
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.

- To the value produced by the built-in function %CHECK:

```
CHGVAR  VAR(&A)  VALUE(%CHECK('0123456789' &B))
```

The value in variable &B is checked and the position of the leftmost character that is not a digit is stored in variable &A. If all the characters in variable &B are digits, a value of zero is stored in variable &A.

- To the value produced by the built-in function %CHECKR:

```
CHGVAR  VAR(&A)  VALUE(%CHECKR('*' &B))
```

The value in variable &B is checked and the position of the rightmost character that is not an asterisk (*) is stored in variable &A. If all the characters in variable &B are asterisks, a value of zero is stored in variable &A.

- To the value produced by the built-in function %SCAN:

```
CHGVAR  VAR(&A)  VALUE(%SCAN('.' &B))
```

The value in variable &B is scanned and the position of the leftmost period (.) character is stored in variable &A. If there are no period characters in variable &B, a value of zero is stored in variable &A.

- To the value produced by the built-in function %TRIM:

```
CHGVAR  VAR(&A)  VALUE(%TRIM(&B '* '))
```

Leading and trailing asterisk (*) and blank characters in variable &B will be trimmed off and the resulting string will be stored in variable &A.

- To the value produced by the built-in function %TRIML:

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

Leading blank characters in variable &B will be trimmed off and the resulting string will be stored in variable &A.

- To the value produced by the built-in function %TRIMR:

```
CHGVAR  VAR(&A)  VALUE(%TRIMR(&B '*'))
```

Trailing asterisk (*) characters in variable &B will be trimmed off and the resulting string will be stored in variable &A.

- To the value produced by the built-in function %CHAR:

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

Variable &B will be converted into character format and the resulting string will be stored in variable &A.

- To the value produced by the built-in function %UPPER:

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

Lowercase letters in variable &B will be converted into uppercase letters and the resulting string will be stored in variable &A.

- To the value produced by the built-in function %SIZE:

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

The number of bytes occupied by variable &B will be stored in variable &A.

- To the value produced by the built-in function %PARMS:

```
CHGVAR  VAR(&A)  VALUE(%PARMS())
```

The number of parameters that were passed to the program will be 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 23. Valid assignments to variables from values

	Logical value	**Character value**	**Decimal value**	**Signed integer value**	**Unsigned integer value**
Logical variable	X				

Table 23. Valid assignments to variables from values (continued)					
	Logical value	**Character value**	**Decimal value**	**Signed integer value**	**Unsigned integer value**
Character variable	X	X	X	X	X
Decimal variable		X	X	X	X
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-aligned 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 of &B is 5 and decimal positions is 2. 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 is a decimal variable, or rounded if it is 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`.

Related reference

%ADDRESS built-in function
The address built-in function (%ADDRESS or %ADDR) can be used to change or test the memory address stored in a CL pointer variable.

%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.

%CHAR built-in function
%CHAR converts logical, decimal, integer, or unsigned integer data to character format. The converted value can be assigned to a CL variable, passed as a character constant to another program or procedure, or specified as a value for a command parameter of a CL command run from compiled CL.

%CHECK built-in function

The check built-in function (%CHECK) returns the first position of a *base string* that contains a character that does not appear in the *comparator string*. If all of the characters in the base string also appear in the comparator string, the function returns 0.

%CHECKR built-in function
The reverse check built-in function (%CHECKR) returns the last position of a *base string* that contains a character that does not appear in the *comparator string*. If all of the characters in the base string also appear in the comparator string, the function returns 0.

%DEC built-in function
%DEC converts character, logical, decimal, integer, or unsigned integer data to packed decimal format. The converted value can be assigned to a CL variable, passed as a numeric constant to another program or procedure, or specified as a value for a command parameter of a CL command run from compiled CL.

%INT built-in function
%INT converts character, logical, decimal, or unsigned integer data to integer format. The converted value can be assigned to a CL variable, passed as a numeric constant to another program or procedure, or specified as a value for a command parameter of a CL command run from compiled CL.

%LEN built-in function
The %LEN built-in function returns the number of digits or characters of the CL numeric or character variable.

%LOWER built-in function
The %LOWER built-in function returns a character string that is the same length as the argument specified with each uppercase letter replaced by the corresponding lowercase letter.

%OFFSET built-in function
The offset built-in function (%OFFSET or %OFS) can be used to store or change the offset portion of a CL pointer variable.

%SCAN built-in function
The scan built-in function (%SCAN) returns the first position of a *search argument* in the *source string*, or 0 if it was not found.

%SIZE built-in function
The %SIZE built-in function returns the number of bytes occupied by the CL variable.

%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.

%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'.

%TRIM built-in function
The trim built-in function (%TRIM) with one parameter produces a character string with any leading and trailing blanks removed. The trim built-in function (%TRIM) with two parameters produces a character string with any leading and trailing characters that are in the *characters to trim* parameter removed.

%TRIML built-in function
The trim left built-in function (%TRIML) with one parameter produces a character string with any leading blanks removed. The trim left built-in function (%TRIML) with two parameters produces a character string with any leading characters that are in the *characters to trim* parameter removed.

%TRIMR built-in function
The trim right built-in function (%TRIMR) with one parameter produces a character string with any trailing blanks removed. The trim right built-in function (%TRIMR) with two parameters produces a character string with any trailing characters that are in the *characters to trim* parameter removed.

%UINT built-in function

%UINT converts character, logical, decimal, or integer data to unsigned integer format. The converted value can be assigned to a CL variable, passed as a numeric constant to another program or procedure, or specified as a value for a command parameter of a CL command run from compiled CL.

%UPPER built-in function
The %UPPER built-in function returns a character string that is the same length as the argument specified with each lowercase letter replaced by the corresponding uppercase letter.

Related information
Change Variable (CHGVAR) command

Trailing blanks on command parameters

In certain command parameters, you can define how trailing blanks are processed.

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 single quotation marks. 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 that 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 **Override with Database File (OVRDBF)** command.

When this condition occurs, the result that you want can be attained for these parameters by constructing a command string that delimits the parameter value with single quotation marks and passing the string to QCMDEXC or QCAPCMD for processing.

Here is an example of a program that can be used to run the **Override with Database File (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.

Note: By using the code example, you agree to the terms of the “Code license and disclaimer information” on page 610.

```
            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.

Related information

Override with Data Base File (OVRDBF) command

Writing comments in CL programs or procedures

To write comments or add comments to commands in your CL programs or 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 in different 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 embedded within a comment.

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

Note: By using the code example, you agree to the terms of the "Code license and disclaimer information" on page 610.

```
       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 program or CL procedure

You can use commands to change the flow of logic within your 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
- CALLSUBR

Conditional branching means that under certain specified conditions, processing can branch to sections or commands that are not consecutive within the procedure. The branching can 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 typically 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

Related concepts

Variables in CL commands
A *variable* is a named changeable value that can be accessed or changed by referring to its name.

Related information

CL command finder

GOTO command and command labels in a CL program or procedure

The **Go To (GOTO)** command processes an unconditional branch.

With the **GOTO** command, processing is directed to another part (identified by a label) of the program or 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 the **GOTO** command to go to a label outside the program or procedure, nor can you use a **GOTO** command to branch into or out of a subroutine defined within the program or 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 program or 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.

The label in the following 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.

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

Related information

CL command finder

Go To (GOTO) command

IF command in a CL program or procedure

The **If (IF)** command is used to state a condition that, if true, specifies a statement or group of statements in the program or procedure to be run.

The **Else (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 can 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.

If the condition described by the logical expression is evaluated as true, the program or procedure processes the CL command on the THEN parameter. This can be a single command or a group of commands. If the condition is not true, the program or 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 example is about 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 program or 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 program or procedure then processes the next command, ENDPGM, which signals the end of the program or procedure, and returns control to the calling program or 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 program or 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. An embedded command is a command that is completely enclosed in the parameter of another command.

In the following examples, the **Change Variable (CHGVAR)** command and the DO command are embedded:

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

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

Related reference

*AND, *OR, and *NOT operators
The logical operators *AND and *OR specify the relationship between operands in a logical expression. The logical operator *NOT is used to negate logical variables or constants.

DO command and DO groups in a CL program or procedure
The **Do (DO)** command allows you to process a group of commands together.

ELSE command in a CL program or procedure
The **Else (ELSE)** command is a way of specifying alternative processing if the condition on the associated **If (IF)** command is false.

Embedded IF commands in a CL program or procedure
An **If (IF)** command can be embedded in another **IF** command.

Related information

CL command finder

If (IF) command

ELSE command in a CL program or procedure

The **Else (ELSE)** command is a way of specifying alternative processing if the condition on the associated **If (IF)** command is false.

The IF command can be used without the **ELSE** command:

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

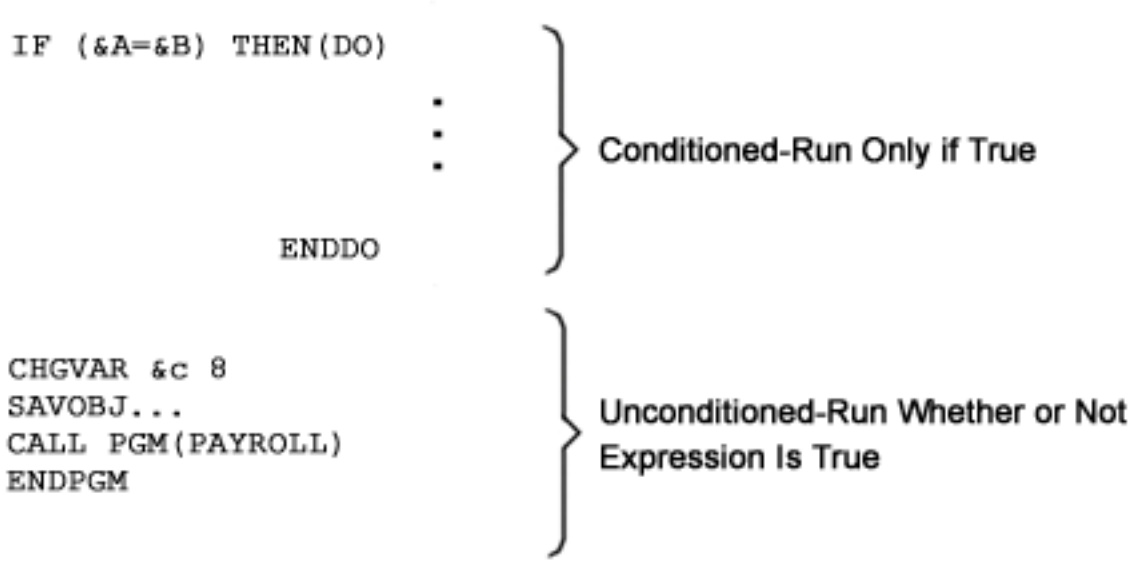
In this case, PROCA is called only if &A=&B, but PROCB is always called.

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.

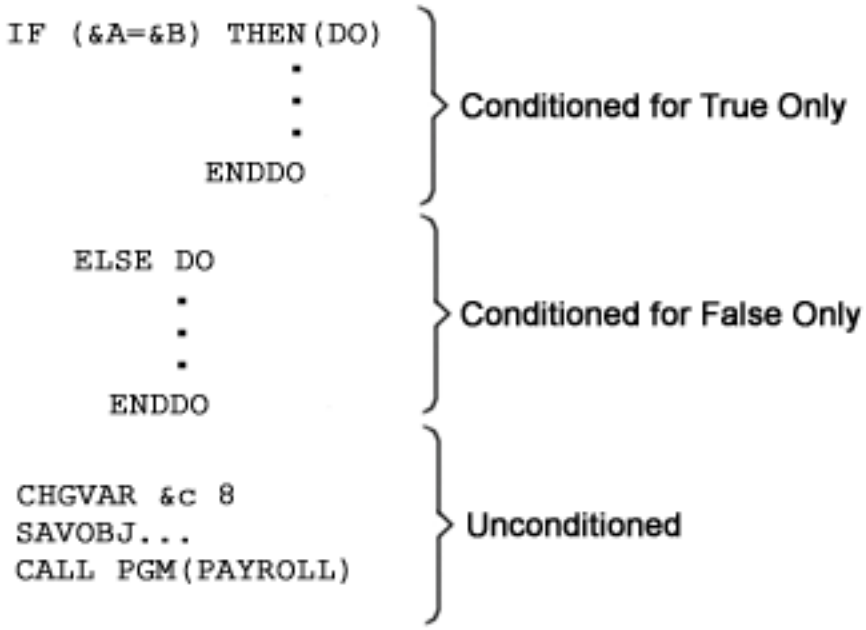
```
IF (&A=&B) THEN(CALLPRC PROCA)
ELSE CMD(CALLPRC PROCB)
CHGVAR &C 8
```

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 might 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 **Reclaim Resources (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
```

Related reference

IF command in a CL program or procedure

The **If (IF)** command is used to state a condition that, if true, specifies a statement or group of statements in the program or procedure to be run.

*AND, *OR, and *NOT operators
The logical operators *AND and *OR specify the relationship between operands in a logical expression. The logical operator *NOT is used to negate logical variables or constants.

Related information
CL command finder
Else (ELSE) command

Embedded IF commands in a CL program or procedure

An **If (IF)** command can be embedded in another **IF** command.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 610.

Embedding an IF command can 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 might want 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 (**Change Variable (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.

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 **Send Program Message (SNDPGMMSG)** command is processed, followed by the **Change Variable (CHGVAR)** command. If the first and second conditions (&RESP=1 and &A=5) are true, but the third (&B=N0) is false, PROCA is called; when PROCA returns, the CHGVAR command is processed. If the second conditions fails, PROCB is called (&B=N0 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)))
```

Related reference

IF command in a CL program or procedure
The **If (IF)** command is used to state a condition that, if true, specifies a statement or group of statements in the program or procedure to be run.

*AND, *OR, and *NOT operators
The logical operators *AND and *OR specify the relationship between operands in a logical expression. The logical operator *NOT is used to negate logical variables or constants.

Related information

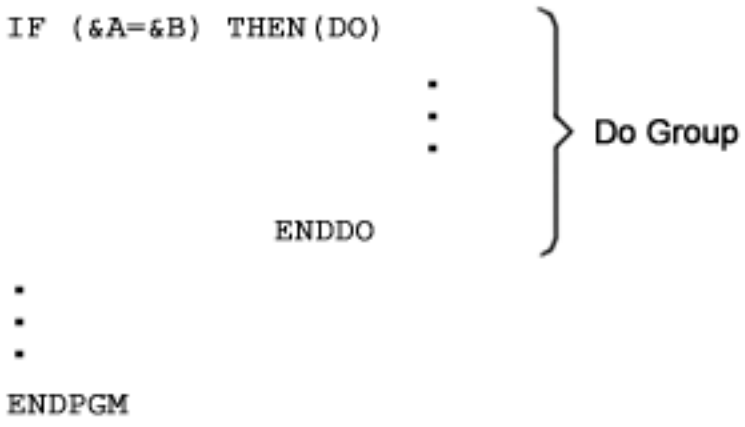
CL command finder
If (IF) command

DO command and DO groups in a CL program or procedure

The **Do (DO)** command allows you to process a group of commands together.

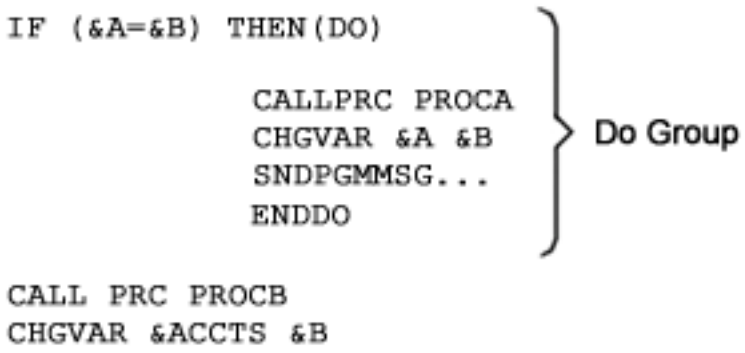
The group is defined as all those commands between the **DO** command and the corresponding **End Do Group (ENDDO)** command.

Processing of the group is typically conditioned on the evaluation of an associated command. Do groups are most frequently associated with the **IF**, **ELSE**, or **MONMSG** commands. Here is an example of a Do group.



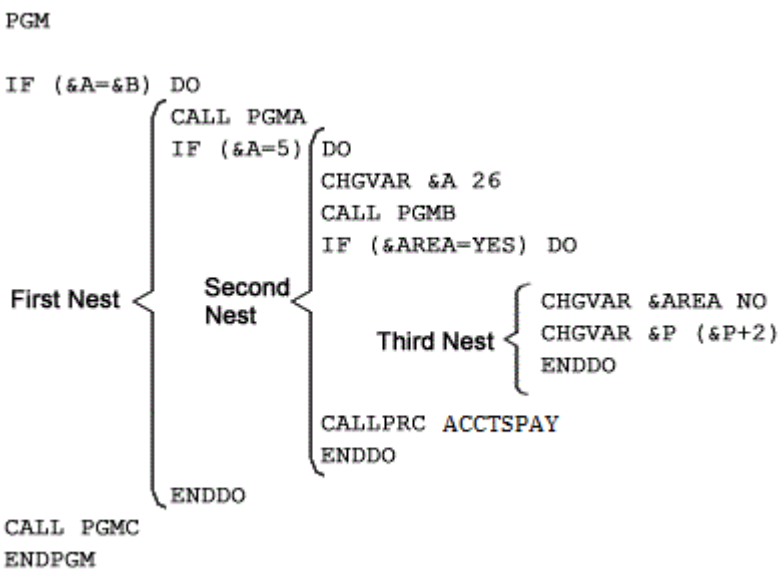
If the logical expression (&A=&B) is true, then the Do group is processed. If the expression is not true, then processing starts after the **ENDDO** command; the Do group is skipped.

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.



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.



In this example, if &A in the first nest does not equal 5, PGMC is called. If &A equals 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.

Related reference

IF command in a CL program or procedure
The **If (IF)** command is used to state a condition that, if true, specifies a statement or group of statements in the program or procedure to be run.

Related information

CL command finder
Do Group (DO) command

Showing DO and SELECT nesting levels

You can specify a CL compiler option to see the nesting level for all types of **DO** commands as well as **SELECT** commands.

Some CL source programs contain **DO** commands or **SELECT** commands where these commands are nested several levels deep. For example, between a **DO** command and the corresponding **ENDDO** command can be a **DOFOR** and another **ENDDO** command. The CL compiler supports up to 25 levels of nesting for **DO** commands and **SELECT** commands.

Changing CL source code with many levels of **DO** command or **SELECT** command nesting can be difficult. You must correctly match each type of **DO** command with a matching **ENDDO** command. You also must match each **SELECT** command with a matching **ENDSELECT** command.

The CL compiler provides an option to show the nesting levels for **Do (DO)**, **Do For (DOFOR)**, **Do Until (DOUNTIL)**, **Do While (DOWHILE)**, and **Select (SELECT)** commands on the CL compiler listing. Specify *DOSLTLVL for the OPTION parameter on the **Create CL Program (CRTCLPGM)** command, **Create CL Module (CRTCLMOD)** command, or **Create Bound CL Program (CRTBNDCL)** command to have the nesting level appear in the compiler listing. If you do not want to see this nesting level information, you can specify *NODOSLTLVL for the OPTION parameter.

Related information

Create CL Program (CRTCLPGM) command
Create Bound CL Program (CRTBNDCL) command

Create CL Module (CRTCLMOD) command

DOUNTIL command in a CL program or procedure

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 **End Do (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 can 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.

The following example is about 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 (LEAVE)** command can be used to exit the DOUNTIL group and resume processing following the ENDDO. The ITERATE command can be used to skip the remaining commands in the group and evaluate the stated condition immediately.

Related reference

*AND, *OR, and *NOT operators
The logical operators *AND and *OR specify the relationship between operands in a logical expression. The logical operator *NOT is used to negate logical variables or constants.

Related information

CL command finder
Do Until (DOUNTIL) command

DOWHILE command in a CL program or procedure

The **Do While (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 program or procedure to run. The group of commands is defined as those commands between the **DOWHILE** and the matching **End Do (ENDDO)** command.

The stated condition is evaluated before the group of commands is processed. The group of commands will never be processed if the stated condition is initially false. 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 can 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.

The following example is about 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 (LEAVE)** command can be used to exit the DOWHILE group and resume processing following the ENDDO. The ITERATE command can be used to skip the remaining commands in the group and evaluate the stated condition immediately.

Related reference

*AND, *OR, and *NOT operators
The logical operators *AND and *OR specify the relationship between operands in a logical expression. The logical operator *NOT is used to negate logical variables or constants.

Related information

CL command finder
Do While (DOWHILE) command

DOFOR command in a CL program or procedure

The **Do For (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 example is about 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. Because &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 can be used to exit the DOFOR group and resume processing following the ENDDO. The ITERATE command can be used to skip the remaining commands in the group, increment the controlling variable, and evaluate the end-value condition immediately.

Related information

CL command finder

Do For (DOFOR) command

ITERATE command in a CL program or procedure

The **Iterate (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 example 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 **Do For (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 **Do For (DOFOR)** command following the DOWHILE.

Related information

CL command finder

Iterate (ITERATE) command

LEAVE command in a CL program or procedure

The **Leave (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 (GOTO)** command. LEAVE is not valid with simple **Do (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 (IF)** command is processed and &A is less than zero, processing continues with the next command following the first ENDDO.

Related information

CL command finder

Leave (LEAVE) command

CALLSUBR command in a CL program or procedure

The **Call Subroutine (CALLSUBR)** command is used in a CL program or procedure for passing control to a subroutine that is defined within the same program or procedure.

The **CALLSUBR** command has two parameters: **Subroutine (SUBR)**, which contains the name of the subroutine to which control is to be transferred to, and **Return value (RTNVAL)**, which specifies the variable that will contain the return value from the called subroutine. See the following example:

```
CALLSUBR SUBR(mysubr) RTNVAL(&myrtnvar)
```

The subroutine `mysubr` must be defined in the program or procedure by the Subroutine (SUBR) parameter of an **SUBR** command. The variable *&myrtnvar* must be defined as TYPE(*INT) LEN(4) so that it contains the value from the Return value (RTNVAL) parameter of either a **Return from Subroutine**

(RTNSUBR) or **End Subroutine (ENDSUBR)** command found in subroutine `mysubr`. If no RTNVAL parameter is defined, the return value from the subroutine is ignored.

The **CALLSUBR** command can be placed anywhere within the program or procedure, including other subroutines, with the exception of a program-level **Monitor Message (MONMSG)** command. Each **CALLSUBR** command, when run, places a return address onto the subroutine stack, and the size of the stack can be changed with the use of the Subroutine stack (SUBRSTACK) parameter of the **Declare Process Options (DCLPRCOPT)** command. If a globally monitored message causes a GOTO command to be run, the subroutine stack will be reset by the next **CALLSUBR** command that is run.

In the following example, the first **CALLSUBR** command passes control to the subroutine SUBR1, and the return value of 12 is placed into the variable `&myrtnvar` when control returns. If a message is monitored by the **MONMSG** command, the **Goto (GOTO)** command is run and control branches to the label DUMP. The CL program or procedure is dumped by the **DMPCLPGM** command , and the next **CALLSUBR** command to subroutine SUBR2 resets the subroutine stack.

```
            PGM
            DCL         VAR(&myrtnvar) TYPE(*INT) LEN(4)
            MONMSG      MSGID(CPF0000) EXEC(GOTO CMDLBL(DUMP))
            :
            CALLSUBR    SUBR(SUBR1) RTNVAL(&myrtnvar)
            :
DUMP:       DMPCLPGM
            CALLSUBR    SUBR(SUBR2)
            :
            SUBR        SUBR(SUBR1)
            :
            ENDSUBR     RTNVAL(12)
            :
            SUBR        SUBR(SUBR2)
            :
            ENDSUBR
            ENDPGM
```

Related reference

SUBR command and subroutines in a CL program or procedure
The **Subroutine (SUBR)** command is used in a CL program or procedure, along with the **End Subroutine (ENDSUBR)** command, to delimit the group of commands that define a subroutine.

Related information

Declare Processing Options (DCLPRCOPT) command
CL command finder
Call Subroutine (CALLSUBR) command

SELECT command and SELECT groups in a CL program or procedure

The **Select (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 can also be specified to be processed when none of the stated conditions are true. Only one of the groups of commands identified by **When (WHEN)** or **Otherwise (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 can 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.

If the condition described by the logical expression is evaluated as true, the procedure processes the CL command on the THEN parameter. This can 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.

Related reference

*AND, *OR, and *NOT operators
The logical operators *AND and *OR specify the relationship between operands in a logical expression. The logical operator *NOT is used to negate logical variables or constants.

Related information

CL command finder
Select (SELECT) command

SUBR command and subroutines in a CL program or procedure

The **Subroutine (SUBR)** command is used in a CL program or procedure, along with the **End Subroutine (ENDSUBR)** command, to delimit the group of commands that define a subroutine.

The name of the subroutine, as used by the CALLSUBR command, is defined by the SUBR parameter on the **SUBR** command.

The first **SUBR** command that is encountered in a program or procedure also marks the end of the mainline of that program or procedure. All commands from this point forward, with the exception of the ENDPGM command, must be contained within a subroutine, in other words, between **SUBR** and **ENDSUBR** commands. The **SUBR** and **ENDSUBR** commands must be matched pairs, and subroutines cannot be nested. In other words, no SUBR SUBR ENDSUBR ENDSUBR nesting of subroutines is allowed.

Both the ENDSUBR and RTNSUBR commands can be used to exit a subroutine, and when processed, control is returned to the command immediately following the **Call Subroutine (CALLSUBR)** command that called the subroutine. Both commands have an optional RTNVAL parameter, which can be anything that can be stored in a CL variable of TYPE(*INT) and LEN(4). If no RTNVAL parameter is defined on the **Return Subroutine (RTNSUBR)** or **ENDSUBR** command, a value of zero is returned.

The following example is about the general structure of a CL procedure that contains a subroutine.

```
             PGM
             DCLPRCOPT  SUBRSTACK(25)
             DCL        VAR(&RTNVAR) TYPE(*INT) LEN(4)
                  :
             CALLSUBR   SUBR(SUBR1) RTNVAL(&RTNVAR)
                  :
             SUBR       SUBR(SUBR1)
                  :
             RTNSUBR    RTNVAL(-1)
                  :

ENDSUBR

             ENDPGM
```

In this example, the **Declare Processing Options (DCLPRCOPT)** command was used to specify the size of the subroutine stack to be 25. The variable &RTNVAR is used to contain the return value from the subroutine. The **CALLSUBR** command will transfer control to the subroutine SUBR1, as defined by the SUBR command. If the **RTNSUBR** command is run, the value of &RTNVAR will be -1, if the **ENDSUBR** command is run, &RTNVAR will equal 0. If no RTNVAL parameter was defined on the **CALLSUBR** command, the return value from the subroutine would be ignored.

Related reference

CALLSUBR command in a CL program or procedure
The **Call Subroutine (CALLSUBR)** command is used in a CL program or procedure for passing control to a subroutine that is defined within the same program or procedure.

Related information

CL command finder
Subroutine (SUBR) command
End Subroutine (ENDSUBR) command

*AND, *OR, and *NOT operators

The logical operators *AND and *OR specify the relationship between operands in a logical expression. The logical operator *NOT is used to negate logical variables or constants.

*AND and *OR are the reserved values 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, ¬<)

The following examples are about 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 following example the entire logical expression is enclosed in parentheses, and both operands are relational expressions, also enclosed separately in parentheses:

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

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 want 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 “Embedded IF commands in a CL program or procedure” on page 211 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.

Related reference
IF command in a CL program or procedure

The **If (IF)** command is used to state a condition that, if true, specifies a statement or group of statements in the program or procedure to be run.

ELSE command in a CL program or procedure
The **Else (ELSE)** command is a way of specifying alternative processing if the condition on the associated **If (IF)** command is false.

DOUNTIL command in a CL program or procedure
The **Do Until (DOUNTIL)** command processes a group of CL commands one or more times.

DOWHILE command in a CL program or procedure
The **Do While (DOWHILE)** command lets you process a group of commands zero or more times while the value of a logical expression is true.

SELECT command and SELECT groups in a CL program or procedure
The **Select (SELECT)** command is used to identify one or more conditions and an associated group of commands to process when that condition is true.

%ADDRESS built-in function

The address built-in function (%ADDRESS or %ADDR) can be used to change or test the memory address stored in a CL pointer variable.

In a CHGVAR command, you can specify the %ADDRESS function to change the value of a pointer variable. On the IF command, %ADDRESS function can be specified on the COND parameter to check the value stored in a pointer variable.

The format of the address built-in function is:

```
%ADDRESS(variable name)
```

or

```
%ADDR(variable name)
```

In the following example, pointer variable &P1 is initialized to the address of the first byte of character variable &C1. Later in the procedure, the pointer is checked using the %ADDRESS function to see if it still points to &C1 and, if not, resets &P1 to the first byte of CL variable &C1 using the %ADDRESS function.

```
PGM
DCL &C1  *CHAR  10
DCL &P1   *PTR  ADDRESS(&C1)
:
IF COND(&P1 *NE %ADDRESS(&C1))  +
   THEN(CHGVAR  &P1  %ADDRESS(&C1))
:
ENDPGM
```

Note: The %ADDRESS function cannot be used to store the address offset stored in a pointer variable. To store the address offset in a pointer variable, you need to use the %OFFSET built-in function.

Related tasks

Changing the value of a variable
You can change the value of a CL variable using the **Change Variable (CHGVAR)** command.

Related reference

Built-in functions for CL

Control language (CL) provides several built-in functions. Built-in functions are used in arithmetic, character string, relational, or logical expressions. All of these built-in functions can only be used in compiled CL programs or procedures.

%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 shown in the following example:

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

You can also code it like this example:

```
%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 can also be used for the starting position.)

You can use the binary built-in function with both the **If (IF)** and **Change Variable (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 **Change Variable (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 **Call Bound Procedure (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 examples are about 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 **Send Program Message (SNDPGMMSG)** 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.

Related tasks

Changing the value of a variable

You can change the value of a CL variable using the **Change Variable (CHGVAR)** command.

Related reference

Built-in functions for CL
  ```

Control language (CL) provides several built-in functions. Built-in functions are used in arithmetic, character string, relational, or logical expressions. All of these built-in functions can only be used in compiled CL programs or procedures.

Related information

CL command finder
If (IF) command
Change Variable (CHGVAR) command
Call Bound Procedure (CALLPRC) command
Send Program Message (SNDPGMMSG) command

%CHAR built-in function

%CHAR converts logical, decimal, integer, or unsigned integer data to character format. The converted value can be assigned to a CL variable, passed as a character constant to another program or procedure, or specified as a value for a command parameter of a CL command run from compiled CL.

The %CHAR built-in function can be used anywhere that CL supports a character expression. %CHAR can be used alone or as part of a more complex character expression. For example, %CHAR can be used to compare a numeric CL variable to a character CL variable in the COND parameter of an **IF** or **WHEN** command. %CHAR can also be used to set the value of a CL command parameter, if the associated command object defines the parameter with EXPR(*YES) and TYPE of *CHAR, *NAME, *SNAME, *CNAME, *PNAME, *GENERIC, *DATE, *TIME, or *X.

The format of the convert to character data built-in function is:

```
%CHAR(convert-argument)
```

The *convert-argument* must be a CL variable with TYPE of *LGL, *DEC, *INT or *UINT.

For logical data, the result will be ether '0' or '1'.

For numeric data, the result will be in decimal format, left-adjusted with a leading negative sign if the value is negative, and without leading zeros. A decimal point is inserted if needed, and the character used for any decimal point will be the character indicated by the decimal format (QDECFMT) system value (default is '.'). For example, %CHAR of a CL variable declared with TYPE(*DEC) and LEN(7,3) CL variable might return the value '-1.234'. The decimal format (QDECFMT) system value also determines the type of zero suppression of the conversion result.

The following are examples of using the %CHAR built-in function:

- Convert packed decimal variable

```
DCL VAR(&MSG) TYPE(*CHAR) LEN(26)
DCL VAR(&ANSWER) TYPE(*DEC) LEN(10 5) VALUE(-54321.09876)

/* &MSG will have the value 'The answer is -54321.09876' */
CHGVAR VAR(&MSG) VALUE('The answer is' *BCAT %CHAR(&ANSWER))
```

- Convert integer or unsigned integer variable

```
DCL VAR(&MSG) TYPE(*CHAR) LEN(60)
DCL VAR(&LENGTH) TYPE(*UINT) LEN(2) VALUE(50)
DCL VAR(&TEMP) TYPE(*INT) LEN(2) VALUE(-10)

/* &MSG will have the value 'The length of this box is 50 centi+
meters.                    ' */
CHGVAR VAR(&MSG) VALUE('The length of this box is' *BCAT %CHAR(&LENGTH) +
*BCAT 'centimeters.')
/* &MSG will have the value 'The temperature dropped to -10 degrees +
Centigrade.          ' */
CHGVAR VAR(&MSG) VALUE('The temperature dropped to' *BCAT %CHAR(&TEMP) +
*BCAT 'degrees Centigrade.')
```

- Convert logical variable

```
DCL VAR(&ENDOFFILE) TYPE(*LGL) VALUE('1')
SNDPGMMSG MSG('End of file?' *BCAT %CHAR(&ENDOFFILE))
```

- Leading zeros will be suppressed but all decimal position zeros will be kept

```
DCL &AMOUNT TYPE(*DEC) LEN(7 4) VALUE(099.5)
/* &MSG will have the value "Your amount comes to 99.5000" */
SNDPGMMSG MSG('Your amount comes to' *BCAT %CHAR(&AMOUNT))
```

- Decimal format and zero suppression

```
DCL VAR(&MSG) TYPE(*CHAR) LEN(20)
DCL VAR(&ANSWER) TYPE(*DEC) LEN(10 2) VALUE(-0.45)

/* &MSG will have the value 'The answer is -.45  ', while +
CHGJOB DECFMT(*BLANK) */
/* &MSG will have the value 'The answer is -0,45 ', while CHGJOB DECFMT(J) */
/* &MSG will have the value 'The answer is -,45  ', while CHGJOB DECFMT(I) */
CHGVAR VAR(&MSG) VALUE('The answer is' *BCAT %CHAR(&ANSWER))
```

Related tasks

Changing the value of a variable
You can change the value of a CL variable using the **Change Variable (CHGVAR)** command.

Related reference

Built-in functions for CL
Control language (CL) provides several built-in functions. Built-in functions are used in arithmetic, character string, relational, or logical expressions. All of these built-in functions can only be used in compiled CL programs or procedures.

%CHECK built-in function

The check built-in function (%CHECK) returns the first position of a *base string* that contains a character that does not appear in the *comparator string*. If all of the characters in the base string also appear in the comparator string, the function returns 0.

The %CHECK built-in function can be used anywhere that CL supports an arithmetic expression. %CHECK can be used alone or as part of a more complex arithmetic expression. For example, %CHECK can be used to compare to a numeric CL variable in the COND parameter of an **IF** or **WHEN** command. %CHECK can also be used to set the value of a CL command parameter, if the associated command object defines the parameter with EXPR(*YES) and TYPE of *DEC, *INT2, *INT4, *UINT2, or *UINT4.

The format of the check built-in function is:

```
%CHECK(comparator-string base-string [starting-position])
```

The *comparator string* must be either a CL character variable or a character literal. The *base string* can be a CL character variable or *LDA. When *LDA is specified, the check function is performed on the contents of the local data area for the job. The *starting position* is optional and defaults to 1. Checking begins at the starting position (start) and continues to the right until a character that is not contained in the comparator string is found. The result is always relative to the first byte in the source string, even if the starting position is specified. The starting position, if specified, must be either a CL integer variable or a CL decimal variable with zero decimal positions or a numeric literal with zero decimal positions. The starting position cannot be zero or negative. If the starting position is greater than the length of the entire base variable or the local data area, an error occurs. The length of the local data area is 1024.

The following examples are about the check built-in function:

- Check for any characters that are not digits (0-9). Here the comparator string is a character literal that contains the character digits 0 through 9. If the CL variable &SN contains any characters that are not digits, a message is sent.

```
PGM PARM(&SN)
DCL VAR(&SN) TYPE(*CHAR) LEN(10)
```

```
IF COND(%CHECK('0123456789' &SN) *NE 0) +
   THEN(SNDPGMMSG ('INVALID CHARACTER FOUND!'))
```

- Find the first character that is not a dollar sign or an asterisk. The characters in variable &PRICE are checked from left to right. Since the first character that is not a dollar sign or asterisk is the eighth character, the value 8 is assigned to CL variable &POS by the **CHGVAR** command.

```
DCL VAR(&PRICE) TYPE(*CHAR) VALUE('$******5.27***      ')
DCL VAR(&COMP) TYPE(*CHAR) LEN(2) VALUE('$*')
DCL VAR(&POS) TYPE(*UINT) LEN(2)
CHGVAR VAR(&POS) VALUE(%CHECK(&COMP &PRICE))
```

- Check if any characters are not digits, starting from a specific position in the string. The characters in CL variable &SN are checked starting with the fifth byte. If there are characters in bytes 5 through 10 of &SN that are not digits, a message is sent.

```
PGM PARM(&SN)
DCL VAR(&SN) TYPE(*CHAR) LEN(10)
DCL VAR(&POS) TYPE(*UINT) LEN(2) VALUE(5)
IF COND(%CHECK('0123456789' &SN &POS) *NE 0) +
   THEN(SNDPGMMSG ('INVALID CHARACTER FOUND!'))
```

- Check characters from the local data area (*LDA). The position of the leftmost byte in the local data area (LDA) that is not a character digit is assigned to CL variable &POS. If all 1024 characters of the LDA are character digits. a value of zero is assigned to variable &POS.

```
DCL VAR(&POS) TYPE(*UINT) LEN(2)
CHGVAR VAR(&POS) VALUE(%CHECK('0123456789' *LDA))
```

Related tasks

Changing the value of a variable

You can change the value of a CL variable using the **Change Variable (CHGVAR)** command.

Related reference

Built-in functions for CL

Control language (CL) provides several built-in functions. Built-in functions are used in arithmetic, character string, relational, or logical expressions. All of these built-in functions can only be used in compiled CL programs or procedures.

%CHECKR built-in function

The reverse check built-in function (%CHECKR) returns the last position of a *base string* that contains a character that does not appear in the *comparator string*. If all of the characters in the base string also appear in the comparator string, the function returns 0.

The %CHECKR built-in function can be used anywhere that CL supports an arithmetic expression. %CHECKR can be used alone or as part of a more complex arithmetic expression. For example, %CHECKR can be used to compare to a numeric CL variable in the COND parameter of an **IF** or **WHEN** command. %CHECKR can also be used to set the value of a CL command parameter, if the associated command object defines the parameter with EXPR(*YES) and TYPE of *DEC, *INT2, *INT4, *UINT2, or *UINT4.

The format of the reverse check built-in function is:

```
%CHECKR(comparator-string base-string [starting-position])
```

The *comparator string* must be either a CL character variable or a character literal. The *base string* can be a CL character variable or *LDA. When *LDA is specified, the reverse check function is performed on the contents of the local data area for the job. The *starting position* is optional and defaults to the last byte of the base string. Checking begins at the starting position (start) and continues to the left until a character that is not contained in the comparator string is found. The result is always relative to the first byte in the source string, even if the starting position is specified. The starting position, if specified, must be either a CL integer variable or a CL decimal variable with zero decimal positions or a numeric literal with zero decimal positions. The starting position cannot be zero or negative. If the starting position is greater than

the length of the entire base variable or the local data area, an error occurs. The length of the local data area is 1024.

The following examples are about the reverse check built-in function:

- Retrieve a numeric value from a string. The first **CHGVAR** uses the %CHECK built-in function to find the leftmost character that is not a dollar sign ($) or an asterisk (*) or a blank. The second **CHGVAR** uses the %CHECKR built-in function to find the rightmost character that is not a dollar sign, asterisk, or blank. The third **CHGVAR** computes the number of characters between the two values and the last **CHGVAR** uses the substring (%SST) built-in function to convert part of the base string to a decimal CL variable.

```
DCL VAR(&PRICE) TYPE(*CHAR) VALUE('$******5.27***      ')
DCL VAR(&COMP) TYPE(*CHAR) LEN(3) VALUE('$* ')
DCL VAR(&SPOS) TYPE(*UINT) LEN(2)
DCL VAR(&EPOS) TYPE(*UINT) LEN(2)
DCL VAR(&DEC) TYPE(*DEC) LEN(3 2)
DCL VAR(&LEN) TYPE(*UINT) LEN(2)
CHGVAR VAR(&SPOS) VALUE(%CHECK(&COMP &PRICE))
CHGVAR VAR(&EPOS) VALUE(%CHECKR(&COMP &PRICE))
CHGVAR VAR(&LEN) VALUE(&EPOS - &SPOS + 1)
CHGVAR VAR(&DEC) VALUE(%SST(&PRICE &SPOS &LEN))
```

- Reverse check if any characters are not digits, starting from a specific position in the string. The characters in CL variable &SN are checked, from right to left, starting with the ninth byte. If there are characters in bytes 9 through 1 of &SN that are not digits, a message is sent.

```
PGM PARM(&SN)
DCL VAR(&SN) TYPE(*CHAR) LEN(10)
DCL VAR(&POS) TYPE(*UINT) LEN(2) VALUE(9)
IF COND(%CHECKR('0123456789' &SN &POS) *NE 0) +
   THEN(SNDPGMMSG ('INVALID CHARACTER FOUND!'))
```

- Reverse check characters from the local data area (*LDA). The position of the rightmost byte in the local data area (LDA) that is not a character digit is assigned to CL variable &POS. If all 1024 characters of the LDA are character digits. a value of zero is assigned to variable &POS.

```
DCL VAR(&POS) TYPE(*UINT) LEN(2)
CHGVAR VAR(&POS) VALUE(%CHECKR('0123456789' *LDA))
```

Related tasks

Changing the value of a variable

You can change the value of a CL variable using the **Change Variable (CHGVAR)** command.

Related reference

Built-in functions for CL

Control language (CL) provides several built-in functions. Built-in functions are used in arithmetic, character string, relational, or logical expressions. All of these built-in functions can only be used in compiled CL programs or procedures.

%DEC built-in function

%DEC converts character, logical, decimal, integer, or unsigned integer data to packed decimal format. The converted value can be assigned to a CL variable, passed as a numeric constant to another program or procedure, or specified as a value for a command parameter of a CL command run from compiled CL.

The %DEC built-in function can be used anywhere that CL supports an arithmetic expression. %DEC can be used alone or as part of a more complex arithmetic expression. For example, %DEC can be used to compare a character CL variable to a numeric CL variable in the COND parameter of an **IF** or **WHEN** command. %DEC can also be used to set the value of a CL command parameter, if the associated command object defines the parameter with EXPR(*YES) and TYPE of *DEC, *INT2, *INT4, *UINT2, or *UINT4.

The format of the convert to packed decimal data built-in function is:

```
%DEC(convert-argument [total-digits decimal-places])
```

The *convert-argument* must be a CL variable with TYPE of *CHAR, *LGL, *DEC, *INT or *UINT.

Parameters *total-digits* and *decimal-places* may be omitted. If these parameters are omitted, for character data, the value of total digits defaults to 15, the value of decimal places defaults to 5; for logical data, the value of total digits defaults to 1, the value of decimal places defaults to 0; for numeric data, the total digits and decimal places are taken from the attributes of the numeric data. The total digits and decimal places, if specified, must be integer literals. The *total-digits* can be a value from 1 to 15. The *decimal-places* can be a value from zero to 9 and cannot be greater than the *total-digits* value.

If the *convert-argument* parameter is a character variable, the following rules apply:

- The sign is optional. It can be '+' or '-'. It can precede or follow the numeric data.
- The decimal point is optional. It can be either a period or a comma.
- Leading and trailing blanks are allowed in the data. For example, ' +3 ' is a valid parameter.
- All-blank value will return a zero value.
- If invalid numeric data is found, an exception occurs with CPF0818.

The following are examples of using the %DEC built-in function:

- Convert character variable

```
DCL VAR(&POINTS) TYPE(*CHAR) LEN(10) VALUE('-123.45')
DCL VAR(&ANSWER) TYPE(*DEC) LEN(10 5)

/* &ANSWER will have the value -00023.45000 */
CHGVAR VAR(&ANSWER) VALUE(100 + %DEC(&POINTS 5 2))
```

- Convert logical variable

```
DCL VAR(&ANSWER1) TYPE(*LGL) VALUE('1')
DCL VAR(&ANSWER2) TYPE(*LGL) VALUE('1')
DCL VAR(&NUM) TYPE(*DEC) LEN(5 0)

/* &NUM will have the value 00002. */
CHGVAR VAR(&NUM) VALUE(%DEC(&ANSWER1 1 0) + %DEC(&ANSWER2))
SNDPGMMSG MSG('The number of YES answers is' *BCAT %CHAR(&NUM))
```

- Convert packed decimal variable

```
DCL VAR(&POINTS1) TYPE(*DEC) LEN(5 2) VALUE(100.23)
DCL VAR(&POINTS2) TYPE(*DEC) LEN(5 2) VALUE(100.45)

IF (%DEC(&POINTS1 3 0) *EQ %DEC(&POINTS2 3 0)) +
THEN(SNDPGMMSG ('The scores are the same!'))
```

- Convert integer or unsigned integer variable

```
DCL VAR(&P1) TYPE(*INT) LEN(2) VALUE(-1)
DCL VAR(&P2) TYPE(*UINT) LEN(2) VALUE(1)
DCL VAR(&NUM) TYPE(*DEC) LEN(5 0)

/* &NUM will have the value 00000. */
CHGVAR VAR(&NUM) VALUE(%DEC(&P1 10 0)+%DEC(&P2 5 2))
```

- Extra decimal digits will be truncated without rounding

```
DCL VAR(&STRING) TYPE(*CHAR) LEN(10) VALUE('-123.567')
DCL VAR(&VALUE) TYPE(*DEC) LEN(7 2)

/* &VALUE will have the value -00123.56 */
CHGVAR VAR(&VALUE) VALUE(%DEC(&STRING 7 2))
```

- All-blank value will return a zero value

```
DCL VAR(&STRING) TYPE(*CHAR) LEN(10) VALUE('          ')
DCL VAR(&VALUE) TYPE(*DEC) LEN(7 2)

/* &VALUE will have the value 00000.00 */
CHGVAR VAR(&VALUE) VALUE(%DEC(&STRING 7 2))
```

Related tasks

Changing the value of a variable
You can change the value of a CL variable using the **Change Variable (CHGVAR)** command.

Related reference

Built-in functions for CL
Control language (CL) provides several built-in functions. Built-in functions are used in arithmetic, character string, relational, or logical expressions. All of these built-in functions can only be used in compiled CL programs or procedures.

%INT built-in function

%INT converts character, logical, decimal, or unsigned integer data to integer format. The converted value can be assigned to a CL variable, passed as a numeric constant to another program or procedure, or specified as a value for a command parameter of a CL command run from compiled CL.

The %INT built-in function can be used anywhere that CL supports an arithmetic expression. %INT can be used alone or as part of a more complex arithmetic expression. For example, %INT can be used to compare a character CL variable to a numeric CL variable in the COND parameter of an **IF** or **WHEN** command. %INT can also be used to set the value of a CL command parameter, if the associated command object defines the parameter with EXPR(*YES) and TYPE of *DEC, *INT2, *INT4, *UINT2, or *UINT4.

The format of the convert to integer data built-in function is:

```
%INT(convert-argument)
```

The *convert-argument* must be a CL variable with TYPE of *CHAR, *LGL, *DEC or *UINT.

If the *convert-argument* parameter is a character variable, the following rules apply:

- The sign is optional. It can be '+' or '-'. It can precede or follow the numeric data.
- The decimal point is optional. It can be either a period or a comma.
- Leading and trailing blanks are allowed in the data. For example, ' +3 ' is a valid parameter.
- All-blank value will return a zero value.
- If invalid numeric data is found, an exception occurs with CPF0818.

The result will be in integer format, any decimal digits are truncated without rounding. And the size of the result is always 4 bytes.

The following are examples of using the %INT built-in function:

- Convert character variable

```
DCL VAR(&POINTS) TYPE(*CHAR) LEN(10) VALUE('-123.45')
DCL VAR(&ANSWER) TYPE(*INT)

/* &ANSWER will have the value -23 */
CHGVAR VAR(&ANSWER) VALUE(100 + %INT(&POINTS))
```

- Convert logical variable

```
DCL VAR(&ANSWER1) TYPE(*LGL) VALUE('1')
DCL VAR(&ANSWER2) TYPE(*LGL) VALUE('1')
DCL VAR(&NUM) TYPE(*INT)

/* &NUM will have the value 2. */
CHGVAR VAR(&NUM) VALUE(%INT(&ANSWER1) + %INT(&ANSWER2))
SNDPGMMSG MSG('The number of YES is' *BCAT %CHAR(&NUM))
```

- Convert packed decimal variable

```
DCL VAR(&POINTS1) TYPE(*DEC) LEN(5 2) VALUE(100.23)
DCL VAR(&POINTS2) TYPE(*DEC) LEN(5 2) VALUE(100.45)
```

```
IF (%INT(&POINTS1) *EQ %INT(&POINTS2)) +
THEN(SNDPGMMSG ('The scores are the same!'))
```

- Convert unsigned integer variable

```
DCL VAR(&P1) TYPE(*INT) LEN(2)
DCL VAR(&P2) TYPE(*UINT) LEN(2) VALUE(1)

CHGVAR VAR(&P1) VALUE(%INT(&P2))
```

- Decimal digits will be truncated without rounding

```
DCL VAR(&STRING) TYPE(*CHAR) LEN(10) VALUE('-123.9')
DCL VAR(&ANSWER) TYPE(*INT)

/* &ANSWER will have the value -123 */
CHGVAR VAR(&ANSWER) VALUE(%INT(&STRING))
```

Related tasks

Changing the value of a variable
You can change the value of a CL variable using the **Change Variable (CHGVAR)** command.

Related reference

Built-in functions for CL
Control language (CL) provides several built-in functions. Built-in functions are used in arithmetic, character string, relational, or logical expressions. All of these built-in functions can only be used in compiled CL programs or procedures.

%LEN built-in function

The %LEN built-in function returns the number of digits or characters of the CL numeric or character variable.

The %LEN built-in function can be used anywhere that CL supports an arithmetic expression. %LEN can be used alone or as part of a more complex arithmetic expression. For example, %LEN can be used to compare the number of digits of a decimal CL variable to a numeric CL variable in the COND parameter of an **IF** or **WHEN** command. %LEN can also be used to set the value of a CL command parameter, if the associated command object defines the parameter with EXPR(*YES) and TYPE of *DEC, *INT2, *INT4, *UINT2, or *UINT4.

The format of the getting length built-in function is:

```
%LEN(variable-argument)
```

The *variable-argument* must be a CL variable with TYPE of *CHAR, *DEC, *INT or *UINT.

For numeric variables, the value returned represents the precision of the variables and not necessarily the actual number of significant digits. For 2-byte *INT or *UINT CL variables, the value returned is always 5. For 4-byte *INT or *UINT CL variables, the value returned is always 10. For 8-byte *INT CL variables, the value returned is always 19. For 8-byte *UINT CL variables, the value returned is always 20.

For character variables, the value returned is the number of the characters.

The following are examples of using the %LEN built-in function:

- Get the length of numeric variables

```
DCL VAR(&NUM1) TYPE(*DEC)
DCL VAR(&NUM2) TYPE(*DEC) LEN(7 2)
DCL VAR(&NUM3) TYPE(*INT) LEN(4)
DCL VAR(&NUM4) TYPE(*UINT) LEN(2)
DCL VAR(&RTN) TYPE(*INT) LEN(2)

/* &RTN will have the value 15. */
CHGVAR VAR(&RTN) VALUE(%LEN(&NUM1))
/* &RTN will have the value 7. */
CHGVAR VAR(&RTN) VALUE(%LEN(&NUM2))
/* &RTN will have the value 10. */
CHGVAR VAR(&RTN) VALUE(%LEN(&NUM3))
```

```
/* &RTN will have the value 5. */
CHGVAR VAR(&RTN) VALUE(%LEN(&NUM4))
```

- Get the length of character variables

```
DCL VAR(&CHAR1) TYPE(*CHAR)
DCL VAR(&CHAR2) TYPE(*CHAR) LEN(20)
DCL VAR(&RTN) TYPE(*INT) LEN(2)

/* &RTN will have the value 32. */
CHGVAR VAR(&RTN) VALUE(%LEN(&CHAR1))
/* &RTN will have the value 20. */
CHGVAR VAR(&RTN) VALUE(%LEN(&CHAR2))
/* &RTN will have the value 52. */
CHGVAR VAR(&RTN) VALUE(%LEN(&CHAR1) + %LEN(&CHAR2))
```

Related tasks

Changing the value of a variable

You can change the value of a CL variable using the **Change Variable (CHGVAR)** command.

Related reference

Built-in functions for CL

Control language (CL) provides several built-in functions. Built-in functions are used in arithmetic, character string, relational, or logical expressions. All of these built-in functions can only be used in compiled CL programs or procedures.

%LOWER built-in function

The %LOWER built-in function returns a character string that is the same length as the argument specified with each uppercase letter replaced by the corresponding lowercase letter.

The %LOWER built-in function can be used anywhere that CL supports a character expression. %LOWER can be used alone or as part of a more complex character expression. For example, %LOWER can be used to convert character CL variables to have only lower case characters before comparing them in the COND parameter of an **IF** or **WHEN** command. %LOWER can also be used to set the value of a CL command parameter, if the associated command object defines the parameter with EXPR(*YES) and TYPE of *CHAR, *NAME, *SNAME, *CNAME, *PNAME, *GENERIC, *DATE, *TIME, or *X.

The format of the convert to lowercase built-in function is:

```
%LOWER(input-string [CCSID])
```

The *input-string* must be a CL variable with TYPE of *CHAR.

The *CCSID* parameter is optional and defaults to the job CCSID. The *CCSID* specifies the coded character set identifier (CCSID) of the input string to be converted. Case conversion is performed based on this *CCSID*. The *CCSID*, if specified, must be a CL integer variable or a CL decimal variable with zero decimal positions or a numeric literal with zero decimal positions. The valid values are:

- 0: The CCSID of the job is used to determine the CCSID of the data to be converted. If the job CCSID is 65535, the CCSID from the default CCSID (DFTCCSID) job attribute is used.
- 1-65533: A valid CCSID in this range.

The following are examples of using the %LOWER built-in function:

- Convert to lowercase

```
DCL VAR(&STR) TYPE(*CHAR) LEN(12) VALUE('Hello World!')
/* 'hello world!' is to be sent */
SNDPGMMSG (%LOWER(&STR))
```

- Convert to lowercase based on CCSID

```
/* define &STR as 'Hello World!' in CCSID 819 (ISO/ANSI Multilingual) */
DCL VAR(&STR) TYPE(*CHAR) LEN(12) +
              VALUE(X'48656C6C6F20576F726C6421')
```

```
/* &STR will have the value x'68656C6C6F20776F726C6421' ('hello world!') */
CHGVAR VAR(&STR) VALUE(%LOWER(&STR 819))
```

Related tasks

Changing the value of a variable
You can change the value of a CL variable using the **Change Variable (CHGVAR)** command.

Related reference

Built-in functions for CL
Control language (CL) provides several built-in functions. Built-in functions are used in arithmetic, character string, relational, or logical expressions. All of these built-in functions can only be used in compiled CL programs or procedures.

%OFFSET built-in function

The offset built-in function (%OFFSET or %OFS) can be used to store or change the offset portion of a CL pointer variable.

In a CHGVAR command, the %OFFSET function can be used two different ways:

- You can specify the %OFFSET function for the variable (VAR parameter) to set the offset portion of a pointer variable.
- You can specify the %OFFSET function for the value (VALUE parameter) to which the variable is to be changed.

In an IF command, the %OFFSET function can be specified in the expression (COND parameter) and is treated like a four-byte unsigned integer value.

The format of the offset built-in function is:

```
%OFFSET(variable name)
```

or

```
%OFS(variable name)
```

In the following example, pointer variable &P1 has no initial value specified and therefore is initialized to null. The first CHGVAR command stores the offset from the null pointer value in integer variable &OFF1; the command runs without an error, and the value of &OFF1 is probably zero, though the offset for a null pointer is not defined. The second CHGVAR command sets &P1 to the address of local character variable &C1; pointer &P1 now points to byte 1 of variable &C1. The third CHGVAR command stores the offset portion of &P1 into the integer variable &OFF2. Note that the offset is from the start of the storage for all automatic variables for the current thread and not from the start of automatic storage for the current program or from the start of variable &C1. The fourth CHGVAR command takes the integer value stored in &OFF2, adds one hundred, and stores the resulting number into the integer variable &OFF3. The fifth CHGVAR command changes the offset portion of pointer &P1 using integer variable &OFF3; pointer &P1 now points to byte 101 of variable &C1. The sixth CHGVAR command calculates the integer expression for the VALUE parameter by storing the offset portion of pointer &P1 into a temporary integer variable and subtracting 20, and then uses this calculated value to reset the offset portion of pointer &P1; pointer &P1 now points to byte 81 of variable &C1.

Note: By using the code example, you agree to the terms of the “Code license and disclaimer information” on page 610.

```
PGM
DCL  &C1  *CHAR 30000
DCL  &P1  *PTR
DCL  &P2  *PTR
DCL  &OFF1  *UINT 4
DCL  &OFF2  *UINT 4
DCL  &OFF3  *UINT 4
CHGVAR &OFF1 %OFFSET(&P1)               /* 1 */
CHGVAR &P1 %ADDRESS(&C1)                /* 2 */
CHGVAR &OFF2 %OFFSET(&P1)               /* 3 */
CHGVAR &OFF3 (&OFF2+100)                /* 4 */
```

```
CHGVAR %OFFSET(&P1) &OFF3                /* 5 */
CHGVAR %OFFSET(&P1) (%OFFSET(&P1)-20)    /* 6 */
ENDPGM
```

The %OFFSET built-in function cannot be used with a pointer variable that addresses teraspace storage.

Related tasks

Changing the value of a variable

You can change the value of a CL variable using the **Change Variable (CHGVAR)** command.

Related reference

Built-in functions for CL

Control language (CL) provides several built-in functions. Built-in functions are used in arithmetic, character string, relational, or logical expressions. All of these built-in functions can only be used in compiled CL programs or procedures.

Related information

CL command finder

Change Variable (CHGVAR) command

%SCAN built-in function

The scan built-in function (%SCAN) returns the first position of a *search argument* in the *source string*, or 0 if it was not found.

The %SCAN built-in function can be used anywhere that CL supports an arithmetic expression. %SCAN can be used alone or as part of a more complex arithmetic expression. For example, %SCAN can be used to compare to a numeric CL variable in the COND parameter of an **IF** or **WHEN** command. %SCAN can also be used to set the value of a CL command parameter, if the associated command object defines the parameter with EXPR(*YES) and TYPE of *DEC, *INT2, *INT4, *UINT2, or *UINT4.

The format of the scan built-in function is:

```
%SCAN(search-argument source-string [starting-position])
```

The *search argument* must be either a CL character variable or a character literal. The *source string* can be a CL character variable or *LDA. When *LDA is specified, the scan function is performed on the contents of the local data area for the job. The *starting position* is optional and defaults to 1. Searching begins at the starting position (start) of the source string. The result is always relative to the first byte in the source string, even if the starting position is specified. The starting position, if specified, must be either a CL integer variable or a CL decimal variable with zero decimal positions or a numeric literal with zero decimal positions. The starting position cannot be zero or negative. If the starting position is greater than the length of the entire source-string variable or the local data area, an error occurs. The length of the local data area is 1024.

The following examples are about the scan built-in function:

- Search for a specific string, CL variable &FNAME is scanned for the string 'John'. If the string 'John' is not found anywhere in variable &FNAME, a message is sent.

  Note: The scan is case-sensitive, so a scan for 'John' does not return a positive result if &FNAME contains the value 'JOHN'.

  ```
  PGM PARM(&FNAME)
  DCL VAR(&FNAME) TYPE(*CHAR) LEN(10)
  IF COND(%SCAN('John' &FNAME) *EQ 0) +
     THEN(SNDPGMMSG ('NOT FOUND!'))
  ```

- Search from a specific position in the string. The %SCAN function is used multiple times to take a date value that contains a date separator and retrieve the month value into a numeric variable. The first %SCAN finds the position of the date separator between the day and month. The second %SCAN starts with the character that follows the first date separator and finds the position of the date separator between the month and year. The substring (%SST) built-in function is then used to convert the month value from character form to a decimal variable.

```
DCL VAR(&YYYYMMDD) TYPE(*CHAR) LEN(10) VALUE('2012/8/29')
DCL VAR(&DATSEP) TYPE(*CHAR) LEN(1) VALUE('/')
DCL VAR(&SPOS) TYPE(*UINT) LEN(2)
DCL VAR(&EPOS) TYPE(*UINT) LEN(2)
DCL VAR(&LEN) TYPE(*UINT) LEN(2)
DCL VAR(&MONTH) TYPE(*DEC) LEN(2)
CHGVAR VAR(&SPOS) VALUE(%SCAN(&DATSEP &YYYYMMDD))
CHGVAR VAR(&SPOS) VALUE(&SPOS + 1)
CHGVAR VAR(&EPOS) VALUE(%SCAN(&DATSEP &YYYYMMDD &SPOS))
CHGVAR VAR(&LEN) VALUE(&EPOS - &SPOS)
CHGVAR VAR(&MONTH) VALUE(%SST(&YYYYMMDD &SPOS &LEN))
```

- Search characters from the local data area (*LDA). The local data area (LDA) is scanned starting from byte 1 of the LDA for the string 'Escape'. If the string is found, the position of the string in the LDA is assigned to CL variable &POS. If the string 'Escape' is not found in the whole LDA, a value of zero is assigned to variable &POS.

```
DCL VAR(&POS) TYPE(*UINT) LEN(2)
CHGVAR VAR(&POS) VALUE(%SCAN('Escape' *LDA))
```

Related tasks

Changing the value of a variable

You can change the value of a CL variable using the **Change Variable (CHGVAR)** command.

Related reference

Built-in functions for CL

Control language (CL) provides several built-in functions. Built-in functions are used in arithmetic, character string, relational, or logical expressions. All of these built-in functions can only be used in compiled CL programs or procedures.

%SIZE built-in function

The %SIZE built-in function returns the number of bytes occupied by the CL variable.

The %SIZE built-in function can be used anywhere that CL supports an arithmetic expression. %SIZE can be used alone or as part of a more complex arithmetic expression. For example, %SIZE can be used to compare the number of bytes used by two CL variables in the COND parameter of an **IF** or **WHEN** command. %SIZE can also be used to set the value of a CL command parameter, if the associated command object defines the parameter with EXPR(*YES) and TYPE of *DEC, *INT2, *INT4, *UINT2, or *UINT4.

The format of the getting size built-in function is:

```
%SIZE(variable-argument)
```

The *variable-argument* must be a CL variable.

The following are examples of using the %SIZE built-in function:

- Get the size of numeric variables

```
DCL VAR(&NUM1) TYPE(*DEC)
DCL VAR(&NUM2) TYPE(*DEC) LEN(7 2)
DCL VAR(&NUM3) TYPE(*INT)
DCL VAR(&NUM4) TYPE(*UINT) LEN(8)
DCL VAR(&RTN) TYPE(*INT) LEN(2)

/* &RTN will have the value 8. */
CHGVAR VAR(&RTN) VALUE(%SIZE(&NUM1))
/* &RTN will have the value 4. */
CHGVAR VAR(&RTN) VALUE(%SIZE(&NUM2))
/* &RTN will have the value 4. */
CHGVAR VAR(&RTN) VALUE(%SIZE(&NUM3))
/* &RTN will have the value 8. */
CHGVAR VAR(&RTN) VALUE(%SIZE(&NUM4))
```

- Get the size of character variables

```
DCL VAR(&CHAR1) TYPE(*CHAR)
DCL VAR(&CHAR2) TYPE(*CHAR) LEN(20)
DCL VAR(&RTN) TYPE(*INT) LEN(2)

/* &RTN will have the value 32. */
CHGVAR VAR(&RTN) VALUE(%SIZE(&CHAR1))
/* &RTN will have the value 20. */
CHGVAR VAR(&RTN) VALUE(%SIZE(&CHAR2))
/* &RTN will have the value 52. */
CHGVAR VAR(&RTN) VALUE(%SIZE(&CHAR1) + %SIZE(&CHAR2))
```

- Get the size of logical variables

```
DCL VAR(&LGL1) TYPE(*LGL)
DCL VAR(&RTN) TYPE(*INT) LEN(2)

/* &RTN will have the value 1. */
CHGVAR VAR(&RTN) VALUE(%SIZE(&LGL1))
```

- Get the size of pointer variables

```
DCL VAR(&PTR1) TYPE(*PTR)
DCL VAR(&RTN) TYPE(*INT) LEN(2)

/* &RTN will have the value 16. */
CHGVAR VAR(&RTN) VALUE(%SIZE(&PTR1))
```

Related tasks

Changing the value of a variable

You can change the value of a CL variable using the **Change Variable (CHGVAR)** command.

Related reference

Built-in functions for CL

Control language (CL) provides several built-in functions. Built-in functions are used in arithmetic, character string, relational, or logical expressions. All of these built-in functions can only be used in compiled CL programs or procedures.

%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.

In a **Change Variable (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 (IF)** command, the %SST function can be specified in the expression.

The format of the substring built-in function is shown in this example:

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

It can also be formatted as shown in this example:

```
%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 examples are about 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.

Note: By using the code example, you agree to the terms of the "Code license and disclaimer information" on page 610.

```
        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.

Related tasks

Variables to use for specifying a list or qualified name
Variables can be used to specify a list or qualified name.

Changing the value of a variable
You can change the value of a CL variable using the **Change Variable (CHGVAR)** command.

Related reference

Built-in functions for CL
Control language (CL) provides several built-in functions. Built-in functions are used in arithmetic, character string, relational, or logical expressions. All of these built-in functions can only be used in compiled CL programs or procedures.

Related information

CL command finder
If (IF) command
Change Variable (CHGVAR) command

%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 **Submit Job (SBMJOB)**, **Change Job (CHGJOB)**, or JOB command; the default for these is the job description setting. Other high-level languages can 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 program or procedure to control the flow of the program or procedure. This function is used in CL programs or procedures with the **If (IF)** and **Change Variable (CHGVAR)** commands. Switches can be changed in a CL program or procedure by the **Change Job (CHGJOB)** command. For CL programs or procedures, these changes take effect immediately.

Related tasks

Changing the value of a variable
You can change the value of a CL variable using the **Change Variable (CHGVAR)** command.

Related reference

Built-in functions for CL
Control language (CL) provides several built-in functions. Built-in functions are used in arithmetic, character string, relational, or logical expressions. All of these built-in functions can only be used in compiled CL programs or procedures.

Related information

CL command finder
Change Job (CHGJOB) command
Submit Job (SBMJOB) command
Change Job Description (CHGJOBD) command
Change Variable (CHGVAR) command

%SWITCH with the IF command

On the **If (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 switch 1 contains 0, job switch 3 contains 1, job switch 4 contains 1, job switch 5 contains 1, and job switch 8 contains 0, 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.

Note: By using the code example, you agree to the terms of the "Code license and disclaimer information" on page 610.

```
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
```

Related information

If (IF) command

%SWITCH with the Change Variable command

On the **Change Variable (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, then the variable &A has a value of '1':

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

Related information

Change Variable (CHGVAR) command

%TRIM built-in function

The trim built-in function (%TRIM) with one parameter produces a character string with any leading and trailing blanks removed. The trim built-in function (%TRIM) with two parameters produces a character string with any leading and trailing characters that are in the *characters to trim* parameter removed.

The %TRIM built-in function can be used anywhere that CL supports a character expression. %TRIM can be used alone or as part of a more complex character expression. For example, %TRIM can be used

to compare to a character CL variable in the COND parameter of an **IF** or **WHEN** command. %TRIM can also be used to set the value of a CL command parameter, if the associated command object defines the parameter with EXPR(*YES) and TYPE of *CHAR, *NAME, *SNAME, *CNAME, *PNAME, *GENERIC, *DATE, *TIME, or *X.

The format of the trim built-in function is:

```
%TRIM(character-variable-name [characters-to-trim])
```

The trim function produces a substring from the contents of the specified CL character variable. If the characters-to-trim parameter is specified, it must be either a CL character variable or a character literal. If, after trimming, no characters are left, the trim function produces a string of blank characters.

The following examples are about the trim built-in function:

- Trim leading and trailing blank characters. The leading and trailing blanks are trimmed from the CL variables &FIRSTNAME and &LASTNAME and the resulting strings are concatenated with a single blank between the two values with the *BCAT operator. The concatenated string is then assigned to CL variable &NAME.

  ```
  DCL VAR(&FIRSTNAME) TYPE(*CHAR) VALUE('   JOHN  ')
  DCL VAR(&LASTNAME) TYPE(*CHAR) VALUE('   SMITH  ')
  DCL VAR(&NAME) TYPE(*CHAR) LEN(10)
  CHGVAR VAR(&NAME) VALUE(%TRIM(&FIRSTNAME) *BCAT %TRIM(&LASTNAME))
  ```

- Trim characters that are specified in a literal string. All asterisks and blanks are trimmed from the beginning and end of CL variable &NUM and the remaining characters (1.23) are assigned to CL variable &TRIMMED. The character variable &TRIMMED is converted to a numeric value and assigned to decimal variable &DEC.

  ```
  DCL VAR(&NUM) TYPE(*CHAR) LEN(10) VALUE('* *1.23* *')
  DCL VAR(&TRIMMED) TYPE(*CHAR) LEN(10)
  DCL VAR(&DEC) TYPE(*DEC) LEN(3 2)
  CHGVAR &TRIMMED %TRIM(&NUM '* ')
  CHGVAR VAR(&DEC) VALUE(&TRIMMED)
  ```

- Trim characters that are specified in a CL variable. Starting from the first character in variable &NAME, trim the character if it matches one of the characters in variable &TCHAR. Also, starting with the last character in variable &NAME, trim the character if it matches one of the characters in variable &TCHAR. The resulting substring of &NAME is compared to the literal string ’12345’ and a message is sent if the values are equal.

  ```
  DCL VAR(&NAME) TYPE(*CHAR) LEN(10) VALUE('*+12345   ')
  DCL VAR(&TCHAR) TYPE(*CHAR) LEN(3) VALUE('*+ ')
  IF COND(%TRIM(&NAME &TCHAR) *EQ '12345') +
     THEN(SNDPGMMSG ('EQUAL!'))
  ```

Related tasks

Changing the value of a variable
You can change the value of a CL variable using the **Change Variable (CHGVAR)** command.

Related reference

Built-in functions for CL
Control language (CL) provides several built-in functions. Built-in functions are used in arithmetic, character string, relational, or logical expressions. All of these built-in functions can only be used in compiled CL programs or procedures.

%TRIML built-in function

The trim left built-in function (%TRIML) with one parameter produces a character string with any leading blanks removed. The trim left built-in function (%TRIML) with two parameters produces a character string with any leading characters that are in the *characters to trim* parameter removed.

The %TRIML built-in function can be used anywhere that CL supports a character expression. %TRIML can be used alone or as part of a more complex character expression. For example, %TRIML can be used

to compare to a character CL variable in the COND parameter of an **IF** or **WHEN** command. %TRIML can also be used to set the value of a CL command parameter, if the associated command object defines the parameter with EXPR(*YES) and TYPE of *CHAR, *NAME, *SNAME, *CNAME, *PNAME, *GENERIC, *DATE, *TIME, or *X.

The format of the trim left built-in function is:

```
%TRIML(character-variable-name [characters-to-trim])
```

The trim left function produces a substring from the contents of the specified CL character variable. If the characters-to-trim parameter is specified, it must be either a CL character variable or a character literal. If, after trimming, no characters are left, the trim left function produces a string of blank characters.

The following examples are about the trim left built-in function:

- Trim leading blank characters. The leading blanks are trimmed from the CL variable &NUMCHAR and the resulting string is converted to a numeric value and assigned to CL variable &NUMDEC.

  ```
  DCL VAR(&NUMCHAR) TYPE(*CHAR) VALUE('     00001')
  DCL VAR(&NUMDEC) TYPE(*DEC) LEN(5)
  CHGVAR VAR(&NUMDEC) VALUE(%TRIML(&NUMCHAR))
  ```

- Trim leading characters that are specified in a literal string. All dollar signs and blanks are trimmed from the beginning of CL variable &PRICE and the remaining characters (5.27) are assigned to CL variable &TRIMMED. The character variable &TRIMMED is converted to a numeric value and assigned to decimal variable &DEC.

  ```
  DCL VAR(&PRICE) TYPE(*CHAR) VALUE('     $5.27')
  DCL VAR(&TRIMMED) TYPE(*CHAR) LEN(5)
  DCL VAR(&DEC) TYPE(*DEC) LEN(3 2)
  CHGVAR VAR(&TRIMMED) VALUE(%TRIML(&PRICE '$ '))
  CHGVAR VAR(&DEC) VALUE(&TRIMMED)
  ```

- Trim leading characters that are specified in a CL variable. Starting from the first character in variable &NAME, trim the character if it matches one of the characters in variable &TCHAR. The resulting substring of &NAME is compared to the literal string ’12345’ and a message is sent if the values are equal.

  ```
  DCL VAR(&NAME) TYPE(*CHAR) LEN(10) VALUE('    +12345')
  DCL VAR(&TCHAR) TYPE(*CHAR) LEN(2) VALUE('+ ')
  IF COND(%TRIML(&NAME &TCHAR) *EQ '12345') +
     THEN(SNDPGMMSG ('EQUAL!'))
  ```

Related tasks

Changing the value of a variable
You can change the value of a CL variable using the **Change Variable (CHGVAR)** command.

Related reference

Built-in functions for CL
Control language (CL) provides several built-in functions. Built-in functions are used in arithmetic, character string, relational, or logical expressions. All of these built-in functions can only be used in compiled CL programs or procedures.

%TRIMR built-in function

The trim right built-in function (%TRIMR) with one parameter produces a character string with any trailing blanks removed. The trim right built-in function (%TRIMR) with two parameters produces a character string with any trailing characters that are in the *characters to trim* parameter removed.

The %TRIMR built-in function can be used anywhere that CL supports a character expression. %TRIMR can be used alone or as part of a more complex character expression. For example, %TRIMR can be used to compare to a character CL variable in the COND parameter of an **IF** or **WHEN** command. %TRIMR can also be used to set the value of a CL command parameter, if the associated command object defines the parameter with EXPR(*YES) and TYPE of *CHAR, *NAME, *SNAME, *CNAME, *PNAME, *GENERIC, *DATE, *TIME, or *X.

The format of the trim right built-in function is:

```
%TRIMR(character-variable-name [characters-to-trim])
```

The trim right function produces a substring from the contents of the specified CL character variable. If the characters-to-trim parameter is specified, it must be either a CL character variable or a character literal. If, after trimming, no characters are left, the trim right function produces a string of blank characters.

The following examples are about the trim right built-in function:

- Trim trailing blank characters. The trailing blank characters are trimmed from the CL variables &FIRSTNAME and &LASTNAME and the resulting strings are concatenated with the *CAT operator. The concatenated string is then assigned to CL variable &NAME.

  ```
  DCL VAR(&FIRSTNAME) TYPE(*CHAR) LEN(10) VALUE('JOHN      ')
  DCL VAR(&LASTNAME) TYPE(*CHAR) LEN(10) VALUE('SMITH     ')
  DCL VAR(&NAME) TYPE(*CHAR) LEN(10)
  CHGVAR VAR(&NAME) VALUE(%TRIM(&FIRSTNAME) *CAT %TRIM(&LASTNAME))
  ```

- Trim trailing characters that are specified in a literal string. All trailing zero and plus sign characters are trimmed from CL variable &PRICE and the remaining characters (5.27) are assigned to CL variable &TRIMMED. The character variable &TRIMMED is converted to a numeric value and assigned to decimal variable &DEC.

  ```
  DCL VAR(&PRICE) TYPE(*CHAR) LEN(10) VALUE('5.2700000+')
  DCL VAR(&TRIMMED) TYPE(*CHAR) LEN(5)
  DCL VAR(&DEC) TYPE(*DEC) LEN(3 2)
  CHGVAR VAR(&TRIMMED) VALUE(%TRIMR(&PRICE '0+'))
  CHGVAR VAR(&DEC) VALUE(&TRIMMED)
  ```

- Trim trailing characters that are specified in a CL variable. Starting from the last character in variable &NAME, trim the character if it matches one of the characters in variable &TCHAR. The resulting substring of &NAME is compared to the literal string ’12345’ and a message is sent if the values are equal.

  ```
  DCL VAR(&NAME) TYPE(*CHAR) LEN(10) VALUE('12345***  ')
  DCL VAR(&TCHAR) TYPE(*CHAR) LEN(2) VALUE('* ')
  IF COND(%TRIMR(&NAME &TCHAR) *EQ '12345') +
     THEN(SNDPGMMSG ('EQUAL!'))
  ```

Related tasks

Changing the value of a variable
You can change the value of a CL variable using the **Change Variable (CHGVAR)** command.

Related reference

Built-in functions for CL
Control language (CL) provides several built-in functions. Built-in functions are used in arithmetic, character string, relational, or logical expressions. All of these built-in functions can only be used in compiled CL programs or procedures.

%UINT built-in function

%UINT converts character, logical, decimal, or integer data to unsigned integer format. The converted value can be assigned to a CL variable, passed as a numeric constant to another program or procedure, or specified as a value for a command parameter of a CL command run from compiled CL.

The %UINT built-in function can be used anywhere that CL supports an arithmetic expression. %UINT can be used alone or as part of a more complex arithmetic expression. For example, %UINT can be used to compare a decimal CL variable to an unsigned integer CL variable in the COND parameter of an **IF** or **WHEN** command. %UINT can also be used to set the value of a CL command parameter, if the associated command object defines the parameter with EXPR(*YES) and TYPE of *DEC, *INT2, *INT4, *UINT2, or *UINT4.

The format of the convert to unsigned integer data built-in function is:

```
%UINT(convert-argument)
```

It can also be formatted as:

```
%UNS(convert-argument)
```

The *convert-argument* must be a CL variable with TYPE of *CHAR, *LGL, *DEC or *INT.

If the *convert-argument* parameter is a character variable, the following rules apply:

- The sign is optional. It can only be '+'. It can precede or follow the numeric data.
- The decimal point is optional. It can be either a period or a comma.
- Leading and trailing blanks are allowed in the data. For example, ' +3 ' is a valid parameter.
- All-blank value will return a zero value.
- If invalid numeric data is found, an exception occurs with CPF0818.

The result will be in unsigned integer format, any decimal digits are truncated without rounding. And the size of the result is always 4 bytes.

The following are examples of using the %UINT built-in function:

Note: *%UNS is allowed for compatibility with RPG but code examples will use %UINT.

- Convert character variable

```
DCL VAR(&POINTS) TYPE(*CHAR) LEN(10) VALUE('+123.45')
DCL VAR(&ANSWER) TYPE(*UINT)

/* &ANSWER will have the value 223 */
CHGVAR VAR(&ANSWER) VALUE(100 + %UINT(&POINTS))
```

- Convert logical variable

```
DCL VAR(&ANSWER1) TYPE(*LGL) VALUE('1')
DCL VAR(&ANSWER2) TYPE(*LGL) VALUE('1')
DCL VAR(&NUM) TYPE(*UINT)

/* &NUM will have the value 2. */
CHGVAR VAR(&NUM) VALUE(%UINT(&ANSWER1) + %UINT(&ANSWER2))
SNDPGMMSG MSG('The number of YES answers is' *BCAT %CHAR(&NUM))
```

- Convert packed decimal variable

```
DCL VAR(&POINTS1) TYPE(*DEC) LEN(5 2) VALUE(100.23)
DCL VAR(&POINTS2) TYPE(*DEC) LEN(5 2) VALUE(100.45)

IF (%UINT(&POINTS1) *EQ %UINT(&POINTS2)) +
THEN(SNDPGMMSG ('The scores are the same!'))
```

- Convert integer variable

```
DCL VAR(&P1) TYPE(*UINT) LEN(2)
DCL VAR(&P2) TYPE(*INT) LEN(2) VALUE(1)

CHGVAR VAR(&P1) VALUE(%UINT(&P2))
```

- Decimal digits will be truncated without rounding

```
DCL VAR(&STRING) TYPE(*CHAR) LEN(10) VALUE('+123.9')
DCL VAR(&ANSWER) TYPE(*UINT)

/* &ANSWER will have the value 123 */
CHGVAR VAR(&ANSWER) VALUE(%UINT(&STRING))
```

Related tasks

Changing the value of a variable

You can change the value of a CL variable using the **Change Variable (CHGVAR)** command.

Related reference

Built-in functions for CL

Control language (CL) provides several built-in functions. Built-in functions are used in arithmetic, character string, relational, or logical expressions. All of these built-in functions can only be used in compiled CL programs or procedures.

%UPPER built-in function

The %UPPER built-in function returns a character string that is the same length as the argument specified with each lowercase letter replaced by the corresponding uppercase letter.

The %UPPER built-in function can be used anywhere that CL supports a character expression. %UPPER can be used alone or as part of a more complex character expression. For example, %UPPER can be used to convert character CL variables to have only upper case characters before comparing them in the COND parameter of an **IF** or **WHEN** command. %UPPER can also be used to set the value of a CL command parameter, if the associated command object defines the parameter with EXPR(*YES) and TYPE of *CHAR, *NAME, *SNAME, *CNAME, *PNAME, *GENERIC, *DATE, *TIME, or *X.

The format of the convert to uppercase built-in function is:

```
%UPPER(input-string [CCSID])
```

The *input-string* must be a CL variable with TYPE of *CHAR.

The *CCSID* parameter is optional and defaults to the job CCSID. The *CCSID* specifies the coded character set identifier (CCSID) of the input string to be converted. Case conversion is performed based on this *CCSID*. The *CCSID*, if specified, must be a CL integer variable or a CL decimal variable with zero decimal positions or a numeric literal with zero decimal positions. The valid values are:

- 0: The CCSID of the job is used to determine the CCSID of the data to be converted. If the job CCSID is 65535, the CCSID from the default CCSID (DFTCCSID) job attribute is used.
- 1-65533: A valid CCSID in this range.

The following are examples of using the %UPPER built-in function:

- Convert to uppercase

```
DCL VAR(&STR) TYPE(*CHAR) LEN(12) VALUE('Hello World!')
/* 'HELLO WORLD!' is to be sent */
SNDPGMMSG (%UPPER(&STR))
```

- Convert to uppercase based on CCSID

```
/* define &STR as 'Hello World!' in CCSID 819 (ISO/ANSI Multilingual) */
DCL VAR(&STR) TYPE(*CHAR) LEN(12) +
              VALUE(X'48656C6C6F20576F726C6421')
/* &STR will have the value x'48454C4C4F20574F524C4421' ('HELLO WORLD!') */
CHGVAR VAR(&STR) VALUE(%UPPER(&STR 819))
```

Related tasks

Changing the value of a variable

You can change the value of a CL variable using the **Change Variable (CHGVAR)** command.

Related reference

Built-in functions for CL

Control language (CL) provides several built-in functions. Built-in functions are used in arithmetic, character string, relational, or logical expressions. All of these built-in functions can only be used in compiled CL programs or procedures.

Monitor Message command

The **Monitor Message (MONMSG)** command is used to monitor for escape, notify, or status messages that are sent to the call stack of the CL program or procedure in which the **MONMSG** command is used.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 610.

Escape messages are sent to a CL program or procedure by the commands in the CL program or procedure and by the program or procedure they call. These escape messages are sent to tell the programs or procedures that errors were detected and that requested functions were not performed. CL programs or 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 program or procedure tries to move a data area that has been deleted, an object-not-found escape message is sent to the program or procedure by the **Move Object (MOVOBJ)** command.

Using the **MONMSG** command, you can direct a program or procedure to take predetermined action if specific errors occur during the processing of the immediately preceding command. 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 program or 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 invocation queue of the program or procedure 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 that are sent by any commands in a CL program or procedure. The following example includes two **MONMSG** commands. The first **MONMSG** command monitors for messages CPF0001 and CPF1999; these messages can be sent by any commands that are run later in the program or procedure. When either message is received from any of the commands running in the program or 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 programs or procedures.

All error conditions monitored for by the **MONMSG** command without the EXEC parameter specified (CPF2105 or MCH1211) are ignored, and program or 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 program or 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. If the GOTO command is run in your program, the subroutine stack will be reset.

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 program or procedure, the program or procedure branches to EXIT2. If message CPF1999 is received on any command, including CMDB, the program or 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 program or procedure, you must decide which ones you want to monitor for and want to handle. Most of these messages are sent to a program or procedure only if an error is detected in the program or procedure. Others are sent because of conditions outside the program or procedure. Generally, a CL program or procedure needs to monitor for those messages that pertain to its basic function and that it can handle. For all other messages, the IBM i operating system assumes an error has occurred and takes default actions.

Related tasks

Defining message descriptions
Predefined messages are stored in a message file.

Messages
Messages are used to communicate between users and programs.

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.

Related information

CL command finder

Monitor Message (MONMSG) command

Retrieving values that can be used as variables

You can retrieve values, such as system values and job attributes, to use as variables in a program or procedure. Note that not every retrievable value is included.

Related information

CL command finder

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 the IBM i operating system is started.

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

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

The CL variable for returned value (RTNVAR) parameter specifies the name of the variable in your CL program or 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.

Related information

System values

Example: Retrieving QTIME system value

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

Note: By using the code example, you agree to the terms of the "Code license and disclaimer information" on page 610.

```
      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
```

Related information

System values

Retrieving the QDATE system value into a CL variable

In many applications, you might want to use the current date in your program or procedure by retrieving the system value QDATE and placing it in a variable. You might also want to change the format of that date for use in your program or procedure.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 610.

To convert the format of a date in a CL program or procedure, use the **Convert Date (CVTDAT)** command.

The format for the system date is the system value QDATFMT. The included value of QDATFMT varies according to country or region. For example, 062488 is the MDY (month day year) 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 three-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 **Convert Date (CVTDAT)** command.

The format for the **Convert Date (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. After 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 **Convert Date (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 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 six 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 eight characters when using no separators.

```
July 28, 1978 would be written as 07281978.
```

b. Use ten characters when using separators.

```
July 28, 1978 would be written as 07-28-1978.
```

3. For Julian dates with 2–digit years

a. Use five characters when using no separators.

```
December 31, 1996 would be written as 96365.
```

b. Use six characters when using separators.

```
December 31, 1996 would be written as 96-365.
```

4. For Julian dates with 4–digit years,

a. Seven characters are required when no separators are used.

```
February 4, 1997 would be written as 1997035.
```

b. Eight 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 can be either 2-byte or 4-byte fields. All converted values are right-aligned 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-aligned 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
```

Related information

Get Current Local Time (CEELOCT) API

Application Programming Interfaces (API)

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 that is generated can be used for the following purposes:

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

Related information

Retrieve Configuration Source (RTVCFGSRC) command

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 program or procedure to check the status of a configuration description.

Related information

Retrieve Configuration Status (RTVCFGSTS) command

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.

Related information

Retrieve Network Attributes (RTVNETA) command

Change Network Attributes (CHGNETA) command

Display Network Attributes (DSPNETA) command

Retrieve Network Attributes (QWCRNETA) API

Example: Using the Retrieve Network Attributes command

This example shows how to retrieve network attributes by using the **Retrieve Network Attributes (RTVNETA)** command.

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

To retrieve the job attributes and place their values in a CL variable to control your applications, use the **Retrieve Job Attributes (RTVJOBA)** command. With this command you can retrieve all job attributes or any combination of them.

In the following CL procedure, a **Retrieve Job Attributes (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 **Retrieve Job Attributes (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.

Related information

Retrieve Job Attributes (RTVJOBA) command

Example: Using the Retrieve Job Attributes command

This example shows how to retrieve job attributes by using the **Retrieve Job Attributes (RTVJOBA)** command.

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 **Retrieve Job Attributes (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 **Retrieve Job Attributes (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 **Submit Job (SBMJOB)** command, so that the submitted job will have the same attributes as the submitting job.

Retrieving user profile attributes

Using the **Retrieve User Profile (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 **Retrieve User Profile(RTVUSRPRF)** command.

Related information

Retrieve User Profile (RTVUSRPRF) command

Example: Using the Retrieve User Profile command

This example shows how to retrieve user profile information by using the **Retrieve User Profile (RTVUSRPRF)** command.

In the following CL procedure, a **Retrieve User Profile(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.

Related information

Retrieve Member Description (RTVMBRD) command

Example: Using the Retrieve Member Description command

This example shows how to retrieve member description information by using the **Retrieve Member Description (RTVMBRD)** command.

In the following CL procedure, a **Retrieve User Profile Attributes (RTVUSRPRF)** 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.

Compiling CL source program

A CL source program must be compiled before it can be run.

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

You can also create a CL 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.

You can use the **Include CL Source (INCLUDE)** command to embed another source file during compilation.

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.

An other example creates the program IFSTEST and places it in library QGPL from an IFS stream source file:

```
CRTBNDCL PGM(QGPL/IFSTEST) SRCSTMF('/home/ifstest/ifstest.clp')
```

The source commands for IFSTEST are in the IFS stream source file /home/ifstest/ifstest.clp. By default, a compiler listing is created.

On the **Create CL Program (CRTCLPGM)** command or 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.

Related information

CL command finder
Create Program (CRTPGM) command
Create Bound CL Program (CRTBNDCL) command
Create CL Module (CRTCLMOD) command
Create Service Program (CRTSRVPGM) command
Include CL Source (INCLUDE) command
Retrieve CL Source (RTVCLSRC) command

Setting create options in the CL source program

With the **Declare Processing Options (DCLPRCOPT)** command, you can set many of the same create option parameters that are on the CL compiler commands. The CL compiler commands include **Create CL Program (CRTCLPGM)**, **Create CL Module (CRTCLMOD)**, and **Create Bound CL Program (CRTBNDCL)**.

Values specified on the **DCLPRCOPT** command take precedence over any values specified on the CL compiler commands. In addition, the **DCLPRCOPT** command supports the Binding directory (BNDDIR) and Binding service program (BNDSRVPGM) parameters that can be used when running the **CRTBNDCL** command. The following examples show how the **DCLPRCOPT** command can be used:

Example: Declaring compiler options to override CRTCLPGM

```
DCLPRCOPT    ALWRTVSRC(*NO)  USRPRF(*OWNER)
```

This command overrides the Allow RTVCLSRC (ALWRTVSRC) and User profile (USRPRF) parameter values that are specified on the **CRTCLPGM** command. The resulting CL program does not allow the CL source code to be retrieved from the *PGM object. When the program object is called, it adopts the authorities of the user profile that owns the *PGM object.

Example: Declaring compiler options to override CRTCLMOD

```
DCLPRCOPT    LOG(*NO)  AUT(*USE)
```

This command overrides the Log commands (LOG) and Authority (AUT) parameter values that are specified on the **CRTCLMOD** command. When the resulting ILE CL module is bound into an ILE program or service program and the ILE CL procedure is called, CL commands that run from this procedure are not logged in the job log. The public authority for the *MODULE object that is created by the **CRTCLMOD** command is *USE.

Example: Declaring compiler options to override CRTBNDCL

```
DCLPRCOPT    ALWRTVSRC(*NO)  DFTACTGRP(*NO)  ACTGRP(MYAPP)  +
             BNDDIR(MYAPPLIB/MYBNDDIR)
```

This command overrides the Allow RTVCLSRC (ALWRTVSRC), Default activation group (DFTACTGRP), Activation group (ACTGRP), and Binding directory (BNDDIR) parameter values that are specified on the **CRTBNDCL** command. The resulting ILE CL program does not contain CL source that can be retrieved using the **RTVCLSRC** command. The program runs in the activation group named MYAPP. When the **CRTPGM** command is used to create the bound CL program, the **CRTBNDCL** command adds binding directory MYBNDDIR in library MYAPPLIB to the Binding directory (BNDDIR) parameter. In this way, the service programs and ILE modules that are referenced by that binding directory can be used to resolve ILE procedures used in the ILE CL program.

Related information

Create CL Program (CRTCLPGM) command
Create Bound CL Program (CRTBNDCL) command
Create CL Module (CRTCLMOD) command
Declare Processing Options (DCLPRCOPT) command

Embedding CL commands from another source member

You can use the **Include CL Source (INCLUDE)** command to split your CL source code, so that the CL source code can be compiled across multiple source file members.

The CL source to be embedded can be located in another member of the same source file that is identified on the Source file (SRCFILE) parameter of the CL compiler commands or a different source file. The CL compiler commands include **Create CL Program (CRTCLPGM)**, **Create CL Module (CRTCLMOD)**, and **Create Bound CL Program (CRTBNDCL)**.

The CL source to be embedded can also be located in an IFS stream file. If your main source file is also an IFS stream file, you can specify one or more directories in parameter Include directory (INCDIR) of **Create CL Module (CRTCLMOD)** command or **Create Bound CL Program (CRTBNDCL)** command, for the search paths used by the compiler to find IFS stream including files.

No matter your main source file is a database file or an IFS source file, you can specify the path name of the stream file to be included by **Include CL Source (INCLUDE)** command in the source statement.

```
INCLUDE   SRCSTMF('/dir/pay_data.cl')
```

You can run the **Retrieve CL Source (RTVCLSRC)** command at a later time to retrieve either the original CL source (which contains just the INCLUDE commands) or the expanded CL source (which contains the embedded CL source commands).

Related information

Create CL Program (CRTCLPGM) command
Create Bound CL Program (CRTBNDCL) command
Create CL Module (CRTCLMOD) command
Include CL Source (INCLUDE) command
Retrieve CL Source (RTVCLSRC) command

Logging CL program or procedure commands

You can specify that most CL commands that are run in a CL program or procedure be written (logged) to the job log.

To log CL commands, specify 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 CL source program 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 **Change Job (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 program or 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 before running the RMVMSG command do not appear in the job log. This affects only the CL program or 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. The following list is about commands that are not logged.

CALLPRC	**CALLSUBR**	**CHGVAR**
DCL	**DCLF**	**DCLPRCOPT**
DO	**DOFOR**	**DOUNTIL**
DOWHILE	**ELSE**	**ENDDO**
ENDPGM	**ENDSELECT**	**ENDSUBR**
GOTO	**IF**	**INCLUDE**
ITERATE	**LEAVE**	**MONMSG**
OTHERWISE	**PGM**	**RTNSUBR**
SELECT	**SUBR**	**WHEN**

If the logging option is on, logging messages are sent to the message queue of the CL program or procedure. If the CL program or 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.

Related concepts
Job log

Each job has an associated job log.

Related information

CL command finder
Create CL Module (CRTCLMOD) command
Create Bound CL Program (CRTBNDCL) command
Change Job (CHGJOB) command
Remove Message (RMVMSG) command

Retrieving CL source code

Depending on the CL compiler options used, you can retrieve the CL source code from a CL program or CL module object.

The CL program or CL module must have been created specifying *YES for the `Allow RTVCLSRC (ALWRTVSRC)` parameter.

The ability to retrieve the CL source from a CL program or CL module can make it easier to diagnose and fix problems in CL code on systems where the original CL source code is not available.

There are CL commands and callable API programs that you can use to determine if the CL program or CL module was created with ALWRTVSRC(*YES).

- If the program (*PGM) object was created using the **Create CL Program (CRTCLPGM)** command, you can use the **Display Program (DSPPGM)** command or the **Retrieve Program Information (QCLRPGMI)** API to see the value that was specified for the ALWRTVSRC parameter when the program was created.
- If the module (*MODULE) object was created using the **Create CL Module (CRTCLMOD)** command, you can use the **Display Module (DSPMOD)** command or the **Retrieve Module Information (QBNRMODI)** API to see the value that was specified for the ALWRTVSRC parameter when the module was created.
- If the CL module was bound into an Integrated Language Environment (ILE) program using the **Create Bound CL Program (CRTBNDCL)** command or the **Create Program (CRTPGM)** command, you can use the **Display Program (DSPPGM)** command with DETAIL(*MODULE) or the **List Program Information (QBNLPGMI)** API to see the value that was specified for the ALWRTVSRC parameter when the module was created.
- If the CL module was bound into an ILE service program (*SRVPGM) object using the **Create Service Program (CRTSRVPGM)** command, you can use the **Display Service Program (DSPSRVPGM)** command with DETAIL(*MODULE) or the **List Service Program Information (QBNLSPGM)** API to see the value that was specified for the ALWRTVSRC parameter when the module was created.

The shipped default value for the ALWRTVSRC parameter is *YES. The ability to save the CL source with ILE CL modules was added in the IBM i 7.1 release of the operating system.

Retrieving CL source from a module in an ILE CL program

```
RTVCLSRC   PGM(MYCLPGM)  MODULE(MOD1)  SRCFILE(MYLIB/QCLSRC)
```

This command retrieves the CL source from module MOD1 in ILE program MYCLPGM. The retrieved CL source is stored in member MOD1 of the source physical file QCLSRC located in library MYLIB.

Related information

Create CL Program (CRTCLPGM) command
Create Bound CL Program (CRTBNDCL) command
Create CL Module (CRTCLMOD) command
Create Program (CRTPGM) command
Create Service Program (CRTSRVPGM) command
Display Program (DSPPGM) command

Display Service Program (DSPSRVPGM) command
Display Module (DSPMOD) command
Retrieve CL Source (RTVCLSRC) command
Retrieve Program Information (QCLRPGMI) API
Retrieve Module Information (QBNRMODI) API
List Program Information (QBNLPGMI) API
List Service Program Information (QBNLSPGM) API

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 example is about compiler listing. The callout numbers refer to descriptions following the listing.

Note: By using the code example, you agree to the terms of the “Code license and disclaimer information” on page 610.

```
 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 i5/OS 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 the IBM i operating system.

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.

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.

Related concepts

Common compilation errors
The types of errors that are detected at compile time include syntax errors, references to variables and labels not defined, and missing statements.

Related information

Create CL Module (CRTCLMOD) command

Common compilation errors

The types of errors that are detected at compile time include syntax errors, references to variables and labels not defined, and missing statements.

In the CL compiler listing, an error condition that relates directly to a specific command is listed after that command. 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 following types of errors stop the program or module 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 program or 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.

Related reference

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.

Obtaining a CL dump

You can obtain a CL program or CL procedure dump while the CL program or CL procedure is running.

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

To obtain a CL dump, complete one of the following tasks:

- Run the **`Dump CL Program (DMPCLPGM)`** command. This command can only be used in a CL program or CL procedure and does not end the CL program or 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 program or 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. 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 program or 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  11:05:032 Page 1

Job name . . . . . . . . :  DSP043  User name . . . . . . :   SMITH 3  Job number  . . . . . . : 01329 3
Program name . . . . . . :  DUMP4   Library . . . . . . . :   MYLIB4   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 the IBM i operating system.

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.

Related information
Controlling the default reply to the query governor inquiry message
Dump CL Program (DMPCLPGM) command

Displaying module attributes

To display the attributes of any module object, including a CL module, use the **Display Module (DSPMOD)** command.

The information displayed or printed can be used to determine the options specified on the command used to create the module.

Related information
Display Module (DSPMON) command

Displaying program attributes

To display the attributes of any program object, including a CL program, use the **Display Program (DSPPGM)** command.

The information displayed or printed can be used to determine the options specified on the command used to create the program.

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 (QCLRPGMI) API`.

Related information

Display Program (DSPPGM) command

Return code summary

A return code can be returned using the Return code (RTNCDE) parameter on the **Retrieve Job Attributes (RTVJOBA)** command .

The return code 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 **Retrieve Job Attributes (RTVJOBA)** command.

The following list summarizes the return codes used by languages supported on the IBM i operating system:

- 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 and OPM COBOL programs

  The return codes sent by running COBOL 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 programs cannot retrieve these return codes. For OPM COBOL, a return code value of 2 sends message LBE9001. For ILE COBOL, a return code value of 2 sends message CEE9001.
- ILE C programs

  The current value of the integer return code is returned by the last ILE C return statement in an ILE C program.

Related information

Retrieve Job Attributes (RTVJOBA) command

Reclaim Resources (RCLRSC) command

Compiling source programs for a previous release

The CL compiler commands allow 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 IBM i operating system the CL program will be run. You can specify *CURRENT, *PRV, or a specific release level.

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

Related information

Create CL Program (CRTCLPGM) command

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 QUSRV7R1M0 library supports a system that runs Version 7 Release 1 Modification level 0 of the IBM i 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. After 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.

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

Related information

Save Object (SAVOBJ) command

Save Changed Object (SAVCHGOBJ) command

Save Library (SAVLIB) command

Installing CL compiler support for a previous release

You can install the *PRV CL compiler support and QSYSVxRxMx libraries.

To install the *PRV CL compiler support, follow these steps:

1. To view the Licensed Program Menu, enter the following command:

```
GO LICPGM
```

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 the following command:

```
DLTLICPGM LICPGM(5761SS1) 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.

Controlling flow and communicating between programs and procedures

The **Call Program (CALL)**, **Call Bound Procedure (CALLPRC)**, and **Return (RETURN)** commands pass control back and forth between programs and procedures.

Each command has slightly different characteristics. Information can 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.

Related concepts

Parts of a CL source program
Although each source statement that is entered as part of a CL source program is actually a CL command, the source can be divided into the basic parts that are used in many typical CL source programs.

Variables in CL commands
A *variable* is a named changeable value that can be accessed or changed by referring to its name.

Related tasks

Displaying the call stack
To display the call stack, use the **Display Debug (DSPDBG)** command.

Related information

Call Bound Procedure (CALLPRC) command
Call Program (CALL) command
Return (RETURN) command
Security reference

Passing control to another program or procedure

To pass control to another program or procedure, you can select different options.

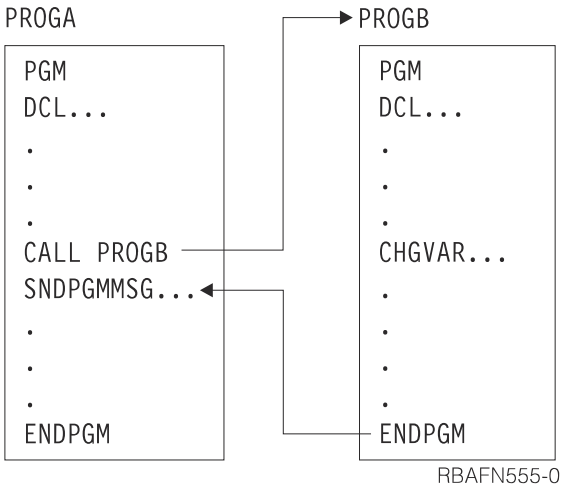
Using the Call Program command to pass control

The **Call Program (CALL)** command calls a program named on the command and passes control to it.

The **CALL** command has the following format:

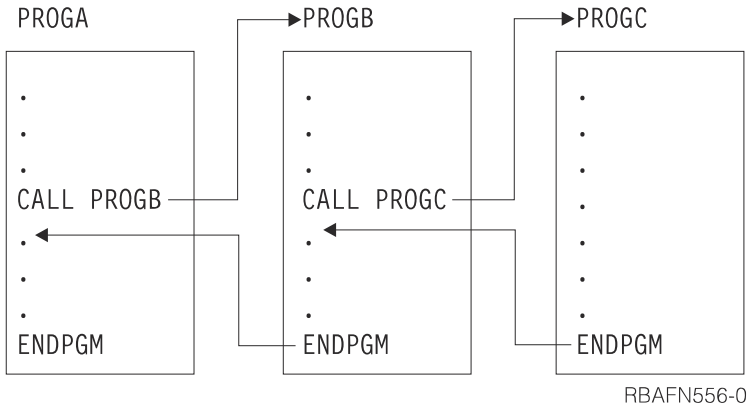
```
CALL PGM(library-name/program-name) PARM(parameter-values)
```

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. When the called program finishes running, control returns to the next command in the calling program.



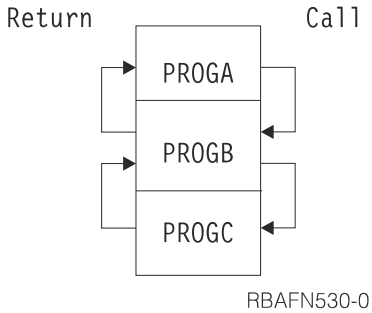
RBAFN555-0

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



RBAFN556-0

In this series, the call stack is as follows.



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 PROGC ends with a RETURN or an ENDPGM command.

A CL program can call itself.

Related tasks

Passing parameters
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.

Using the Call Program command to pass control to a called program
When the **Call Program (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.

Related information

Call Program (CALL) command

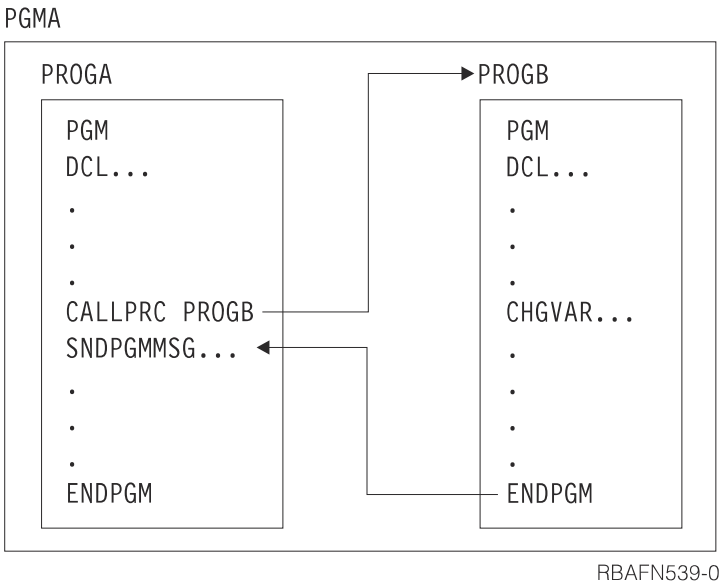
Using the Call Bound Procedure command to pass control

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:

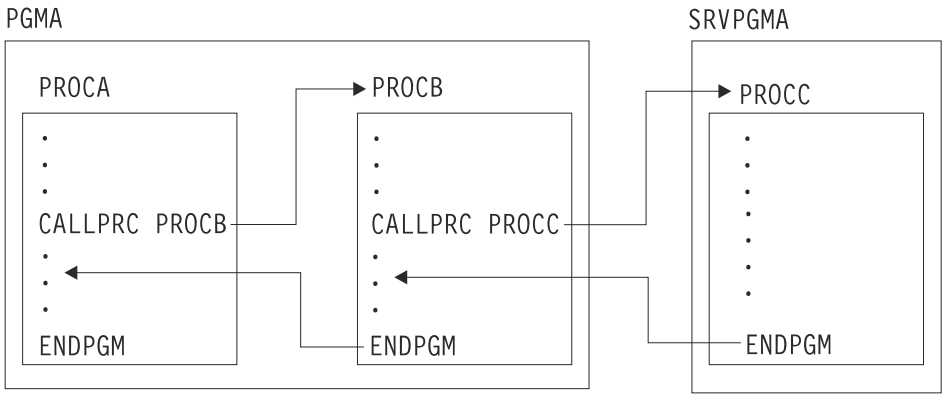
```
CALLPRC PRC(procedure-name) PARM(parameter-values)
RTNVAL(return-value-variable)
```

The procedure name may not be a variable. When the called procedure finishes running, control returns to the next command in the calling procedure.



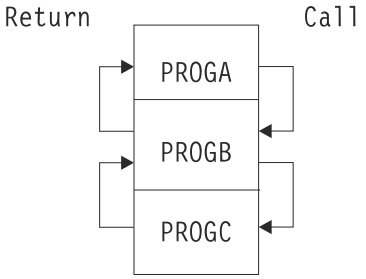
RBAFN539-0

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



RBAGN540-0

In this series, the call stack is as follows.



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 PROGC ends with a **Return (RETURN)** or an **End Program (ENDPGM)** command.

A CL procedure can call itself.

Related tasks

Passing parameters
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.

Related information

Call Bound Procedure (CALLPRC) command

Using the Return command to pass control

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

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

If a **Monitor Message (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 want to avoid this for security reasons.

Related information

Call Bound Procedure (CALLPRC) command
Monitor Message (MONMSG) command
Return (RETURN) command

Passing parameters

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.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 610.

You can specify the information to be passed on the PARM parameter on the **Call (CALL)** command or the **Call Bound Procedure (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 (CALL)** command or the **Call Bound Procedure (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 (CALL)** command or the **Call Bound Procedure (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 **Call Bound Procedure (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.

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 **Call Bound Procedure (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 an original program model (OPM) CL program is called, 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 program or procedure calls. Calling a program or 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 programs or procedures written in other ILE languages that call CL programs or procedures or are called by CL programs or procedures.

This also gives you more flexibility when you write ILE CL programs or procedures, because you can write programs or 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 programs or procedure need not to 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 on the **Call Bound Procedure (CALLPRC)** command. If you specify *OMIT for a parameter, the calling program or procedure passes a null pointer. The program or 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 omissible parameter. The program or procedure must take appropriate action if it receives a MCH3601.

Note: The special value *OMIT is only valid for the PARM parameter on the **Call Bound Procedure (CALLPRC)** command.

When calling programs or procedures, you can pass arguments by reference and by value.

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
```

Related tasks

Using the Call Program command to pass control

The **Call Program (CALL)** command calls a program named on the command and passes control to it.

Using the Call Bound Procedure command to pass control

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

Example: Using the Transfer Control command

This is an example of transferring control to improve performance.

Related information

Call Bound Procedure (CALLPRC) command

Call Program (CALL) command

Retrieve Operational Descriptor Information (CEEDOD) API

ILE Concepts

Using the Call Program command to pass control to a called program

When the **Call Program (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 Integrated Language Environment (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 single quotation marks 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.

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 single quotation marks.
- 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 high-level language (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. Single quotation marks 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`.

Related concepts

Length of CL command parameter value
The length (LEN) parameter is used to specify the length of a parameter.

Related tasks

Using the Call Program command to pass control
The **Call Program (CALL)** command calls a program named on the command and passes control to it.

Related information

Call Program (CALL) command
CL command finder

Common errors when calling programs and procedures

In passing values on a **Call Program (CALL)** command or a **Call Bound Procedure (CALLPRC)** command, you might encounter some errors. Some of these errors can be very difficult to debug, and some have serious consequences for program functions.

Data type errors using the CALL command

When you use the **Call Program (CALL)** command, data type errors might occur.

The total length of the command string includes the command name, spaces, parameter names, parentheses, contents of variables and single quotation marks 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.

When the **Call Program (CALL)** command is used with the CMD parameter on the **Submit Job (SBMJOB)** command, unexpected results might 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 Program (CALL)** command is converted to a command string that is run at a later time when the batch subsystem initiates it. When the **Call Program (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, typically 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 Program (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 **Submit Job (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 nonblank 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 Program (CALL)** command. The command definition ensures the parameters are passed to the command processing program as expected.

Related concepts

Variables in CL commands

A *variable* is a named changeable value that can be accessed or changed by referring to its name.

Related information

Call Program (CALL) command

Submit Job (SBMJOB) command

CL command finder

Data type errors when passing parameters

In addition to errors that occur using the **Call Program (CALL)** command, other types of data type errors can occur.

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 single quotation marks, it is passed as a character string. However, if the constant is not enclosed in single quotation marks, 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 runtime 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
When a decimal value is passed with incorrect decimal length and precision, errors might occur.

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 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
When you pass character values with incorrect length, errors might occur.

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 (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 **Call Procedure (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
```

Related information

Call Bound Procedure (CALLPRC) command

Call Program (CALL) command

Communicating between programs and procedures

Programs and procedures can communicate by using data queues and data areas.

Using data queues

Data queues are a type of system object that you can create, to which one high-level language (HLL) procedure or program can send data, and from which another HLL procedure or program can receive data.

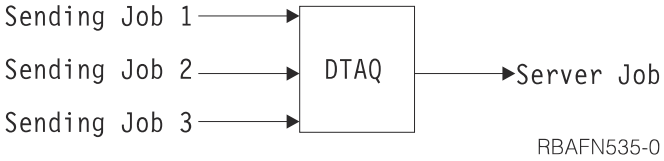
The receiving program can be already waiting for the data, or can receive the data later.

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, clear, retrieve the contents or description of a data queue and change some attributes of a data queue in any HLL procedures or programs by calling APIs QSNDDTAQ, QRCVDTAQ, QCLRDTAQ, QMHRDQM, QMHQRDQD, and QMHQCDQ without exiting the HLL procedure or program or calling a CL procedure or program to perform the data queue operation.
- 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 required 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.

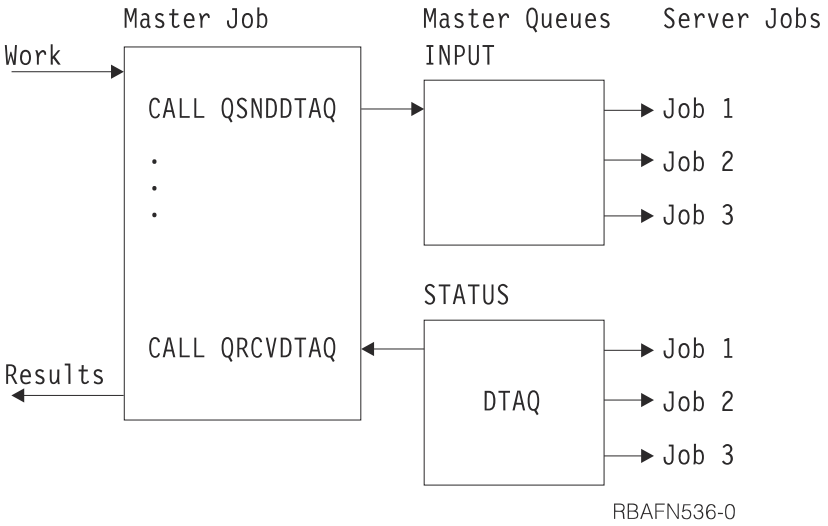
In addition to these advantages, you can journal your data queues. This allows you to recover the object to a consistent state, even if the object was in the middle of some change action when the abnormal initial program load (IPL) or crash occurred. Journaling also provides for replication of the data 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.

The following example shows 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

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.

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.

Related concepts

Basic printing

Related reference

Example: Waiting for input from a display file and an ICF file

This example shows a program waiting for input from a display file and an ICF file, using a data queue.

Example: Waiting for input from a display file and a data queue

This example shows a program in a job waiting for input from a display file and for input on a data queue in another job.

Related information

CL command finder

Journal management

Change Output Queue (CHGOUTQ) command

Create Output Queue (CRTOUTQ) command

Remote data queues

Remote data queues are data queues that reside on a remote system.

You can access remote data queues with distributed data management (DDM) files. DDM files make it possible for a program residing on one system to access a data queue on a remote system 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.

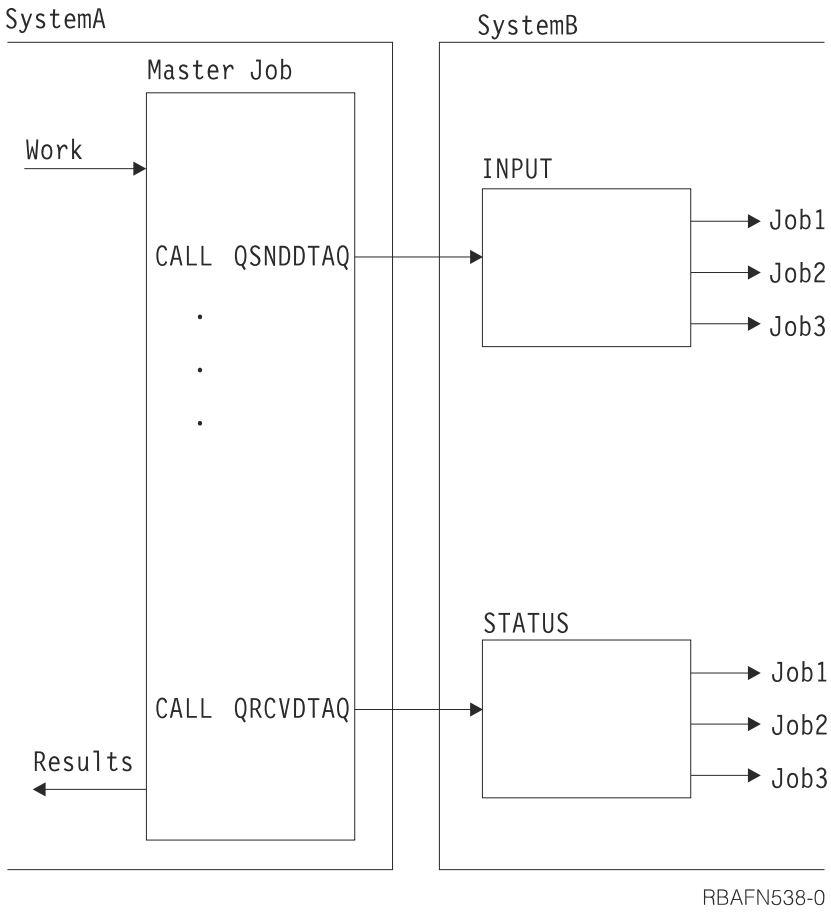


Figure 1. Example of accessing a remote data queue

Related information

Create Data Queue (CRTDTAQ) command

Distributed database programming

Comparisons with using database files as queues

Using data queues and using database files differ in many ways.

- Data queues have been improved to communicate between active procedures and programs, not to store large volumes of data or large numbers of entries.
- 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. Recreating 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.

Related tasks

Managing the storage used by a data queue

Because smaller data queues have better performance than large ones, it is important to manage the size of the storage allocated to a data queue.

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.

Related information

Create Data Queue (CRTDTAQ) command

Managing the storage used by a data queue

Because smaller data queues have better performance than large ones, it is important to manage the size of the storage allocated to 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 **Create Data Queue (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 need to delete and re-create it as described in the preceding paragraph or use the Change Data Queue (QMHQCDQ) API to change the automatic reclaim attribute. Automatic reclaim might be expensive depending on the size of the queue, so the initial number of entries specified on the **Create Data Queue (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 run the reclaim function more frequently.

Related concepts

Comparisons with using database files as queues

Using data queues and using database files differ in many ways.

Related information

Create Data Queue (CRTDTAQ) command
Delete Data Queue (DLTDTAQ) command

Allocating data queues

If all users of a data queue allocate it before using it, this helps to ensure that a data queue is not accessed by more than one job at a time.

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 error message CPF1002. The **Monitor Message (MONMSG)** command can be used in the application program or procedure to monitor for this message and to respond to the error message. Possible responses include sending a message to the user and attempting to allocate the data queue again.

A data queue can be changed to indicate that IBM-supplied operations enforce a lock on a data queue. Before V6R1, locks placed on a data queue by the ALCOBJ function are ignored by IBM-supplied operations. Performance of the data queue can be impacted with the use of this attribute to enforce data queue locks. See the Change Data Queue (QMHQCDQ) API for information about how to change a data queue to enforce data queue locks and to obtain details on the locks to be used for the various data queue operations.

Related tasks

Monitoring for messages in a CL program or procedure
Messages that can be monitored are *ESCAPE, *STATUS, and *NOTIFY messages that are issued by each CL command used in the program or procedure.

Related information

Allocate Object (ALCOBJ) command
Deallocate Object (DLCOBJ) command
Monitor Message (MONMSG) command
Change Data Queue (QMHQCDQ) API

Examples: Using a data queue

These examples illustrate various methods for processing data queues.

Example: Waiting up to two hours to receive data from data queue

This example shows a program waiting up to two hours to receive an entry from a data queue.

In the following example, program B specifies to wait up to two 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 two hours, program B receives the entries from this data queue. Processing begins immediately. If two 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.

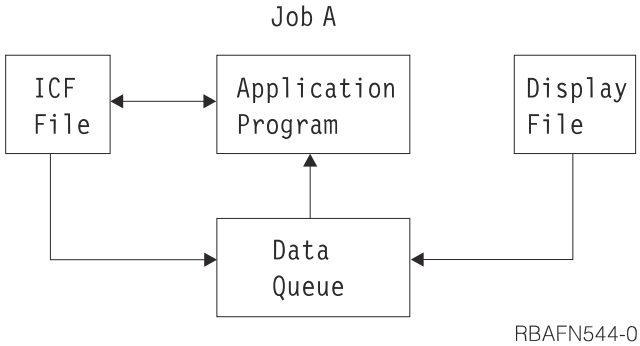
In program A, the following statements relate to the data queue:

```
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: Waiting for input from a display file and an ICF file
This example shows a program waiting for input from a display file and an ICF file, using a data queue.



The following example is different from the typical 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 before 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
```

Related tasks

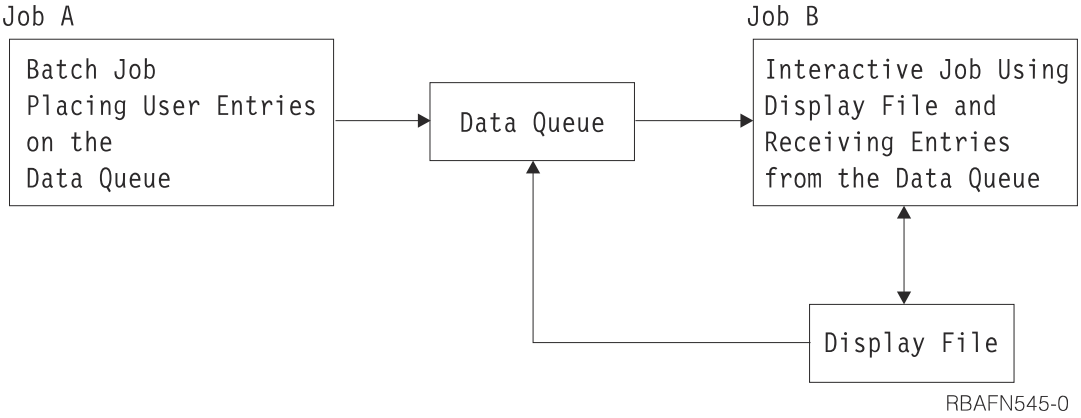
Using data queues

Data queues are a type of system object that you can create, to which one high-level language (HLL) procedure or program can send data, and from which another HLL procedure or program can receive data.

Example: Waiting for input from a display file and a data queue

This example shows a program in a job waiting for input from a display file and for input on a data queue in another job.

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
```

Related tasks

Using data queues

Data queues are a type of system object that you can create, to which one high-level language (HLL) procedure or program can send data, and from which another HLL procedure or program can receive data.

Creating data queues associated with an output queue

You can associate a data queue with an output queue so that when a spooled file on the output queue goes to a ready (RDY) status, an entry is sent to the data queue.

To view the layout or content of the data queue entry that is sent, see Record type 01 data queue entry format.

To associate the data queue and output 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 is *FIFO or *LIFO. Then, on the DTAQ keyword of the **Create Output Queue (CRTOUTQ)** command or the **Change Output Queue (CHGOUTQ)** command, specify the data queue that is created with the **CRTDTAQ** command.

Related information

Create Data Queue (CRTDTAQ) command

Data queue support

Creating data queues associated with jobs

You can associate a data queue with exit point QIBM_QWT_JOBNOTIFY to place an entry on a data queue for certain job transitions.

An entry can be placed on a data queue when a job is placed on a job queue, when a job starts or a when a job ends. See the Job Notification Exit Point (QIBM_QWT_JOBNOTIFY) for details on how to create the data queue and the format of the information placed on the data queue.

Using data areas

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.

Related information

Create Data Area (CRTDTAARA) command
Retrieve Data Area (RTVDTAARA) command
Change Data Area (CHGDTAARA) command
Display Data Area (DSPDTAARA) command
Delete Data Area (DLTDTAARA) command
Journal management

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.

Related information

Create Data Area (CRTDTAARA) command
Retrieve Data Area (RTVDTAARA) command
Change Data Area (CHGDTAARA) command
Display Data Area (DSPDTAARA) command
Delete Data Area (DLTDTAARA) command

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.

Related information

CL command finder
Change Group Attributes (CHGGRPA) command
Create Data Area (CRTDTAARA) command
Retrieve Data Area (RTVDTAARA) command
Change Data Area (CHGDTAARA) command
Display Data Area (DSPDTAARA) command
Delete Data Area (DLTDTAARA) command

Program Initialization Parameter data area

A Program Initialization Parameter (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.

Related information

Create Data Area (CRTDTAARA) command
Retrieve Data Area (RTVDTAARA) command
Change Data Area (CHGDTAARA) command
Display Data Area (DSPDTAARA) command
Delete Data Area (DLTDTAARA) command

Remote data areas

A remote data area is a data area on a remote system.

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 system when it retrieves data that resides on a remote system. To ensure that you are accessing the correct data area, you might need to do one of the following tasks:

- 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 system 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.

Related information

Distributed database programming
Create Data Area (CRTDTAARA) command
Retrieve Data Area (RTVDTAARA) command
Change Data Area (CHGDTAARA) command
Display Data Area (DSPDTAARA) command
Delete Data Area (DLTDTAARA) command

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')
```

Related information

Create Data Area (CRTDTAARA) command

Data area locking and allocation

Locking and allocating a data area helps to ensure that the data area is not accessed by more than one job at a time.

The **Change Data Area (CHGDTAARA)** command uses a *SHRUPD (shared for update) lock on the data area during command processing. The **Retrieve Data Area (RTVDTAARA)** and **Display Data Area (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.

For information about handling data areas in other (non-CL) languages, refer to the appropriate high-level language (HLL) reference manual.

Related concepts

Objects and libraries

Tasks and concepts specific to objects and libraries include performing functions on objects, creating libraries, and specifying object authority.

Related information

Create Data Area (CRTDTAARA) command

Retrieve Data Area (RTVDTAARA) command

Display Data Area (DSPDTAARA) command

Displaying a data area

By using the **Display Data Area (DSPDTAARA)** command, you can display the attributes (name, library, type, length, data area text description) and the value of a data area. The display uses the 24-digit format with leading zeros suppressed.

Related information

Display Data Area (DSPDTAARA) command

Changing a data area

To change the value of a data area, use the **Change Data Area (CHGDTAARA)** command.

The **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 program or procedure, the data area does not need to exist when the program or procedure is created.

Related information

Change Data Area (CHGDTAARA) command

Retrieving a data area

You can retrieve a data area and copy it to a variable by using the **Retrieve Data Area (RTVDTAARA)** command.

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.

Related information

Retrieve Data Area (RTVDTAARA) command

Examples: Retrieving a data area

These examples show different ways of retrieving a data area.

Example: Retrieving data area ORDINFO

This is an example of using a data area to track the status of an order file.

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 can declare these fields in your program or 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: Retrieving data area DA1
This example shows how to retrieve a data area and copy it into a variable.

The following example of the **Retrieve Data Area (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'.

Related information
Retrieve Data Area (RTVDTAARA) command

Example: Retrieving data area DA2
This example shows how to retrieve a decimal data area and copy it into a decimal variable.

The following example of the **Retrieve Data Area (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).

Related information
Retrieve Data Area (RTVDTAARA) command

Example: Changing and retrieving a data area

This example shows how to change and retrieve a data area.

The following is an example of using the **Change Data Area (CHGDTAARA)** and **Retrieve Data Area (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'`.

Related information

Retrieve Data Area (RTVDTAARA) command

Change Data Area (CHGDTAARA) command

Defining and documenting CL commands

This information helps you define your own CL commands and provide documentation for them.

Defining CL commands

CL commands enable you to request a broad range of functions. You can use IBM-supplied commands, change the default values for command parameters, and define your own 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.

When you define and create your own CL commands, you might also want to provide documentation for them.

Related concepts

Variables that replace reserved or numeric parameter values
Character variables can be used for some commands to represent a value on the command parameter.

Related tasks

Variables to use for specifying a list or qualified name
Variables can be used to specify a list or qualified name.

Documenting a CL command
If you define and create your own CL commands, you can also create online command help to describe your commands.

Related reference

Abbreviations used in CL commands and keywords

Most CL command names that are part of the IBM i operating system and other licensed programs that run on the operating system follow a consistent naming style.

CL command definition statements

Command definition statements allows system users to create additional CL commands to meet specific application needs.

These commands are similar to the system commands. The defined command calls a program to perform some function. Users can define commands by using command definition statements. The defined command can include the following:

- Keyword notation parameters for passing data to programs
- Default values for omitted parameters
- Parameter validity checking so the program performing the function will have correct input
- Prompt text for prompting interactive users

Each command on the system has a Command definition object and a command processing program (CPP).

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.

Table 24. Statements for defining CL commands

Statement type	Statement name	Related command	Description
Command	CMD	Command Definition (CMD) command	Specifies the prompt text for the command being created. The CMD statement can be anywhere in the source file that is referred to by the Create Command (CRTCMD) command. Only one CMD statement can be used in the source file, even if no prompt text is specified for the created command. Several create option parameters on the CRTCMD command can also be specified on the CMD statement.
Parameter	PARM	Parameter Definition (PARM) command	Defines a parameter of a command being created. A parameter is the means by which a value is passed to the command processing program (CPP). One PARM statement must be used for each parameter that appears in the command being defined.
Element	ELEM	Element Definition (ELEM) command	Used to define the elements of a mixed list (list elements) parameter on a command. A list parameter is a parameter that accepts multiple values that are passed together as consecutive values pointed to by a single keyword.

Table 24. Statements for defining CL commands (continued)

Statement type	Statement name	Related command	Description
Qualifier	QUAL	Qualifier Definition (QUAL) command	Describes one part of a qualified name. If a name is the allowed value of a parameter or list element defined in a PARM or ELEM statement, it can be changed to a qualified name by using a QUAL statement for each qualifier used to qualify the name.
Dependency	DEP	Dependent Definition (DEP) command	Defines a required relationship between parameters and parameter values that must be checked. This relationship can refer to either the specific value of a parameter or parameters or to the required presence of parameters.
Prompt control	PMTCTL	Prompt Control Definition (PMTCTL) command	Specifies a condition that is tested to determine if prompting is done for the parameters whose PARM statement refers to the PMTCTL statement.

Related concepts

CL command delimiter characters
Command delimiter characters are special characters or spaces that identify the beginning or end of a group of characters in a command.

Simple and qualified object names
The name of a specific object that is located in a library can be specified as a simple name or as a qualified name.

CL command coding rules
This summary of general information about command coding rules can help you properly code CL commands.

CL command definition
Command definition allows you to create additional control language commands to meet specific application needs. These commands are similar to the system commands.

Naming within commands
The type of name you specify in control language (CL) determines the characters you can use to specify a name.

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.

Related tasks

Specifying prompt control for a CL command parameter
You can control which parameters are displayed for a command during prompting by using prompt control specifications.

Documenting a CL command
If you define and create your own CL commands, you can also create online command help to describe your commands.

Related information

Integrated file system

Creating user-defined CL commands
You can define a CL command by entering command definition statements into a source file and running a **Create Command (CRTCMD)** command using the source file as input.

The *command definition* of each command contains one or more *command definition statements*.

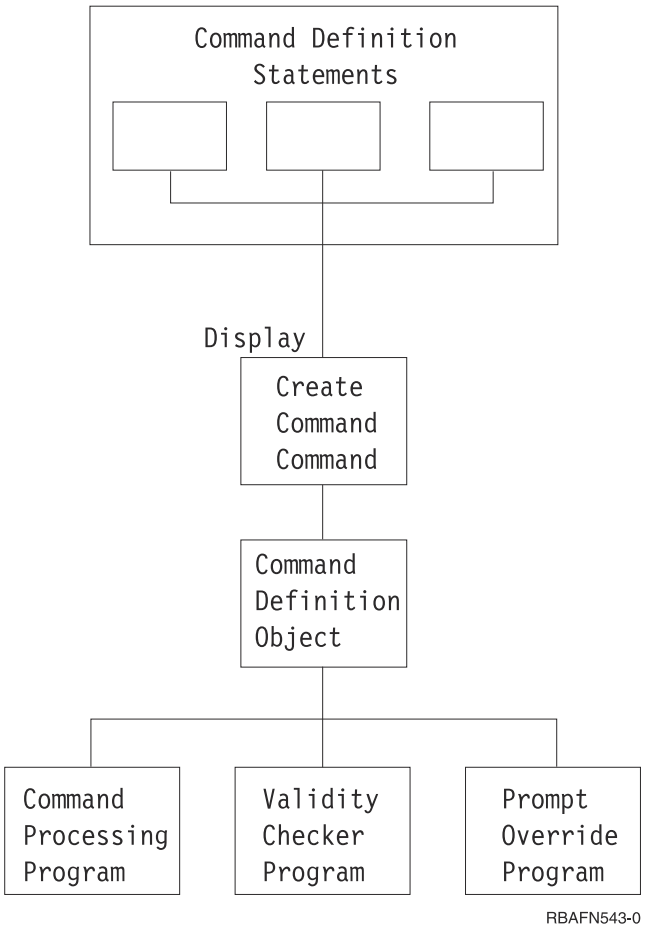
One and only one Command (CMD) statement must be somewhere in the source file. A Parameter (PARM) statement must be provided for each parameter that appears on the command being created. Complex parameters can be defined by using Element (ELEM) statement and Qualifier (QUAL) statement to define the parts of the parameter. If any special keyword relationships need checking, the Dependent (DEP) statement is used to define the relationships. The DEP statement can refer only to parameters that have been previously defined. These statements can appear in any order. Prompt Control (PMTCTL) statements can be used to selectively prompt command parameters.

Only one command can be defined in each source member in the source file. The **Create Command (CRTCMD)** command is run to create the command definition object from the command definition statements in one source file member. Other users can then be authorized to use the new command by the **Grant Object Authority (GRTOBJAUT)** command or the **Edit Object Authority (EDTOBJAUT)** command.

You can enter command definition statements into a database source file as before, or into an IFS stream file. To use IFS stream file, you need to specify source stream file name for the parameter of SRCSTM for **CRTCMD** command.

CL command definition process
The overall process to define your own CL command involves several different phases, some of which are optional.

The following illustration shows the process of creating a command. The text that follows the illustration describes each phase of the process.



Writing your own validity checking and prompt override programs are optional steps.

The command definition statements contain the information that is necessary to prompt the command 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.

Create Command command
The **Create Command (CRTCMD)** command processes the command definition statements to create the command definition object.

The **Create Command (CRTCMD)** command may be run interactively or in a batch job.

Related information
Create Command (CRTCMD) command

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.

The command definition object defines the command, including:

- The command name
- The command processing program (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

Related reference

CL command definition statements
Command definition statements allows system users to create additional CL commands to meet specific application needs.

CL command validity checking
The system performs validity checking on commands. You can 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 high-level language (HLL).
- A CL module or original program model (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* 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 source program 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.

Related concepts

Command-related APIs
Some application programming interface programs can be used with commands.

Related tasks

Validity checking program for a CL command

To detect syntax errors and send diagnostic messages for your command, write a validity checking program.

CL command 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. A prompt override program is optional.

Related tasks
Key parameters and prompt override programs for a CL command
The prompt override program allows current values rather than defaults to be displayed when a command is prompted. Key parameters are parameters, such as the name of an object, that uniquely identify the object.

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 high-level language (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.

CL command exit programs and independent ASPs
Command exit programs needed by a CL command cannot be in a different independent auxiliary storage pool (ASP) than the CL command.

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 defining CL commands
When creating a CL command, you need certain authorities to programs and libraries.

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 as follows.

- 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: Creating a CL command
This example shows how to create a command to allow the system operator to call a program to start the system.

Note: By using the code example, you agree to the terms of the “Code license and disclaimer information” on page 610.

The following example assumes you are using IBM-supplied source files.

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.

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
```

Related information

Create Bound CL Program (CRTBNDCL) command

Defining a CL command

To create a command, you must first define the command through command definition statements.

The general format of the command definition statements and a summary of the coding rules follow.

Statement	Coding rules
CMD	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.

Statement	Coding rules
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. Use a source editor to enter the command definition statements into a database source file member or **an IFS stream file**. Command definition statements can be prompted similar to CL commands.

Related concepts

Dynamic prompt message for control language

By using the message identifiers that are stored in the command (*CMD) object when the command was created, prompt messages can be dynamically retrieved from the message file. This function enables a single command to have prompt messages in more than one national language.

Using the CMD statement

When you define a CL command, you must include 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 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.

Create options in command definition source
Several create option parameters on the **Create Command (CRTCMD)** command can also be specified on the CMD statement. Values that are specified on the CMD statement take precedence over any values specified on the **CRTCMD** command.

Example: Defining Prompt Text and Command Creation Options

```
CMD   PROMPT(UCD0002) PMTFILE(MYCMDPMT *DYNAMIC)+
      MSGF(MYCMDMSG)  TEXT(*CMDPMT)  MAXPOS(2)  +
      PRDLIB(MYAPPLIB)  HLPID(*CMD) +
      HLPPNLGRP(MYAPPHLP)
```

This statement sets the prompt text for the command from message UCD0002 in message file MYCMDPMT, which is located using the library list. All prompt text messages defined for the command are retrieved dynamically when the command is prompted. Error messages sent from the Dependency (DEP) statements in the command definition are found in message file MYCMDMSG, which is located using the library list. The text for the command object is the same as the command prompt text. Only the first two parameters of the command can be specified in positional form without the associated parameter keyword. Library MYAPPLIB is automatically added to the library list before the command is run and is removed from the library list when the command is completed. The help identifier for the command help modules in panel group MYAPPHLP starts with the name of the command object that is created by the **CRTCMD** command.

Defining CL command parameters
To define a CL command parameter, you must use the PARM statement.

You can define as many as 99 parameters for each command.

On the PARM statement, you specify the following:

- Name of the keyword for the parameter
- Whether 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).

Related tasks

Using CL or other HLLs for simple list CL command parameters
When a command is run using CL or another high-level language (HLL), the elements in a simple list are passed to the command processing program in this format.

Using CL or other HLLs for mixed list CL command parameters
When a CL command is run, the elements in a mixed list are passed to the command processing program in this format.

Defining a qualified name CL command parameter
A qualified name is the name of an object preceded by the name of the library in which the object is stored.

Using REXX for a qualified name CL command parameter
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.

Naming the keyword for the CL command parameter
The name of the keyword that you choose for a CL command parameter should be descriptive of the information that is 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.

CL command parameter types
Different types of parameters are used in CL commands.

The basic parameter types are (parameter TYPE value given in parentheses) as follows:

- 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 single quotation marks 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 single quotation marks removed, is left-aligned, 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-aligned 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 single quotation marks 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 single quotation marks removed, is left-aligned, 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 single quotation marks, 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, a decimal point, or all of them; 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-aligned 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 the IBM i operating system, you must enclose the name in single quotation marks.
- 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 CL command parameter value
The length (LEN) parameter is used to specify the length of a parameter.

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
*SNAME	256	10
*CNAME	256	10
*GENERIC	256	10
*HEX	256	1
*X	(256 24 9)	(1 15 5)
*VARNAME	11	11
*CMDSTR	20000	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 IBM i CL command prompter displays a maximum input field of 512 characters, even if the parameter length is longer than 512 characters.

Related tasks

Using the Call Program command to pass control to a called program
When the **Call Program (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.

Default CL command parameter values
If you are defining an optional parameter, you can specify 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: Defining a CL command parameter
This example shows how to define a parameter used in calling an application through a CL command.

The following example defines a parameter OETYPE for a CL 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.

Related tasks

Creating a CL command

After you have defined your command through the command definition statements, use the **Create Command (CRTCMD)** command to create the command.

Data type and parameter restrictions

These tables show 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

	MIN	MAX	ALWUNPRT	ALWVAR	PGM	DTAARA	FILE	FULL	EXPR	VARY
*NAME	X	X		X	X	X	X	X	X	1
*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					

	PASSATR	PASSVAL	CASE	LISTDSPL	DSPINPUT
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
*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 table 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 that describes the restriction at the bottom of the table.

	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		

	LEN	RTNVAL	CONSTANT	RSTD	DFT	VALUES	REL	RANGE	SPCVAL	SNGVAL
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								
PASSATR		3								
PASSVAL		13						11		
CASE										
LISTDSPL										
CHOICE			14							
CHOICEPGM										
PMTCTL			15							
PMTCTLPGM			15							
PROMPT			6							
INLPMTLEN		17	17	17						

	LEN	RTNVAL	CONSTANT	RSTD	DFT	VALUES	REL	RANGE	SPCVAL	SNGVAL

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										
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										

	MIN	MAX	ALWUNPRT	ALWVAR	PGM	DTAARA	FILE	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					

	PASSATR	PASSVAL	CASE	LISTDSPL	DSPINPUT
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						
FULL						12

	CHOICE	CHOICEPGM	PMTCTL	PMTCTLPGM	PROMPT	INLPMTLEN
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 a CL command list parameter

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 command parameter
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.

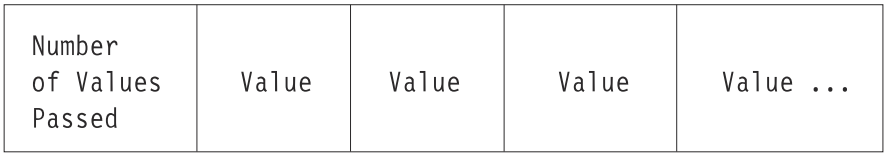
The following example defines a parameter USER for which the display station user can specify up to five user names (a simple list).

```
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 another high-level language (HLL), or REXX.

Using CL or other HLLs for simple list CL command parameters
When a command is run using CL or another high-level language (HLL), the elements in a simple list are passed to the command processing program in this 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. 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.

The following example shows 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 2. Simple list example

The 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

Related tasks

Defining CL command parameters

To define a CL command parameter, you must use the PARM statement.

Using REXX for simple list CL command parameters
When the same command is run, the elements in a simple list are passed to the REXX procedure in the argument string in the format where `valueN` is the last value in the simple list.

```
. . . USER(value1 value2
. . . valueN) . . .
```

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 following REXX example produces the same result as the CL procedure in "Example: Creating a CL command" on page 304:

```
        .
        .
        .
        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 3. 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() . . .
```

Related information

REXX/400 Programmer's Guide PDF

REXX/400 Reference PDF

Defining a mixed list CL command parameter

A mixed list CL command parameter accepts a set of separately-defined values.

These values typically 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.

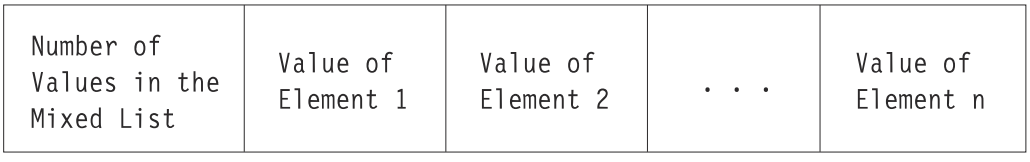
In the following example, a parameter CMPVAL is defined for which you can specify a comparison value and a starting position for the comparison (a mixed list).

```
      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 another high-level language) or REXX.

Using CL or other HLLs for mixed list CL command parameters

When a CL command is run, the elements in a mixed list are passed to the command processing program in this format.



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. 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 program or procedure, you must use the substring built-in function.

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.

0000

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 is to be processed by a CL program or procedure, the parameter value can be received into a 14-character CL variable. The first two characters can be compared to either of the following:

- a 2-character variable initialized to hexadecimal 0000 using the %SUBSTRING function.
- a decimal 0 using the %BINARY built-in function.

Related concepts

CL programming

To program using CL, you must understand the procedures and concepts specific to CL programming.

Related tasks

Defining CL command parameters

To define a CL command parameter, you must use the PARM statement.

Using REXX for mixed list CL command parameters

When a CL command is run using REXX, elements in a mixed list are passed to the command processing program in the format where `valueN` is the last value in the mixed list.

```
. . . CMPVAL(value1 value2
. . . valueN) . . .
```

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 for a CL command parameter
A list within a list can be either a list that can be specified more than once for a CL command parameter (simple or mixed list), or 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.

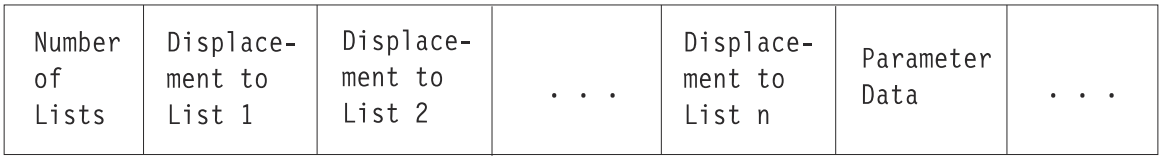
```
        PARM KWD(PARM2) TYPE(LIST2)
LIST2:  ELEM TYPE(*CHAR) LEN(10) MAX(5)
        ELEM TYPE(*DEC) LEN(3 0)
```

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 another high-level language) or REXX.

Using CL or other HLLs for lists within lists
When a CL command is run, a command parameter that is a list within a list is passed to the command processing program in this 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.

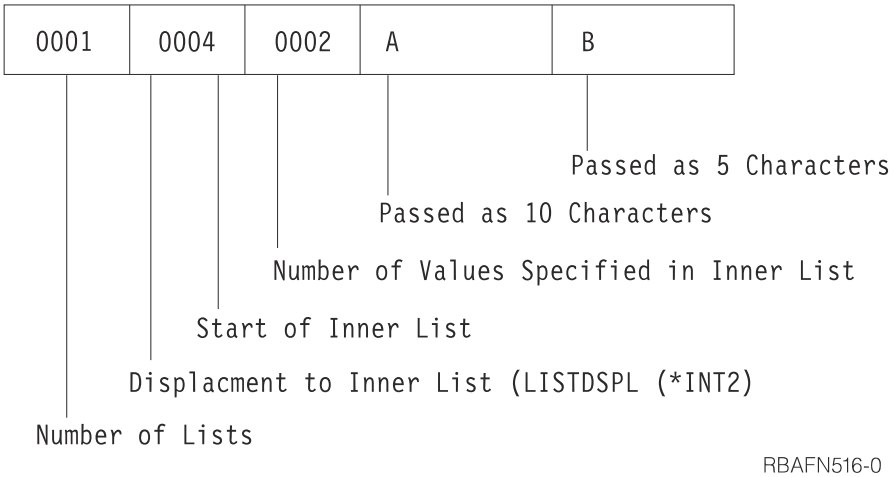
The following example shows a definition for a parameter KWD2 (which is a mixed list within a simple list) how the parameter can be specified by the display station user, and what is passed. The parameter definition is:

```
       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.

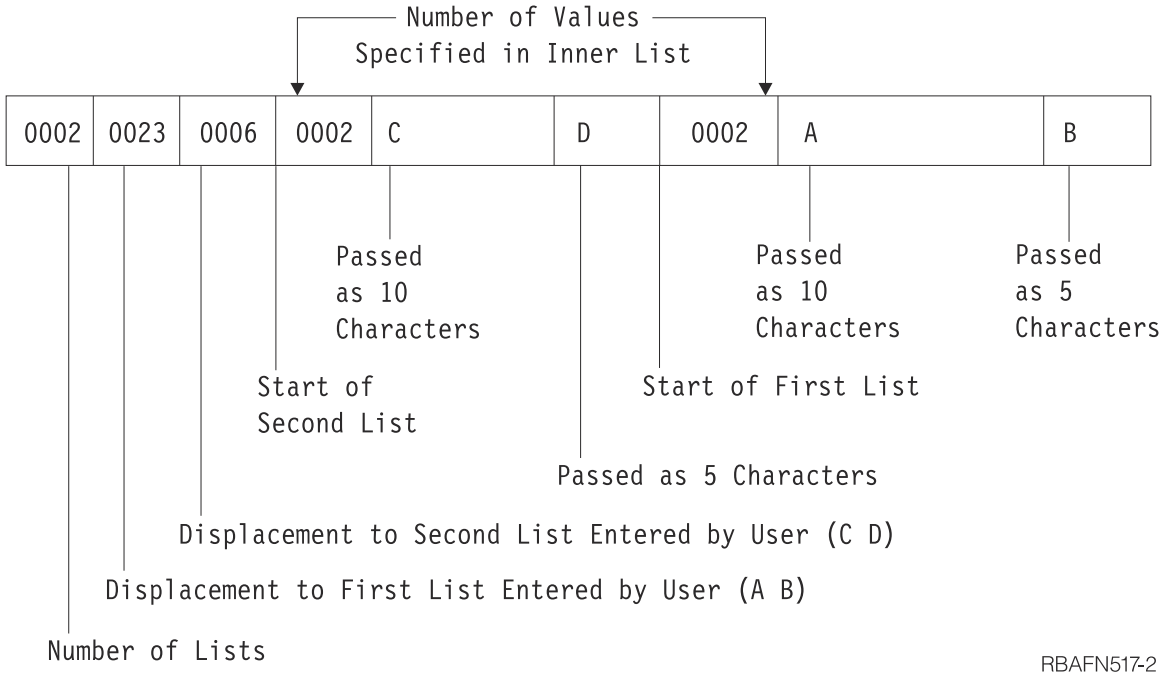


RBAFN516-0

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.

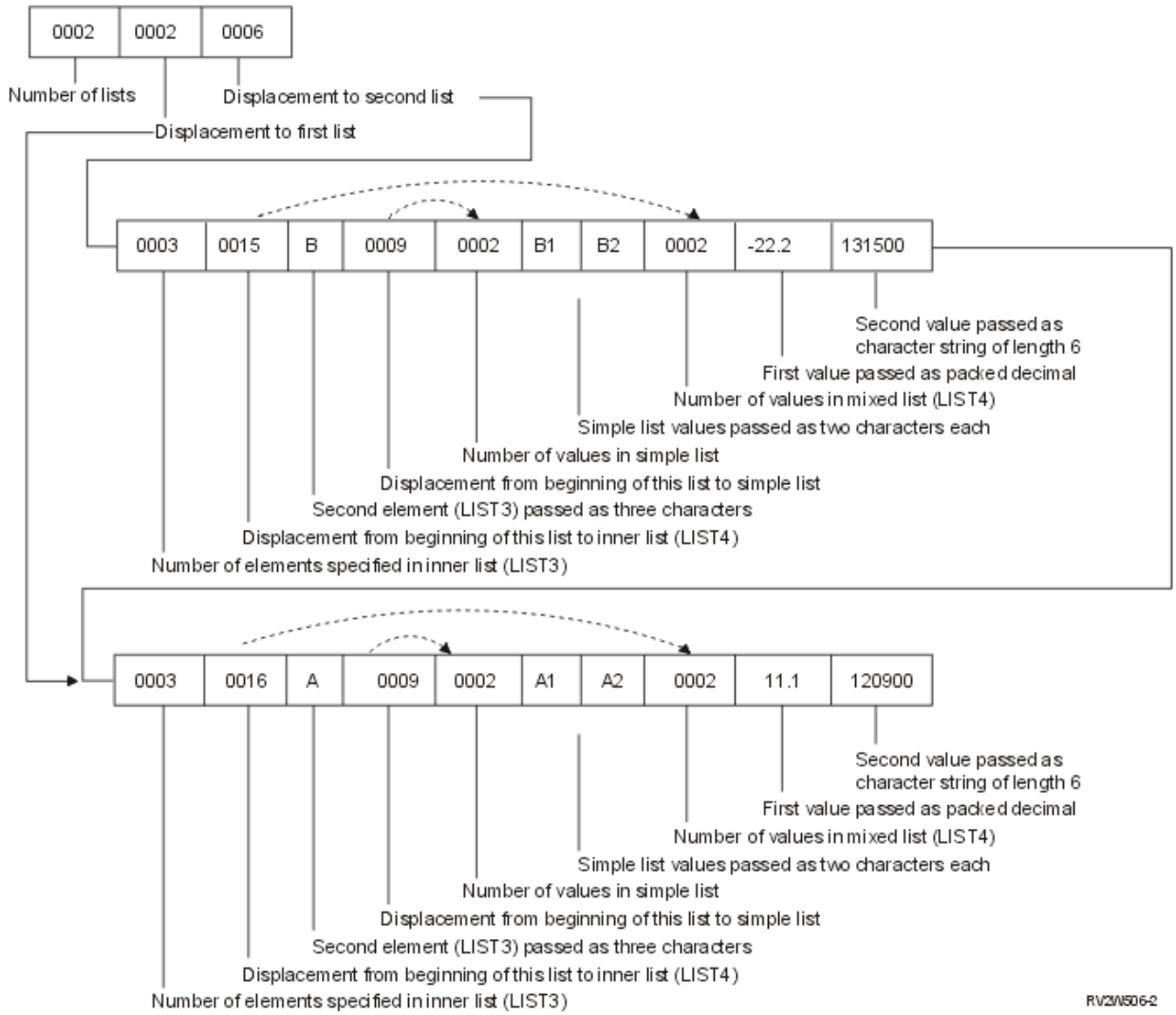
The following is a more complex example of lists within lists. The parameter definition is:

```
        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 shown in the following example:

```
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 CL command is run using REXX, a command parameter that is 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)))
```

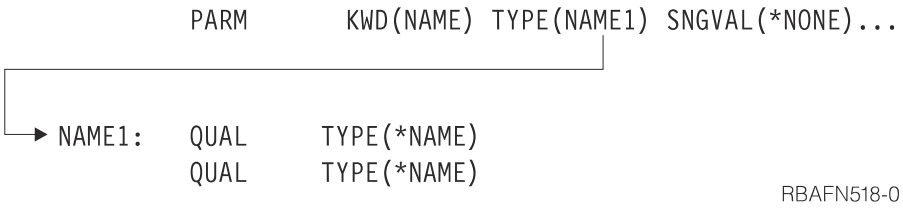
The following is passed to the command processing program:

```
PARM1(((11.1  12D900)  A  (A1  A2))  ((-22.2  131500)  B  (B1  B2)))
```

Defining a qualified name CL command parameter
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 that 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. 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 another high-level language) or REXX.

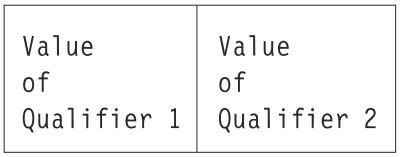
Related tasks

Defining CL command parameters

To define a CL command parameter, you must use the PARM statement.

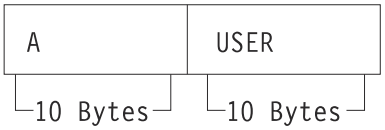
Using CL or other HLLs for a qualified name CL command parameter

A qualified name is passed to the command processing program in this format when you use CL or another high-level language (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 CL command parameters" on page 307). 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

The CL variables &OBJ and &LIB are defined over the two parts of the qualified name parameter passed to the CL command processing program.

```
PGM PARM(&QLFDNAM)
DCL &QLFDNAM TYPE(*CHAR) LEN(20)
DCL &OBJ TYPE(*CHAR) LEN(10) STG(*DEFINED) DEFVAR(&QLFDNAM 1)
DCL &LIB TYPE(*CHAR) LEN(10) STG(*DEFINED) DEFVAR(&QLFDNAM 11)
  .
  .
  .
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.

Number of Qualified Names	Value 1 Qualifier 1	Value 1 Qualifier 2	Value 2 Qualifier 1	Value 2 Qualifier 2	. . .

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)
```

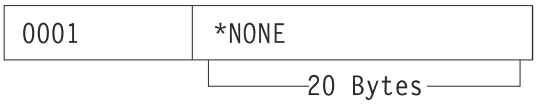
If the display station user enters the following:

```
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 CL command parameter
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.

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)
```

Related tasks

Defining CL command parameters

To define a CL command parameter, you must use the PARM statement.

Defining a dependent relationship between CL command parameters

A dependent relationship is a required relationship that must exist between parameters of a CL command.

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 as follows:

- 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 for a CL command parameter

To determine the choices and values that can be defined for a CL command parameter, refer to this information.

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 10240-byte field into which a permissible values list is to be returned.

- **Parameter 2**: A 30- or 10240-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 10240-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.

Specifying prompt control for a CL command parameter

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.

Related concepts

CL command delimiter characters

Command delimiter characters are special characters or spaces that identify the beginning or end of a group of characters in a command.

CL command definition
Command definition allows you to create additional control language commands to meet specific application needs. These commands are similar to the system commands.

Related reference

CL command definition statements
Command definition statements allows system users to create additional CL commands to meet specific application needs.

Specifying conditional prompting for a CL command parameter
You can specify that a parameter is prompted for only when certain conditions are met.

When prompting the user for a command, a parameter that 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 preceded by a length value. If the controlling parameter is PASSATR(*YES), the attribute byte is 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.

Note: By using the code example, you agree to the terms of the "Code license and disclaimer information" on page 610.

```
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
```

Hiding additional or advanced parameters when prompting a CL command
If you want a parameter that is infrequently used not to be prompted for unless the user requests additional parameters by pressing a function key during prompting, specify 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.

Related tasks

Key parameters and prompt override programs for a CL command
The prompt override program allows current values rather than defaults to be displayed when a command is prompted. Key parameters are parameters, such as the name of an object, that uniquely identify the object.

Key parameters and prompt override programs for a CL command

The prompt override program allows current values rather than defaults to be displayed when a command is prompted. Key parameters are parameters, such as the name of an object, that uniquely identify the object.

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.

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.

To exit the command prompt, press F3=Exit.

Related concepts

CL command prompt override program
You can write prompt override programs to supply current values for parameter defaults when prompting the command.

Hiding additional or advanced parameters when prompting a CL command
If you want a parameter that is infrequently used not to be prompted for unless the user requests additional parameters by pressing a function key during prompting, specify PMTCTL(*PMTRQS) on the PARM statement for the parameter.

Using a prompt override program for a CL command
A prompt override program can be used to allow current values rather than defaults to be displayed when a command is prompted.

To use a prompt override program, follow these steps:

1. Specify any parameters that are to be key parameters on the PARM statement in the command definition source.
2. Write a prompt override program.
3. Specify the name of the prompt override program on the PMTOVRPGM parameter when you create or change the command.

Identifying key parameters for a CL command
The number of key parameters should be limited to the number of parameters that is needed to uniquely define the object that is to be changed.

To ensure that a key parameter is coded correctly in the command definition source, consider these requirements:

- 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.

Prompt override program for a CL command
A prompt override program needs to have certain information passed 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.

Related concepts
Example: Using a prompt override program
This example shows the command source for a command and the prompt override program.

Parameters passed to the prompt override program
The prompt override program is passed several 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.

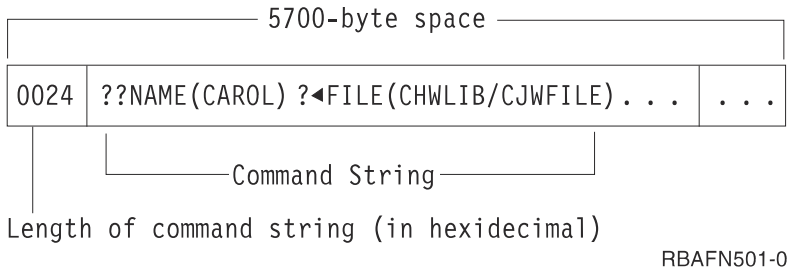
  When using selective prompts, do the following:

  – 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.



RBAFN501-0

- 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.

  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 single quotation marks, 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.

Related concepts

Example: Using a prompt override program
This example shows the command source for a command and the prompt override program.

Related tasks

Using selective prompting for CL commands
Selective prompting for CL commands is especially helpful when you are using some of the longer commands and do not want to be prompted for certain parameters.

Related information

Change Debug (CHGDBG) command

Add Program (ADDPGM) command

Allowing for errors in a prompt override program
Your prompt override program should include error handling.

If the prompt override program detects an error, it should follow these steps to handle the error:

1. Set the command string length to zero so that the defaults rather than current values are displayed when the command is prompted.
2. Send a diagnostic message to the previous program on the call stack.
3. 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' vv &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.

Related information

Send Program Message (SNDPGMMSG) command

Specifying a prompt override program when creating or changing a CL command
To use a prompt override program for a command that 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 the **CRTCMD** and **CHGCMD** 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 no key parameters are defined in the command definition source. In this case, the prompt override program is called before the command is prompted; informational message CPD029A is sent when the command is created or changed.

Related information
Create Command (CRTCMD) command
Change Command (CHGCMD) command

Example: Using a prompt override program
This example shows the command source for a command and the prompt override program.

The following 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.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 610.

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')
```

Prompt override program

The following prompt override program uses the "?^" selective prompt characters.

```
 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:
/********************************************************************/
/*                                                                  */
/* Set the command string length to zero                            */
/*                                                                  */
/********************************************************************/
```

```
    CHGVAR VAR(%BIN(&rtnstring 1 2)) VALUE(0)
```

```
/*****************************************************************/
/*                                                               */
/* Send error message(s)                                         */
/*                                                               */
/* NOTE: If you want 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 want 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
```

Related concepts

Prompt override program for a CL command
A prompt override program needs to have certain information passed to return current values when a command is prompted.

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.

Creating a CL command

After you have defined your command through the command definition statements, 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.
- 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: Defining a CL command parameter" on page 311.

```
CRTCMD     CMD(DSTPRODLB/ORDENTRY) +
           PGM(*LIBL/ORDENT) +
           TEXT('Calls order entry application')
```

The resulting command is as follows where the value can be DAILY, WEEKLY, or MONTHLY:

```
ORDENTRY  OETYPE(value)
```

After you have created a command, you can perform the following tasks:

- 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

Related concepts

Command-related APIs
Some application programming interface programs can be used with commands.

Related tasks

Example: Defining a CL command parameter
This example shows how to define a parameter used in calling an application through a CL command.

Related information

Create Command (CRTCMD) command
CL command finder

CL command definition source listing
When you create a CL command, a source listing is produced.

Here is a sample source list. The numbers refer to descriptions following the list.

```
5770SS1 V7R1M0  1004231                 Command Definition        DSTPRODLB/ORDENTRY   SYSNAME  11/20/12 14:53:322Page 13

Command name  . . . . . . . . . . . . . . . . . :   ORDENTRY
  Library . . . . . . . . . . . . . . . . . . . :     DSTPRODLB
Command processing program  . . . . . . . . . . :   ORDENT     4
  Library . . . . . . . . . . . . . . . . . . . :     *LIBL
Source file . . . . . . . . . . . . . . . . . . :   QCMDSRC
  Library . . . . . . . . . . . . . . . . . . . :     QGPL
Source file member  . . . . . . . . . . . . . . :   ORDENTRY     11/20/12  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 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')                                            11/20/12
7
   200-               PARM       KWD(OETYPE) TYPE(*CHAR) RSTD(*YES) +                                     11/20/12
   300                             VALUES(DAILY WEEKLY MONTHLY) MIN(1) +                                  11/20/12
   400                             PROMPT('Type of order entry:')                                         11/20/12
                                        * * * * *  E N D  O F  S O U R C E  * * * * *

5770SS1 V7R1M0  100423                Command Definition               DSTPRODLB/ORDENTRY     11/20/12 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  * * * * *

5770SS1 V7R1M0  100423                Command Definition               DSTPRODLB/ORDENTRY     11/20/12 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 the IBM i operating system.

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.

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.

Related concepts

Common errors when processing CL 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.

Common errors when processing CL 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 is encountered that stops the command from being created, 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, use the EDTF (Edit File) command, or you can use the Source Entry Utility (SEU), which is part of the WebSphere Development Studio.

In the command definition source list, an error condition that relates directly to a specific source statement is listed after that command. 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.

Related reference

CL command definition source listing
When you create a CL command, a source listing is produced.

Displaying a CL command definition

To display or print the values that are specified as parameters on the **Create Command (CRTCMD)** command, use the **Display Command (DSPCMD)** 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 **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.

Related information

Create Command (CRTCMD) command
Change Command (CHGCMD) command
Display Command (DSPCMD) command
Retrieve Command Information (QCDRCMDI) API
Retrieve Command Definition (QCDRCMDD) API

Effect of changing the command definition of a CL command in a procedure or program

Some changes can be made to the command definition of a command with no further action. Other changes might require the program or procedure to be re-created, or might cause the program or procedure to function differently.

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 recreating 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 CL command defaults
To change the default value of a command parameter, use the **Change Command Default (CHGCMDDFT)** command.

The command parameter must have an existing default to 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 recommendations pertain to 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 **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.

   Note: By using the code example, you agree to the terms of the "Code license and disclaimer information" on page 610.

```
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.

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 following example:

```
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 following example:

```
CHGCMDDFT USRQSYS/CRTCLPGM 'AUT(*EXCLUDE) +
TEXT('Isn''t this nice text')'
```

10. Because there are embedded single quotation marks 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.

Related information

CL command finder

Application Programming Interfaces

Examples: Changing CL command defaults

These examples show how to change default values for CL commands.

- To provide a default value of *NOMAX for the MAXMBRS keyword of command **Create Physical File (CRTPF)**, do the following:

```
CRTPF FILE(FILE1) RCDLEN(96) MAXMBRS(1)
  .
  .
CHGCMDDFT CMD(CRTPF) NEWDFT('MAXMBRS(*NOMAX)')
```

- To provide a default value of 10 for the MAXMBRS keyword of the command **Create Physical File (CRTPF)**, do the following:

```
CRTPF FILE(FILE1) RCDLEN(96) MAXMBRS(*NOMAX)
  .
  .
CHGCMDDFT CMD(CRTPF) NEWDFT('MAXMBRS(10)')
```

- 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 **Create CL Program (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)')
```

- The following provides a default value of 'Isn't this print text' for the PRTTXT keyword of the command **Change Job (CHGJOB)**. Because the NEWDFT keyword has embedded single quotation marks, you must not double these single quotation marks, or the process will not run correctly.

```
CHGJOB PRTTXT('Isn''t this print text')
   .
   .
CHGCMDDFT CMD(CHGJOB) +
          NEWDFT('PRTTXT(''Isn''''t this print text'')')
```

- 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 **Create Logical File (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.

- To create a command that will display the spool files for a job, do the following:

```
CRTDUPOBJ OBJ(WRKJOB) FROMLIB(QSYS) +
          TOLIB(MYLIB) NEWOBJ(WRKJOBSPLF)
WRKJOBSPLF  OPTION(*SPLF)
```

```
    .
    .
CHGCMDDFT CMD(MYLIB/WRKJOBSPLF) +
          NEWDFT('OPTION(*SPLF)')
```

Command processing program for a CL command

A command processing program (CPP) is a program that processes a command. This program performs some validity checking and processes the command so that the requested function is performed.

A command processing program (CPP) can be a CL or high-level language (HLL) program, or a REXX procedure. Programs written in CL or HLL can also be called directly with the **Call (CALL)** 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.

Related information

Create Command (CRTCMD) command
Call Program (CALL) command
Start REXX Procedure (STRREXPRC) command

CL or HLL command processing program
This information describes the command processing program written in CL or another high-level language (HLL).

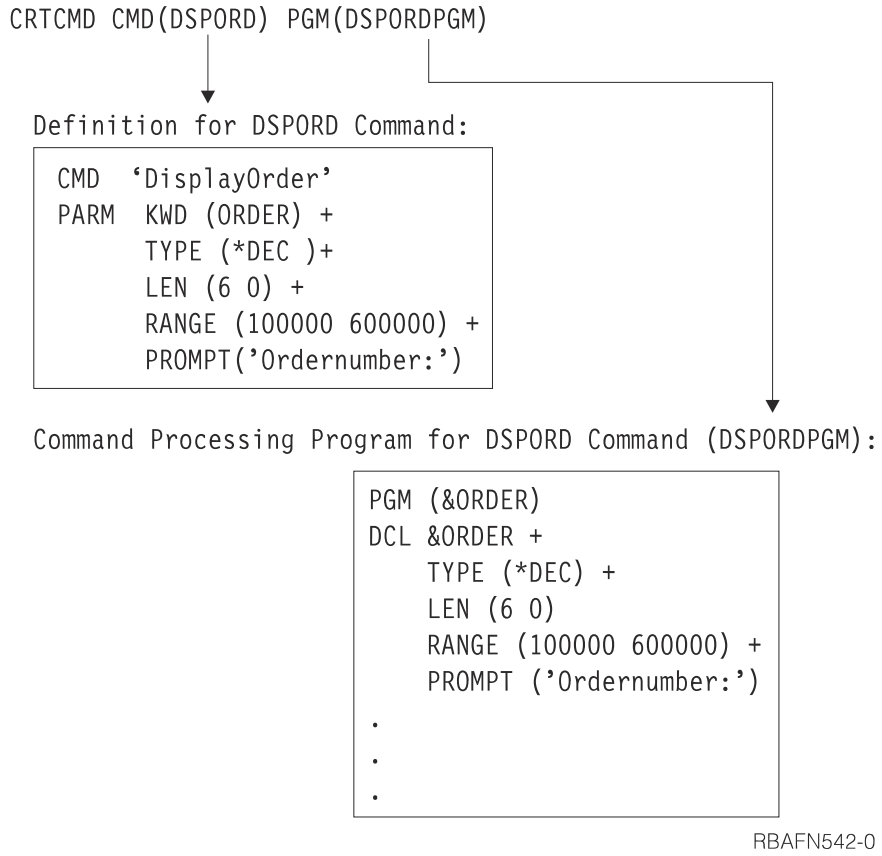


Figure 4. Command relationships for CL and HLL

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 table shows this correspondence.

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

PARM statement type	PARM statement length	Declared variable type	Declared variable length
*UINT2	n	*UINT or *CHAR	2
*UINT4	n	*UINT or *CHAR	4

Notes:

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.

Related tasks

Examples: Defining and creating CL commands
These examples show how to define and create CL commands.

Related information

Create Command (CRTCMD) command

REXX command processing procedure for a CL command
This information describes a REXX command processing procedure for a CL command.

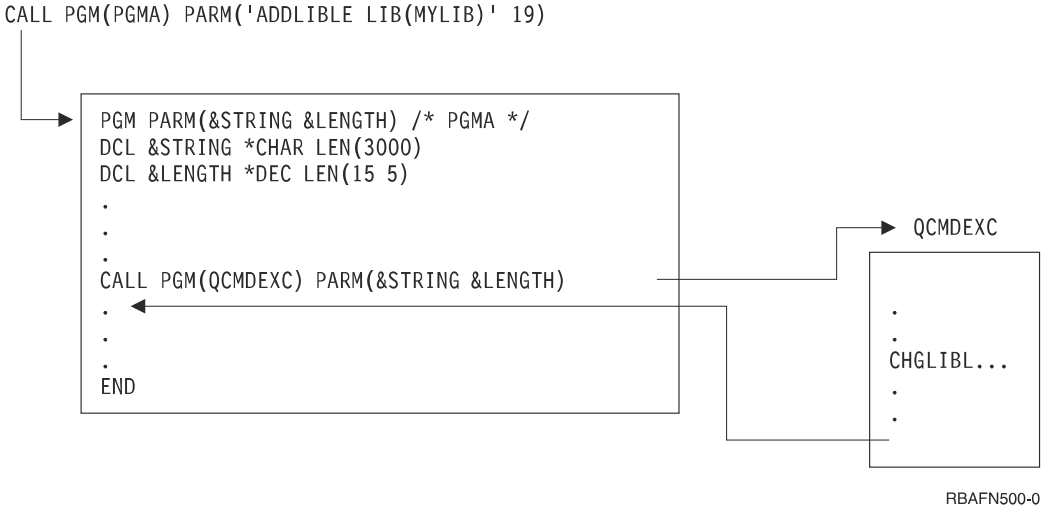


Figure 5. Command relationships for REXX

Related information

Create Command (CRTCMD) command

Validity checking program for a CL command

To detect syntax errors and send diagnostic messages for your command, write a validity checking program.

To 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 **Create Command (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
```

Related concepts

CL command validity checking
The system performs validity checking on commands. You can also write your own validity checking program although it is not required.

Related information

Send Program Message (SNDPGMMSG) command

Create Command (CRTCMD) command

Examples: Defining and creating CL commands

These examples show how to define and create CL commands.

Related tasks

CL or HLL command processing program
This information describes the command processing program written in CL or another high-level language (HLL).

Example: Calling application programs
This example shows how to define and create a CL command to call an application program.

You can create commands to call application programs. If you create a command to call an application program, the IBM i operating system performs validity checking on the parameters passed to the program. However, if you use the **Call (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 program, the following specifications would be made in the program to receive the following parameters:

```
*ENTRY   PLIST
         PARM    CUST   5
         PARM    COUNT  30
         PARM    FORM   3
```

The **Create Command (CRTCMD)** command is as follows:

```
CRTCMD CMD(LBLWRT) PGM(LBLWRT) SRCMBR(LBLWRT)
```

Related information

Create Command (CRTCMD) command

Example: Substituting a default value

This example shows how to define and create a CL command that provides defaults for an IBM-supplied CL command.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 610.

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 to 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 **Save Library to Tape (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 **Create Command (CRTCMD)** command is:

```
CRTCMD CMD(SAVLIBTAP) PGM(SAVLIBTAP) SRCMBR(SAVLIBTAP)
```

Related information

Create Command (CRTCMD) command

Example: Displaying an output queue

This example shows how to define and create a CL command that displays an output queue.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 610.

You can create a command to display an output queue that defaults to display the output queue PGMR. The following command, **Display Output Queue (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 **Display Output Queue (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 **Display Output Queue (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 **Create Command (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, the IBM i operating system 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.

Related information

Create Command (CRTCMD) command

Example: Displaying messages from IBM-supplied commands
This example shows how to define and create a CL command that re-displays messages.

The **Clear Output Queue (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 **Clear Output Queue (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 **Clear Output Queue (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 **Create Command (CRTCMD)** command is:

```
CRTCMD CMD(CQ2) PGM(CQ2)
```

The CPP, which receives the completion message and displays it, is as follows.

Note: By using the code example, you agree to the terms of the "Code license and disclaimer information" on page 610.

```
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.

Related information
Create Command (CRTCMD) command

Example: Creating abbreviated change job CL command
This example shows how to define and create an abbreviated command.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 610.

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 **Change Job (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: **Create Command (CRTCMD)** command

```
    CRTCMD  CMD(CJ)  PGM(CJ)  SRCMBR(CJ)
```

Related information

Create Command (CRTCMD) command

Example: Creating abbreviated printer writer CL command

This example shows how to define and create an abbreviated CL command.

Note: By using the code example, you agree to the terms of the "Code license and disclaimer information" on page 610.

You can 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 **Create Command (CRTCMD)** command is:

```
CRTCMD CMD(DW1) PGM(DW1) SRCMBR(DW1)
```

Related information

Create Command (CRTCMD) command

Example: CL command for deleting files and source members

This example shows how to define and create a CL command that deletes files and source members.

You can create a CL command to delete files and their corresponding source members in QDDSSRC.

The command definition statements for the command named **Delete File and Source (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.

Here is the command processing program.

Note: By using the code example, you agree to the terms of the "Code license and disclaimer information" on page 610.

```
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
```

Example: CL command for deleting program objects
This example shows how to define and create a CL command that deletes program objects and source members.

You can create a command to delete high-level language (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. Here is the command processing program.

Note: By using the code example, you agree to the terms of the "Code license and disclaimer information" on page 610.

```
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
```

Documenting a CL command

If you define and create your own CL commands, you can also create online command help to describe your commands.

Related concepts

CL command information and documentation
IBM provides documentation for CL commands. In addition, you can write documentation for your own commands.

Related tasks

Defining CL commands
CL commands enable you to request a broad range of functions. You can use IBM-supplied commands, change the default values for command parameters, and define your own commands.

Related reference

CL command definition statements
Command definition statements allows system users to create additional CL commands to meet specific application needs.

CL commands and command online 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; the same capabilities for help provided for IBM-provided commands are also available for you to use to create online help.

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 **Create Command (CRTCMD)** 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), the IBM i operating system 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 **Change Command (CHGCMD)** command to have a help panel group associated with the command, the help stored in the panel group is retrieved, formatted, and displayed. The operating system 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), the operating system 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, the operating system 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 slash character followed by COMMAND/EXAMPLES. For example, if command STRPAY has a HLPID value of STRPAY, the operating system 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, the operating system 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.

Related information

Create Command (CRTCMD) command

Writing CL command help

After you understand how a CL command (*CMD) object can be connected to an online help panel group (*PNLGRP) object, you can write the help text that goes into the four types of help modules for a command.

Online help for CL commands is written in a tag language known as user interface manager (UIM). The UIM source is compiled using the **Create Panel Group (CRTPNLGRP)** command to create a *PNLGRP object.

Related information

Create Panel Group (CRTPNLGRP) command

Application Display Programming

Generating UIM source for CL command help

As a simpler alternative to learning UIM syntax, you can use the **Generate Command Documentation (GENCMDDOC)** command to generate online help for CL commands. With this command, you can create a file that contains 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 **Generate Command Documentation (GENCMDDOC)** to create the UIM source can simplify writing online help for your CL commands.

The following is an example of using the **Generate Command Documentation (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 embed 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.

After you have edited the UIM source to tailor it to your command, you can create the online help panel group by using the **Create Panel Group (CRTPNLGRP)** command. The following example shows how to use the **Create Panel Group (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.

Related information

Create Panel Group (CRTPNLGRP) command

Generate Command Documentation (GENCMDDOC) command

Common help sharing between CL commands

Because of the naming scheme used for connecting command online help modules with a command, you can 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 the :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 embedded dynamically using :IMHELP tags.

Related information

Application Display Programming

Organizing CL online help text into help modules

You can choose how to organize the CL command help text that goes into the help modules in your panel group.

Using the UIM capability to embed 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 embedded 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 CL command documentation

To generate Hyper Text Markup Language (HTML) information to be viewed using a browser from your CL command online help, use the **Generate Command Documentation (GENCMDDOC)** command and specify *HTML for the GENOPT (Generation options) parameter.

In addition to creating help information for your command, you might want to create command documentation that can be viewed either when you are not connected to a system running the IBM i operating system or when the information is printed. The operating system allows you to generate command documentation in 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 that contains information that is retrieved from the command object (*CMD) that you specify and any command help panel group (*PNLGRP) objects that already exist for the command.

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 IBM i Information Center. IBM builds the Information Center command documentation from the online command help.

The following is an example of using the **Generate Command Documentation (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 **Display File (DSPF)** command, or in HTML browser format using any standard internet browser software, or by command.

Related information

Generate Command Documentation (GENCMDDOC) command

Proxy CL commands

Proxy commands are used to create shortcuts to target CL commands.

A proxy command differs from a typical IBM i command in its ability to run a target command, rather than a command processing program.

Use the **Create Proxy Command (CRTPRXCMD)** to create a proxy command. **Create Proxy Command (CRTPRXCMD)** requires a Command name (CMD) and a Target command (TGTCMD). Optional parameters include a Text description (TEXT), Authority (AUT) and a Replace command (REPLACE) option. For example, to create a proxy command that has a target command of QSYS/DSPJOB, enter the following command:

```
CRTPRXCMD CMD(MYLIB/PCMD1) TGTCMD(QSYS/DSPJOB) TEXT('dspjob proxy')
```

Specifying a proxy command for the TGTCMD parameter is allowed. A *proxy chain* can be up to five proxy commands in length, ending with a sixth non-proxy command. Running a proxy command with a chain of more than five proxy commands causes a CPD0196 error "Number of chained proxy commands exceeds 5."

When you create a proxy command, you do not specify command definition source. A proxy command does not contain any parameter information, prompt control information, or inter-parameter dependency information. When a proxy command is used, all that information is inherited from the target command defined in the proxy command. You specify all the same parameters for the proxy command that you would for the target command.

Use the **Change Proxy Command (CHGPRXCMD)** command, to change a proxy command. Use the **Delete Command (DLTCMD)** command to delete a proxy command.

To run or process a command that is resolved to through a proxy chain, the user must have the correct authority for that command and library. The user must also have *USE authority for each of the proxy commands and libraries in the proxy command chain.

Since proxy commands run other commands, attempting to register an exit program for a proxy command is not allowed. Attempting to register an exit program for a proxy command causes error message CPF019A to be issued. When a proxy command is run and the non-proxy command that it resolves to has a registered exit program, the proxy chain that was followed is placed in the RTVC0100 or CHGC0100 returned format data.

Related information

Create Proxy Command (CRTPRXCMD) command
Change Proxy Command (CHGPRXCMD) command
Delete Command (DLTCMD) command

Command-related APIs

Some application programming interface programs can be used with commands.

Related concepts

CL command validity checking
The system performs validity checking on commands. You can also write your own validity checking program although it is not required.

Related tasks

Creating a CL command
After you have defined your command through the command definition statements, use the **Create Command (CRTCMD)** command to create the command.

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 before 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.

Related concepts

Variables in CL commands

A *variable* is a named changeable value that can be accessed or changed by referring to its name.

Related information

Process Commands (QCAPCMD) API

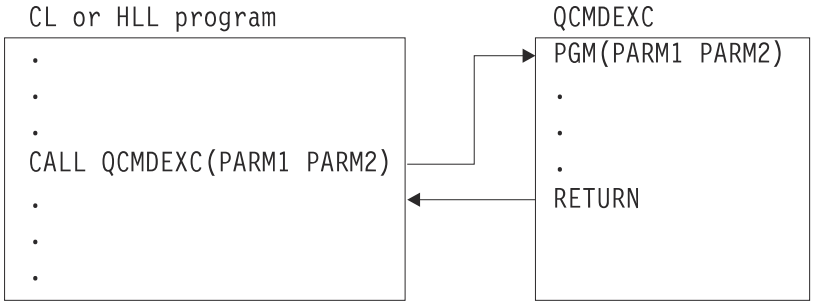
QCMDEXC program

The Execute Command (QCMDEXC) API is an IBM-supplied program that runs a single command.

This API 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.



RBAFN558-0

After the command runs, control returns to your HLL or CL procedure or program.

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 want 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 single quotation marks if it contains blanks. The maximum length of the character string is 32,702 characters; never count the delimiters (the single quotation marks) 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 runtime flexibility when used this way.

Providing runtime 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, examine the following figure.

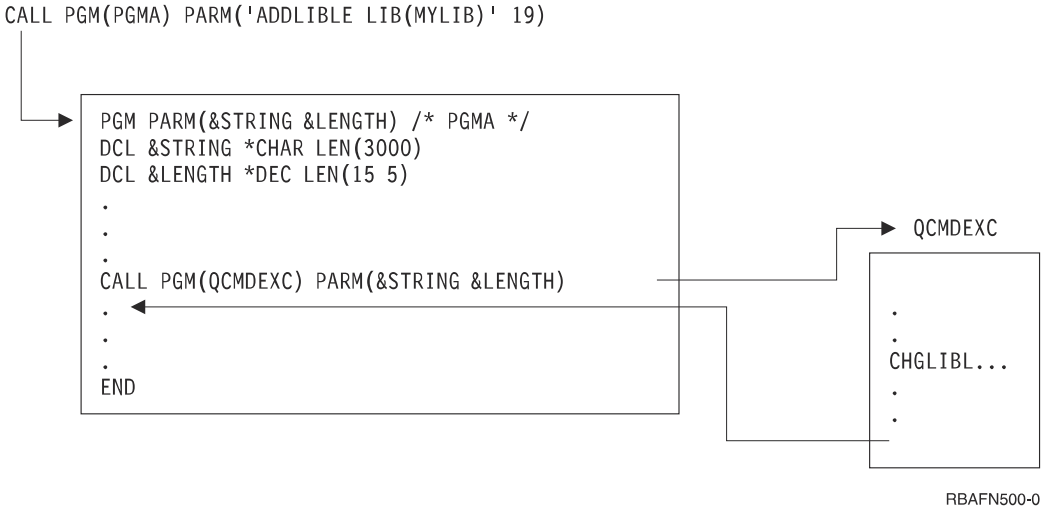


Figure 6. Example of call PGM

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

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.

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 language reference book for information about how high-level language programs handle errors on calls.

Related concepts

Variables in CL commands

A *variable* is a named changeable value that can be accessed or changed by referring to its name.

Related tasks

Defining message descriptions

Predefined messages are stored in a message file.

Using selective prompting for CL commands
Selective prompting for CL commands is especially helpful when you are using some of the longer commands and do not want to be prompted for certain parameters.

Related information
Execute Commands (QCMDEXC) API

QCMDEXC program with DBCS data

You can use the Execute Command (QCMDEXC) API 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 **Send Message (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
```

Related information
Execute Commands (QCMDEXC) API

QCMDCHK program

The Check Command Syntax (QCMDCHK) API 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 **Create CL Program (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 **Change Command Definition (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.

Related tasks

Using selective prompting for CL commands
Selective prompting for CL commands is especially helpful when you are using some of the longer commands and do not want to be prompted for certain parameters.

Related information

Check Command Syntax (QCMDCHK) API

Prompting for user input at run time

With most CL programs and procedures, the workstation user provides input by specifying command parameter values that are passed to the program or by typing into input-capable fields on a display prompt.

You can 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 source program, the system displays a prompt for the CL command. Parameter values you have already specified in your source program are filled in and cannot be changed by the workstation user.
- 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 source program which CL command is to be used at processing time.

Related concepts

Variables in CL commands

A *variable* is a named changeable value that can be accessed or changed by referring to its name.

Using the IBM i 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 **Display Library (DSPLIB)** command waits until you have entered values for required parameters and pressed the Enter key.

Any values specified in the source program 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. The following restrictions apply to the use of the prompter within a CL program or procedure at runtime:

- When the prompter is called from a CL 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 at run time.

CALL	**CALLPRC**	**CALLSUBR**	**CHGVAR**
COPYRIGHT	**DCL**	**DCLF**	**DCLR**
DO	**DOFOR**	**DOUNTIL**	**DOWHILE**
ELSE	**ENDDO**	**ENDPGM**	**ENDRCV**
ENDSELECT	**ENDSUBR**	**GOTO**	**IF**
ITERATE	**LEAVE**	**MONMSG**	**OTHERWISE**
PGM	**RCVF**	**RETURN**	**RTNSUBR**

SELECT	**SNDF**	**SNDRCVF**	**SUBR**
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 **Monitor Message (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.

Related tasks

Using QCMDEXC with prompting in CL procedures and programs
The Execute Command (QCMDEXC) program can be used to call the prompter.

Related information

CL command finder

Using selective prompting for CL commands

Selective prompting for CL commands 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 **Change Command Definition (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)
```

The following table contains 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. When 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. When displayed, it is not input-capable. The command default is sent to the CPP.

Related concepts

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.

Related tasks

QCMDEXC program
The Execute Command (QCMDEXC) API is an IBM-supplied program that runs a single command.

QCMDCHK program
The Check Command Syntax (QCMDCHK) API is an IBM-supplied program that performs syntax checking for a single command, and optionally prompts for the command.

Using QCMDEXC with prompting in CL procedures and programs

The Execute Command (QCMDEXC) program can 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 in the following example, you can specify a value for any parameter, including FILE. In this example, FILEX is the default.:

```
CALL QCMDEXC PARM('?OVRDBF FILE(FILEX)' 19)
```

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)
```

Related tasks

Using the IBM i prompter within a CL procedure or program
You can request prompting within the interactive processing of a CL procedure or program.

Related information

Execute Command (QCMDEXC) API

Entering program source

The programmer menu can be used to enter program source.

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.

Related information

Start Programmer Menu (STRPGMMNU) command

Using the Start Programmer Menu command

The **Start Programmer Menu (STRPGMMNU)** command performs the same function as a call to QPGMMENU and fills in the standard input fields.

To fill in the standard input fields at the bottom of the menu, you can use the following command parameters:

- 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.

Note: By using the code example, you agree to the terms of the "Code license and disclaimer information" on page 610.

```
      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. For more information, see "Passing parameters" on page 273.

Related information

Start Programmer Menu (STRPGMMNU) command

Using the EXITPGM parameter of the Start Programmer Menu command

The EXITPGM parameter can be used for various 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 **Submit Job (SBMJOB)** command.

  For example, the user parameter of the **SBMJOB** command can 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 **Retrieve Job Attributes (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 **Create Physical File (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 control language (CL) commands on the system.

- The QIBM_QCA_CHG_COMMAND exit point can register only one exit program for a specific CL command. The program that is specified for this exit point is called by the command analyzer before it passes control to the prompter.
- You can register up to 10 exit programs for each CL command for the QIBM_QCA_RTV_COMMAND exit point. The command analyzer calls these exit programs after running the validity checking program (VCP). The exit program can be called either before or after running the command processing program (CPP) for the command.

Related information

API Exit Programs

Designing application programs for DBCS data

When you design application programs to process double-byte data or convert alphanumeric application programs to double-byte programs, consider this information.

Designing DBCS application programs

You can design your application programs for processing double-byte data in the same way you design application programs for processing alphanumeric data, with some 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.

Related information
Working with DBCS data

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 change an application program to a double-byte application program, perform these steps:

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

Different keyboard shifts can be used within a CL program.

In the following program, note how the double-byte data is used only as text values in this program; the commands themselves are in alphanumeric characters.

Note: By using the code example, you agree to the terms of the "Code license and disclaimer information" on page 610.

```
            PGM
            DCLF       FILE(IGCTEST)
START:      CHGVAR     VAR(&OUTPUTA) VALUE('ABCDETGHIJ')
            CHGVAR     VAR(&OUTPUTJ) VALUE('ABCD')
            CHGVAR     VAR(&BOTHJ) VALUE('ABCD')
            CHGVAR     VAR(&OUTPUTE) VALUE('EFGH')
            CHGVAR     VAR(&OUTPUTO) VALUE('ABCDF')
LOOP:       SNDRCVF
            IF         COND(&IN01 *EQ '0') THEN(RETURN)
            CHGVAR     VAR(&OUTPUTA) VALUE(&INPUTA)
            CHGVAR     VAR(&OUTPUTJ) VALUE(&INPUTJ)
            CHGVAR     VAR(&OUTPUTE) VALUE(&INPUTE)
            CHGVAR     VAR(&BOTHE) VALUE(&INPUTE)
            CHGVAR     VAR(&OUTPUTO) VALUE(&INPUTO)
            GOTO       CMDLBL(LOOP)
            ENDPGM
```

When run, this program shows you how the different keyboard shifts for DDS display files are used.

```
    ALPHANUMERIC SHIFT (A) . . . . . .  __________     ABCDETGHIJ
    DBCS-ONLY (J) . . . . . . . . . .   ________       ABCD
    DBCS-EITHER (E) . . . . . . . . .   ________       EFGH
    DBCS-OPEN (O) . . . . . . . . . .   __________     ABCDF
    DBCS-ONLY (J) INPUT-OUTPUT . . . .  ________
    DBCS-EITHER (E) INPUT-OUTPUT . . .  ________
    INPUT 0 TO END . . . . . . . . . .  _
```

The following code snippet shows the data description specifications (DDS) of the display file IGCTEST.

Note: By using the code example, you agree to the terms of the "Code license and disclaimer information" on page 610.

```
|...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
00010A         R IGCTEST
00020A                                 04 07'ALPHANUMERIC SHIFT (A) . . . . . .'
00030A           INPUTA        10A  I 04 43
00040A           OUTPUTA       10A  O 04 58
00050A                                 05 07'DBCS-ONLY (J) . . . . . . . . . . '
00060A           INPUTJ        08J  I 05 43
00070A           OUTPUTJ       08J  O 05 58
00080A                                 06 07'DBCS-EITHER (E) . . . . . . . . . '
00090A           INPUTE        08E  I 06 43
00100A           OUTPUTE       08E  O 06 58
00110A                                 07 07'DBCS-OPEN (O) . . . . . . . . . . '
00120A           INPUTO        09O  I 07 43
00130A           OUTPUTO       09O  O 07 58
00140A                                 08 07'DBCS-ONLY (J) INPUT-OUTPUT . . . .'
00150A           BOTHJ         08J  B 08 43
00160A                                 09 07'DBCS-EITHER (E) INPUT-OUTPUT . . .'
00170A           BOTHE         08E  B 09 43
00180A                                 10 07'INPUT 0 TO END . . . . . .'
00190A           INO1          01A  B 10 43
```

Note:

- Specify IGCDTA(*YES) on the CRTDSPF command while creating the display file IGCTEST.
- Specify IGCDTA(*YES) on the CRTSRCPF command while creating the source physical file for the CL program .

Unicode support in control language

Control language (CL) supports Unicode (UTF-16) parameter values. With this support, your programs can pass a whole set of Unicode characters instead of just the job's EBCDIC set.

Unicode overview

Unicode is a standard that precisely defines a character set as well as a small number of encodings for it. It enables you to handle text in any language efficiently.

What is Unicode

Unicode is a standard that enables you to handle text in any language. With Unicode, a single application can work for users all over the world.

Before Unicode, the encoding systems that existed did not cover all the necessary numbers, characters, and symbols in use. Different encoding systems might assign the same number to different characters. If you used the wrong encoding system, your output might not have been what you expected to see.

Unicode provides a unique number for every character, regardless of the platform, language, or program.

By using Unicode, you can develop a software product that works with various platforms, languages, and countries.

Why use Unicode

The operating system provides multilingual support. Unicode provides the means to store and retrieve data in the national language that you choose in a single file. Unicode, therefore, provides one database file to support all text needs, regardless of the language of the input device. For example, the same file can have text in Greek, Russian, and English.

Related information

Working with Unicode

Design of Unicode in control language

With the Unicode support in control language (CL), the command processing program (CPP) can always get its data in either extended binary-coded decimal interchange code (EBCDIC) or UTF-16, regardless of how the data is passed to CL. This can best be described as having two related parts that together help you pass Unicode data to your application.

Parameter support

With the PARM support in CL, you can specify if you want this parameter viewed as EBCDIC or Unicode. EBCDIC is the default. By specifying Unicode, the CPP always receives this value in Unicode, even if it is provided to CL as EBCDIC.

Parser support

The parser support in CL converts the provided input, so that the CPP always gets the type of value you specified in either EBCDIC or Unicode. It can convert Unicode to the job CCSID or convert the job CCSID to Unicode.

The CPP does not need to identify whether the input was provided as Unicode, because the CL run time converts the supplied parameter value to the requested type automatically.

Because of this support, one CL command can support calls to both Unicode and non-Unicode data.

How to specify passing Unicode to the CPP

The Unicode support is an option on the CCSID of value (CCSID) parameter of the Parameter (PARM) command . This value can be specified only when the TYPE parameter value is *CHAR or *PNAME. Following are the possible values for the CCSID parameter:

***JOB**
If the command string was originally in Unicode, the value is converted to the job CCSID. If the original command string was not in Unicode, the job CCSID is assumed and no conversion is done.

***UTF16**
The parameter value is converted to UTF-16. If the original input was not in Unicode, it is assumed to be in the job CCSID.

Example: Passing the EBCDIC and Unicode value

The example shows how to specify the command to pass the extended binary-coded decimal interchange code (EBCDIC) and Unicode value.

```
START:      CMD        PROMPT('EXAMPLE FOR UNICODE')
            PARM       KWD(STRING1) TYPE(*CHAR) LEN(40) DFT(ABC123) +
                         MIN(0) CCSID(*JOB) PROMPT('String one') +
                         /* Passed in job CCSID (EBCDIC) */

            PARM       KWD(STRING2) TYPE(*CHAR) LEN(40) DFT(ABC123) +
                         MIN(0) CCSID(*UTF16) PROMPT('String two') +
                         /* Passed in Unicode (UTF-16)*/
```

In the example, STRING1 is sent to the command processing program (CPP) in Job CCSID and STRING2 is sent in Unicode. CL converts whatever values are provided to ensure that they match the specified values. Unicode is only supported for the *PNAME parameter or the *CHAR parameter.

If *PNAME or *CHAR is specified for the TYPE parameter and *UTF16 is specified for the CCSID of value (CCSID) parameter, the number of bytes passed to the CPP can be twice the number that is specified for the LEN value. The value that is passed to the CPP is increased with UTF-16 blank characters on the right. For example, if the data X'00410042' is only 2 characters long but the field is 4 characters long, then the data passed to the CPP is x'0041004200200020'.

The Parameter Definition (PARM) command information contains more about the impacts of the Unicode specification for the parameter values.

Related information
Parameter Definition (PARM)

Calling Unicode-enabled commands

Read this information to know how to pass Unicode to a Unicode-enabled CL command. This information applies to system-provided commands that have been Unicode enabled as well as any user-written commands that take advantage of the Unicode-enabled support.

CL command support for Unicode

Several system-provided CL commands support Unicode on some parameters. These commands include a note in the documentation that states the parameter is Unicode enabled. Depending on how the command is called, you can pass these commands a Unicode value for the parameter that can contain any Unicode character.

For example, the Directory (DIR) parameter of the Create Directory (MKDIR) command is Unicode enabled.

Before the DIR parameter was Unicode enabled, if you had a program that called the Process Commands (QCAPCMD) API to issue a MKDIR command, you had to pass the DIR name in the job's CCSID. This forced you to choose an EBCDIC CCSID and this might have limited your choice. For example, this limitation would show up if you wanted to create a directory that had both Greek and Russian characters. You could not do this because no EBCDIC CCSID contains both sets of characters.

Now that the DIR parameter is Unicode enabled, you can change your program to call the QCAPCMD API and pass it a Unicode directory path name on the MKDIR command. Because the name is in Unicode, it can contain both Greek and Russian characters.

If a command has been Unicode enabled, this does not affect any other invocations of the command. For example, there are no changes to any calls to the command from the QCMD prompt line. However, such calls cannot take advantage of the Unicode enablement either.

How to call the QCAPCMD API to pass Unicode to a CL command

When you call the QCAPCMD API, you have the option to pass the entire string to the QCAPCMD API as an EBCDIC or a Unicode value.

These are the possible CCSID values:

- 0. The command input string is in the job CCSID.
- 1208. The command string is in UTF-8.
- 1200. The command string is in UTF-16.

If you set this value to 1208 or 1200, the entire input string must be passed in Unicode. That is, all data must be passed as the Unicode value, not just the parameters you have set to Unicode.

How to use UTF-8 source files to pass Unicode to a CL command

You can create a source member that can contain any Unicode character (character that is encoded in UTF-8) and submit this to CL for processing. For example, this allows a batch job to copy a whole set of integrated file system objects regardless of the languages in the names. Before CL is Unicode enabled, you could not do this in a single batch job because all information would be converted to one EBCDIC job CCSID and no Job CCSID supports more than one or two languages.

Here is an example of a source member that is encoded in UTF-8 with Russian and Greek file names:

```
//BCHJOB
MKDIR DIR('~/sample')
MKDIR DIR('~/sample/one')
MKDIR DIR('~/sample/Κέντρα πληροφοριών')
MKDIR DIR('~/sample/получения дополнительной')
```

```
MKDIR DIR('~/sample/my backup info')
//ENDBCHJOB
```

The UTF-8 file can be created by the following command:

```
CRTSRCPF FILE(MYLIB/UTF8TEST) MBR(TEST) TEXT('test of utf8 file') CCSID(1208)
```

After that, you need to use a UTF-8 enabled method to update the file. For example, you can use the Rational Development Studio for i licensed program to edit this UTF-8 file directly. Then, you can use the Submit Data Base Jobs (SBMDBJOB) command to issue the request.

QCMD prompt line support for Unicode

The system QCMD prompt line does not support Unicode.

Related information

Process Commands (QCAPCMD) API

Loading and running an application from tape or optical media

The **Load and Run Media Program (LODRUN)** command loads and runs an application written by another user or by a software vendor from tapes or optical media that is 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 **Restore Object (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.

Related information

Load and Run (LODRUN) command

Example: QINSTAPP program

This example program can be saved to tape or optical media and then loaded on the system by using the **Load and Run Media Program (LODRUN)** command.

The **LODRUN** command passes control of the system to the program, which then performs the tasks written into the program.

This example 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

Note: By using the code example, you agree to the terms of the “Code license and disclaimer information” on page 610.

```
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 7. Example of an application using the LODRUN command

Related information

Load and Run (LODRUN) command

Transferring control to improve performance

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 (CALL)** command is used, the program called returns control to the program containing the **Call (CALL)** command. When a **Transfer Control (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 (CALL)** command.

Note: The **Transfer Control (TFRCTL)** command is not valid in Integrated Language Environment (ILE) CL procedures.

Related information

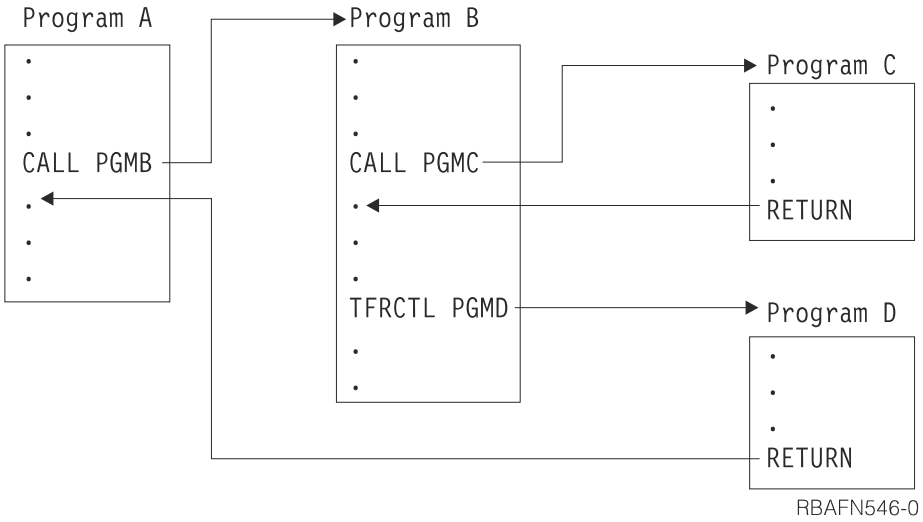
Transfer Control (TFRCTL) command

Call Program (CALL) command

Example: Using the Transfer Control command

This is an example of transferring control to improve performance.

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 **Transfer Control (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 **Transfer Control (TFRCTL)** command cannot pass a variable that was not passed to the program running the **Transfer Control (TFRCTL)** command.

In the following example, the first **Transfer Control (TFRCTL)** is valid. The second **Transfer Control (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.

Note: By using the code example, you agree to the terms of the "Code license and disclaimer information" on page 610.

```
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
```

Related tasks

Passing parameters
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.

Related information

Transfer Control (TFRCTL) command

Passing parameters using the Transfer Control command

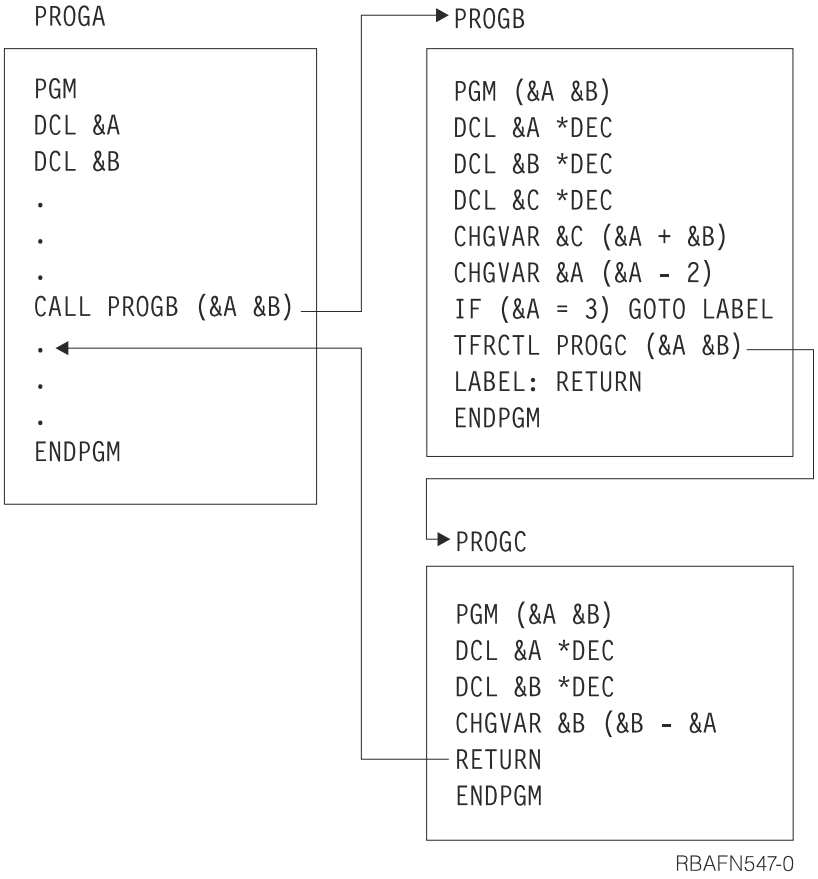
The **Transfer Control (TFRCTL)** command passes parameters to the program being called.

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 original program model (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.



Related information

Transfer Control (TFRCTL) command

Examples: CL programming

These examples demonstrate the flexibility, simplicity, and versatility of CL programs.

These CL 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 program or procedure as threadsafe unless all the commands the CL program or procedure uses are threadsafe. You can use the **Display Command (DSPCMD)** command to determine if a command is threadsafe. In addition, information (online help and command documentation) for each command indicates whether the command is threadsafe.

Related information

Multithreaded applications

Example: Initial program for setup (programmer)

In this example, 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.

Note: By using the code example, you agree to the terms of the "Code license and disclaimer information" on page 610.

```
PGM
CHGLIBL LIBL(TESTLIB QGPL QTEMP)
CHGJOB OUTQ(WSPRTR)
TFRCTL QPGMMENU
ENDPGM
```

Example: Saving specific objects in an application (system operator)

This example program ensures consistent command entry for regularly repeated procedures.

Note: By using the code example, you agree to the terms of the "Code license and disclaimer information" on page 610.

```
PGM
SAVOBJ OBJ(FILE1 FILE2) LIB(LIBA) OBJTYPE(*FILE) DEV(TAP01) +
  CLEAR(*ALL)
SAVOBJ OBJ(DTAARA1) LIB(LIBA) OBJTYPE(*DTAARA) DEV(TAP01)
SNDPGMMSG MSG('Save of daily backup of LIBA completed') +
  MSGTYPE(*COMP)
ENDPGM
```

Additional **Save Object (SAVOBJ)** commands can, of course, be added. However, this program relies on the operator selecting the correct tape for each periodic backup of each application. This can be controlled by assigning unique names to each 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 tape differently and write the program to compare the name of the tape against the correct name for that week.

Example: Recovery from abnormal end (system operator)

In this example, the system operator recovers from an abnormal end.

Note: By using the code example, you agree to the terms of the "Code license and disclaimer information" on page 610.

```
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
```

Example: Timing out while waiting for input from a device display

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.

Note: By using the code example, you agree to the terms of the "Code license and disclaimer information" on page 610.

```
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
```

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 (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 **Monitor Message (MONMSG)** command issues the SIGNOFF command.

A **Send File (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.

Example: Performing date arithmetic

This program illustrates how to write a CL program that adds or subtracts a given number of days for the current system date.

Note: By using the code example, you agree to the terms of the "Code license and disclaimer information" on page 610.

```
/* 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
```

Debugging CL programs and procedures

Debugging means detecting, diagnosing, and eliminating errors in Integrated Language Environment (ILE) or original program model (OPM) programs.

Debugging ILE programs

To debug your Integrated Language Environment (ILE) programs, use the ILE source debugger.

You can perform the following tasks:

- 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.

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
- 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 parameter (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.

Related tasks

Starting debug mode
Debug mode is a special environment in which the testing functions can be used in addition to the normal system functions.

Related reference

Parameter values used for testing and debugging
The IBM i operating system includes functions that let a programmer observe operations performed as a program runs.

Related information

Change Debug (CHGDBG) command

Start Debug (STRDBG) command
CL command finder
ILE Concepts

Debug commands

Many debug commands are available for use with the Integrated Language Environment (ILE) source debugger.

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

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

The following table summarizes these debug commands. The online help for the ILE source debugger describes the debug commands and explains their allowed abbreviations.

Table 25. 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 of text.
UP	Moves the displayed window of source towards the beginning of the view by the amount entered.
DOWN	Moves the displayed window of source towards the end of the view by the amount entered.
LEFT	Moves the displayed window of source to the left by the number of characters entered.

Table 25. ILE source debugger commands (continued)	
Debug command	**Description**
RIGHT	Moves the displayed window of source to the right by the number of characters entered.
TOP	Positions the view to show the first line.
BOTTOM	Positions the view to show the last line.
NEXT	Positions the view to the next breakpoint in the source currently displayed.
PREVIOUS	Positions the view to the previous breakpoint in the source currently displayed.
HELP	Shows the online help information for the available source debugger commands.
SET	Allows you to change debug options such as case sensitivity for FIND requests, whether production files may be updated while in debug mode, and the enabling or disabling of original program model (OPM) source debug support.
WATCH	Displays a list of the currently active watch conditions.

Related tasks

Adding program objects to a debug session
After you start a debug session, you can add more program objects to the session.

Changing the value of variables
To change the value of a variable, use the EVAL command with the assignment operator (=).

Equating a name with a variable, an expression, or a command
To equate a name with a variable, an expression, or a debug command for shorthand use, use the EQUATE debug command.

Related reference

Using the BREAK and CLEAR debug commands to set and remove conditional breakpoints
An alternate method of setting and removing conditional breakpoints is to use the BREAK and CLEAR debug commands.

Setting and removing conditional breakpoints
You can use the Work with Breakpoints display to set or remove a conditional breakpoint, the BREAK debug command to set a breakpoint, or the CLEAR debug command to remove a breakpoint.

Displaying variables
To display the value of a variable, use the Display Module Source display or the EVAL debug command.

Displaying variable attributes
The variable attributes are the size (in bytes) and type of the variable. To display the attributes, use the Attribute (ATTR) debug command.

Preparing a program object for a debug session

Before you can use the Integrated Language Environment (ILE) source debugger, you must prepare the program object.

To do this, you must use either the **Create CL Module (CRTCLMOD)** or **Create Bound CL (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

Related information

Create Bound CL Program (CRTBNDCL) command

Create CL Module (CRTCLMOD) command

Using a root source view to debug ILE programs

A root source view contains the source statements of the source member.

To use the root source view with the Integrated Language Environment (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 **Create CL Module (CRTCLMOD)** or **Create Bound CL (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*.

Related information

Create Bound CL Program (CRTBNDCL) command

Create CL Module (CRTCLMOD) command

Using a listing view to debug ILE programs

A listing view is similar to the source code portion of the compile listing or spool file produced by the Integrated Language Environment (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 **Create CL Module (CRTCLMOD)** or **Create Bound CL (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)
```

Related information

Create Bound CL Program (CRTBNDCL) command

Create CL Module (CRTCLMOD) command

Encrypting the debug listing view

You can store an encrypted version of a debug listing view for a compiled CL procedure in the generated module (*MODULE) object.

You can create a compiler listing view of the ILE CL procedure in a CL module by specifying *LIST for the Debugging view (DBGVIEW) parameter on the **Create CL Module (CRTCLMOD)** command or the **Create Bound CL Program (CRTBNDCL)** command. The listing view can make it easy to debug your ILE CL procedures. However, the listing view can be seen by anyone with sufficient authority to the program or service program object that contains the CL module.

Support was added in IBM i 7.1 to allow you to encrypt the listing view debug data when the ILE CL module is created. You can specify an encryption key value for the Debug encryption key (DBGENCKEY) parameter on the **CRTCLMOD** and **CRTBNDCL** commands. When you start a debug session, you are prompted for the encryption key value. If the same value is not specified for the debug session that was specified when the CL module was created, no listing view is shown.

This encrypted debug listing view support allows you to send a program or service program to another system without allowing users on that system to see your CL source code by using the debug listing view. If a problem occurs in the CL procedure, you could sign on to the other system remotely. Once you are signed on to the other system, you can start a debug session and provide the encryption key. Once the encryption key is entered, the listing view data is available for your debug session.

Related information

Create Bound CL Program (CRTBNDCL) command

Create CL Module (CRTCLMOD) command

Using a statement view to debug ILE programs

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 Integrated Language Environment (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 **Create CL Module (CRTCLMOD)** or **Create Bound CL (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)
```

Related information

Create Bound CL Program (CRTBNDCL) command

Create CL Module (CRTCLMOD) command

Starting the ILE source debugger

After you create the debug view, you can begin debugging your application using the Integrated Language Environment (ILE) source debugger.

To start the ILE source debugger, use the **Start Debug (STRDBG)** command. After the debugger is started, it remains active until you enter the **End Debug (ENDDBG)** command.

Initially, you can add as many as 20 program objects and 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 **Start Debug (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)
```

Related information

Start Debug (STRDBG) command

Adding program objects to a debug session

After you start a debug session, you can add more program objects to the session.

To add Integrated Language Environment (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. 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 8. Adding an ILE program object to a 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 9. Adding an ILE program object to a 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 original program model (OPM) programs to a debug session, use the **Add Program (ADDPGM)** command. A debug session can include up to 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 **Start Debug (STRDBG)** command. If the OPMSRC parameter was not specified on the **Start Debug (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.

Related concepts

Debug commands

Many debug commands are available for use with the Integrated Language Environment (ILE) source debugger.

Related information

Change Debug (CHGDBG) command

Add Program (ADDPGM) command

Start Debug (STRDBG) command

Removing program objects from a debug session

After you start a debug session, you can remove program objects from the session.

To remove Integrated Language Environment (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. 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 10. Removing an ILE program object from a 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 11. 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 original program model (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

To view the source of a program object, use the Display Module Source display.

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(*LIST)

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 Integrated Language Environment (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

To change the module object that is shown on the Display Module Source display, use option 5 (Display module source) on the Work with Module List display. 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 as follows.

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 12. Displaying 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 module object view

You change the view of the module object 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 following views of an ILE CL module object are available depending on the values you specify when you create an ILE CL module object:

- Root source view
- Listing view
- Statement view

The current view is listed at the top of the window, and the other views that are available are shown as follows. 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 13. 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

Breakpoints stop a program object at a specific point when the program 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 (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

The system enables you to set and remove unconditional breakpoints in several ways.

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 the following on the debug command line:

```
BREAK line-number
```

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 the following on the debug command line:

```
CLEAR line-number
```

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 the following on the debug command line:

```
BREAK procedure-name/statement-number
```

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 use the Work with Breakpoints display to set or remove a conditional breakpoint, the BREAK debug command to set a breakpoint, or the CLEAR debug command to remove a breakpoint.

Related concepts

Debug commands

Many debug commands are available for use with the Integrated Language Environment (ILE) source debugger.

Using the Work with Breakpoints display

To set or remove breakpoints, use the Work with Breakpoints display.

Note: The relational operators supported for conditional breakpoints are <, >, =, <=, >=, and <> (not equal).

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 the following figure. To set a conditional breakpoint, type the following and press Enter:

- 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,

For example, to set a conditional breakpoint at debugger line 35, as shown in the following figure, type the following and press Enter:

- 1 (Add) in the *Opt* field,
- 35 in the *Line* field,
- type &I=21 in the *Condition* field,

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 14. 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 to set and remove conditional breakpoints

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 the following on the debug command line:

```
BREAK line-number WHEN expression
```

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.

To remove a conditional breakpoint by using the CLEAR debug command, type the following on the debug command line:

```
CLEAR line-number
```

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 the following on the debug command line:

```
BREAK procedure-name/statement-name WHEN expression
```

Procedure-name is the name of your CL module. *Statement-number*(from the compiler listing) is the statement number where you want to stop.

Related concepts

Debug commands
Many debug commands are available for use with the Integrated Language Environment (ILE) source debugger.

National Language Sort Sequence
National Language Sort Sequence (NLSS) applies only to non-numeric conditional breakpoint expressions of type Char-8.

National Language Sort Sequence

National Language Sort Sequence (NLSS) applies only to non-numeric conditional breakpoint expressions of type Char-8.

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)

Table 26 on page 405 shows 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 **Create CL Module (CRTCLMOD)** or **Create Bound CL Program (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 26. 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]

Notes:

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>.

Related reference
Using the BREAK and CLEAR debug commands to set and remove conditional breakpoints

An alternate method of setting and removing conditional breakpoints is to use the BREAK and CLEAR debug commands.

Examples: Conditional breakpoint

These examples show how to set conditional breakpoints.

```
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

To remove all breakpoints, conditional and unconditional, from a program object that has a module object shown on the Display Module Source display, use the CLEAR PGM debug command.

To use the debug command, type the following on the debug command line:

```
CLEAR PGM
```

The breakpoints are removed from all of the modules bound to the program or service program.

Using instruction stepping

After a breakpoint is encountered, you can run a specified number of statements of a program object, then stop the program again.

After stopping the program, you 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.

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. Step over is the default mode of F10 (Step). 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.

F10 (Step) to step over program objects or F22 (Step into) to step into program objects

When you step over the called program object, then the CALL statement and the called program object are run as a single step. When you step into the called program object, then each statement in the called program object is run as a single step.

The called program object is run to completion before the calling program object is stopped at the next step. 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.

Step over is the default step mode. You can step over program objects by using:

- F10 (Step) on the Display Module Source display
- The STEP OVER debug command

You can step into program objects by using:

- F22 (Step into) on the Display Module Source display
- The STEP INTO debug command

Using the STEP debug command to step through a program object

The default number of statements to run, using the STEP debug command, is one. However, you can change the step number.

To step through a program object using the STEP debug command, type the following on the debug command line:

```
STEP number-of-statements
```

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, type the following on the debug command line:

```
STEP 5
```

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.

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 the following on the debug command line:

```
STEP number-of-statements OVER
```

If one of the statements that are run contains a CALL statement to another program object, the Integrated Language Environment (ILE) source debugger steps over the called program object.

You can also 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 the following on the debug command line:

```
STEP number-of-statements INTO
```

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, type the following on the debug command line:

```
STEP 5 INTO
```

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

To display the value of a variable, use the Display Module Source display or the EVAL debug command.

To display a variable using the Display Module Source display , place your cursor on the variable that you want to display and press F11 (Display variable). The current value of the variable is shown on the message line at the bottom of the Display Module Source display.

```
                          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 15. Displaying a variable using F11 (Display variable)

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 the following on the debug command line:

```
EVAL variable-name
```

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 the previous example, type:

```
EVAL &DAYOFWK
```

The message line of the Display Module Source display shows `&DAYOFWK = 3.` as in the previous example.

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.

Related concepts

Debug commands

Many debug commands are available for use with the Integrated Language Environment (ILE) source debugger.

Example: Displaying logical variable

This example uses the EVAL debug command to display the value of logical variables.

```
CL declarations:   DCL    VAR(&LGL1) TYPE(*LGL) VALUE('1')

Debug command:     EVAL &LGL1

Result:            &LGL1 = '1'
```

Examples: Displaying character variable

This example uses the EVAL debug command to display the value of character variables.

```
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.

Example: Displaying decimal variable

This example uses the EVAL debug command to display the value of decimal variables.

```
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
```

Example: Displaying variables as hexadecimal values

To display the value of a variable in hexadecimal format, use the EVAL debug command.

To display a variable in hexadecimal format, on the debug command line, type the following:

```
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

To change the value of a variable, use 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 the following on the debug command line:

```
EVAL variable-name = value
```

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*.

Look at the next example:

```
EVAL &COUNTER = 3.0
```

The previous example changes the value of *&COUNTER*; to 3.0 and shows the following on the message line of the Display Module Source display:

```
&COUNTER = 3.0 = 3.0
```

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'

Related concepts
Debug commands

Many debug commands are available for use with the Integrated Language Environment (ILE) source debugger.

Example: Changing logical variable

To change the value of a logical variable, use the EVAL debug command.

```
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'
```

Examples: Changing character variable

This example changes the value of a character variable using the EVAL debug command.

```
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.

Examples: Changing decimal variable

To change the value of a decimal variable, use the EVAL debug command.

```
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
```

Displaying variable attributes

The variable attributes are the size (in bytes) and type of the variable. To display the attributes, use the Attribute (ATTR) debug command.

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
```

Related concepts

Debug commands

Many debug commands are available for use with the Integrated Language Environment (ILE) source debugger.

Equating a name with a variable, an expression, or a command

To equate a name with a variable, an expression, or a debug command for shorthand use, use the EQUATE debug command.

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 the following on the debug command line:

```
EQUATE shorthand-name definition
```

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 the following on the debug command line:

```
EQUATE DC EVAL &COUNTER
```

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 the following on the debug command line:

```
DISPLAY EQUATE
```

A list of the active names is shown on the Evaluate Expression display.

Related concepts

Debug commands

Many debug commands are available for use with the Integrated Language Environment (ILE) source debugger.

Source debug and IBM i globalization

When you are working with source debug, be aware of certain conditions regarding IBM i globalization.

The following conditions exist 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*

This 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 cannot recognize a CL identifier that contains 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.

Related information

Display Module (DSPMOD) command

ILE Concepts

Operations that temporarily remove steps

When you use certain control language (CL) commands to specify your library or program, breakpoints or steps might be temporarily removed from a program while debugging.

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 CL commands can cause the breakpoint or step to be temporarily removed.

CHKOBJITG	CPY CPYLIB	CPROBJ CRTDUPOBJ	RSTLIB RSTOBJ	SAVLIB 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 original program model programs

To debug your original program model (OPM) programs, use testing functions. These functions are available through a set of commands that can be used interactively or in a batch job.

Testing functions are designed to help you write and maintain your applications. You can use testing functions to 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. You can use the functions to perform the following tasks:

- 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.

Note:

1. Debugging a high-use and concurrent program directly in the system will affect the performance of those jobs which running that particular high-use program in this system. Creating a copy of the program, and debugging the copied version will solve the performance problem.

Related tasks

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.

Related reference

Parameter values used for testing and debugging

The IBM i operating system includes functions that let a programmer observe operations performed as a program runs.

Starting debug mode

Debug mode is a special environment in which the testing functions can be used in addition to the normal system functions.

To begin testing, your program must be put in debug mode. 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 System i Navigator has been used to select Debug to check the system, issuing the **Start Debug (STRDBG)**) command causes the graphical interface to be presented. In this case, whenever one of the users specified in the user list issues the **Start Debug (STRDBG)** command, they see the System i Navigator System Debug Manager rather than the Command Entry display. If the **End Debug (ENDDBG)** command is entered and Debug in the System Debug Manager is not currently selected, issuing the **Start Debug (STRDBG)** command, again, starts the system debugger using the Command Entry display.

The following **Start Debug (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 Integrated Language Environment (ILE) source debugger to debug original program model (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)
```

Related tasks

Debugging ILE programs
To debug your Integrated Language Environment (ILE) programs, use the ILE source debugger.

Related information

Change Debug (CHGDBG) command
Start Debug (STRDBG) command
End Debug (ENDDBG) command
Remove Program (RMVPGM) command
CL command finder

Adding programs to debug mode

Before you can debug a program, you must put it in debug mode.

Any program can be run in debug mode. You can place a program in debug mode by specifying it in the PGM parameter on the **Start Debug (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 20 programs for debug mode (using either the **Start Debug (STRDBG)** or **Add Program (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 **Add Program (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 **Remove Program (RMVPGM)** command) may result in a function check. You must either remove the program using a RMVPGM command or end debug mode using an **End Debug (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 (**Add Program (ADDPGM)** command).

Related information

Change Debug (CHGDBG) command
Add Program (ADDPGM) command
Start Debug (STRDBG) command
Remove Program (RMVPGM) command
CL command finder

Preventing updates to database files in production libraries

You can use files in production libraries while you are in debug mode, and therefore need to prevent the files from being unintentionally updated.

To prevent database files in production libraries from being unintentionally changed, you can specify UPDPROD(*NO) or default to *NO on the **Start Debug (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.

Displaying the call stack

To display the call stack, use the **Display Debug (DSPDBG)** command.

The call stack 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.

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.

Related tasks

Controlling flow and communicating between programs and procedures
The **Call Program (CALL)**, **Call Bound Procedure (CALLPRC)**, and **Return (RETURN)** commands pass control back and forth between programs and procedures.

Related information

Display Debug (DSPDBG) command

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 or an ILE RPG program compiled with DFTACTGRP(*YES) 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.

Related information

Reclaim Resources (RCLRSC) command

Handling unmonitored messages

When a function check occurs in an interactive debug job, the system provides default handling and gives you control instead of stopping the program.

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, high-level language (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 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.









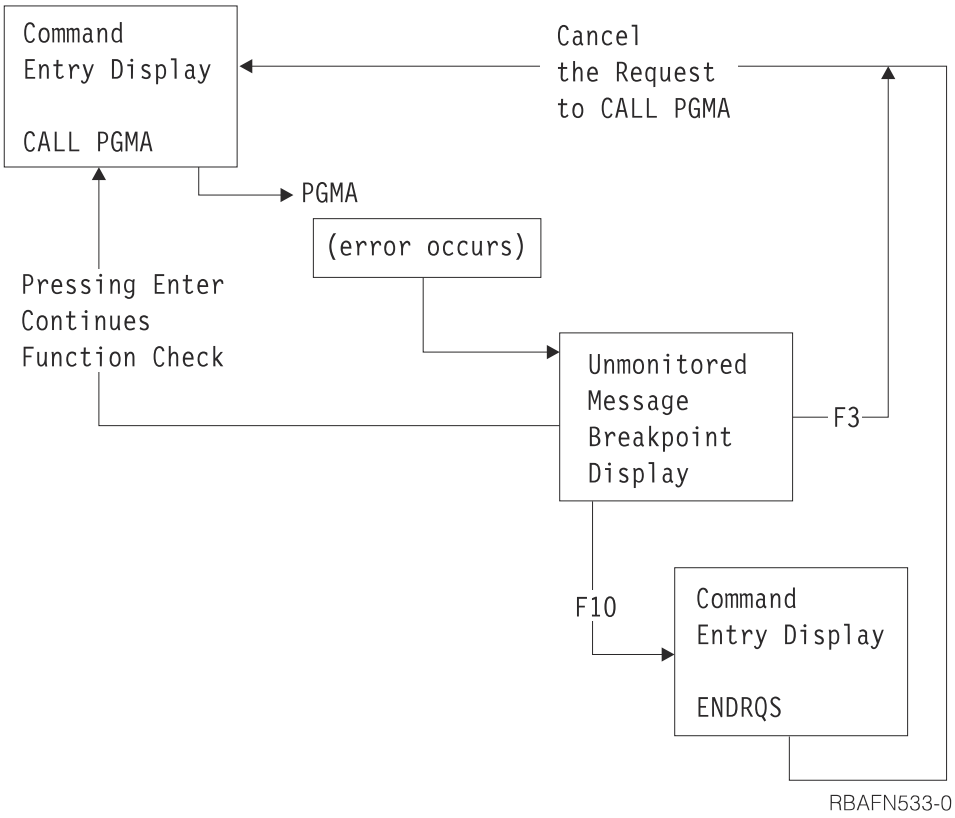
Press Enter to continue.

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 typical 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 figure shows how the **ENDRQS** command works.



Program calls are destroyed when the **ENDRQS** command is entered. (In the previous figure, the program call of PGMA is destroyed.)

Related information

End Request (ENDRQS) command

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).

Related information

Add Breakpoint (ADDBKP) command

Adding breakpoints to programs

To add breakpoints to the program you want to debug, use the **Add Breakpoint (ADDBKP)** command.

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.

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: You must enclose CL variables within single quotation marks.

The source for CUS310 looks like this.

Note: By using the code example, you agree to the terms of the “Code license and disclaimer information” on page 610.

```
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.

Related information

Resume Breakpoint (RSMBKP) command

Add Breakpoint (ADDBKP) command

CL command finder

Adding conditional breakpoints

To 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.
- 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 ')
```

Related reference

Operators in expressions
Operators are used in expressions to indicate an action to be performed on the operands in the expression or the relationship between the operands.

Related information

Add Breakpoint (ADDBKP) command
Display Breakpoint (DSPBKP) command

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.

Related information

Remove Breakpoint (RMVBKP) command

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.

Related information

Add Trace (ADDTRC) command
DisplayTrace Data (DSPTRCDTA) 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 **Start Debug (STRDBG)** or **Change Debug (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 (**Remove Trace (RMVTRC)** command), clear the trace data (**Clear Trace Data (CLRTRCDTA)** command), or change the maximum (MAXTRC parameter on the **Change Debug (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 quotation marks. 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.

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
```

Related tasks

Debugging at the machine interface level

To debug your programs at the machine interface (MI) level, 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.

Related information

Change Debug (CHGDBG) command

Start Debug (STRDBG) command

CL command finder

Using instruction stepping

To step through the instructions of a program, use the **Start Debug (STRDBG)** command or the **Change Debug (CHGDBG)** command.

When using these commands, set the MAXTRC parameter to 1 and the TRCFULL parameter to *STOPTRC. When you specify a trace range (**Add Trace (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 (**Display 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

To specify whether the trace information is removed from the system or left on the system after the information is displayed, use the **Display Trace Data (DSPTRCDTA)** command.

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.

Related information

Display Trace Data (DSPTRCDTA) command

Clear Trace Data (CLRTRCDTA) command

Removing traces from programs

The **Remove Trace (RMVTRC)** command removes all or some of the trace ranges specified in one or more **Add Trace (ADDTRC)** commands.

Removing a trace range consists of specifying the statement identifiers used on the **Remove Trace (RMVTRC)** command, or specifying that all ranges be removed.

You can use the STMT parameter on the **Remove Trace (RMVTRC)** command to specify:

- All high-level language (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 **Add Trace (ADDTRC)** command.
- The start and stop trace location of the HLL statements or system instructions or both to be removed.

The **Remove Program (RMVPGM)** and **End Debug (ENDDBG)** commands also remove traces, but they also remove the program from debug mode.

Related information

Add Trace (ADDTRC) command

Remove Trace (RMVTRC) command

End Debug Mode (ENDDBG) command

Remove Program (RMVPGM) command

Displaying testing information

Displaying testing information can help you review your job in debug mode.

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.

Related information

Display Debug (DSPDBG) command
Display Trace (DSPTRC) command
Display Breakpoints (DSPBKP) command

Displaying the values of variables

If you specify the variable names on the **Add Breakpoint (ADDBKP)** command, the values of program variables are displayed automatically on the breakpoint display. You can also 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 **Display Program Variable (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 single quotation marks. 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 **Display Program Variable (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 high-level languages (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.

Related information

Add Breakpoint (ADDBKP) command

Display Program Variable (DSPPGMVAR) command

Changing the values of variables

To change the value of a program variable in debug mode, 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.

Related information

Change Program Variable (CHGPGMVAR) command
Change Pointer (CHGPTR) command
Change HLL Pointer (CHGHLLPTR) command

Reasons for using a job to debug another job

You might want to use a separate job to debug programs that run in another job in these situations.

- 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 that are 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 **Submit Job (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 want 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.

Related information

Add Breakpoint (ADDBKP) command
Add Trace (ADDTRC) command
CL command finder

Debugging batch jobs that are not started from job queues

You can debug batch jobs that are started on the system but are not submitted to a job queue. These jobs cannot be stopped before they start running, but they can typically be debugged.

To debug jobs not started from a job queue, follow these steps:

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.

Related information

Add Breakpoint (ADDBKP) command
Add Trace (ADDTRC) command
CL command finder

Debugging a job that is running

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 required debug commands when the program is stopped at a breakpoint.

Related information

Display Debug (DSPDBG) command
Display Trace Data (DSPTRCDTA) command
Add Breakpoint (ADDBKP) command
Add Trace (ADDTRC) command
CL command finder

Debugging another interactive job

Whether a job is running or waiting at a menu or command entry display, you can debug the job from another display.

To debug another interactive job, follow these steps:

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 you want. 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.

Related information

Start Debug (STRDBG) command
Display Debug (DSPDBG) command
CL command finder

Considerations when debugging one job from another job

Although most jobs can be debugged from another job, there are some considerations you should follow.

- 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, 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 does at a high-level language (HLL) statement number. You must always precede the ODV or MI instruction number with a slash (/) and enclose it in single quotation marks (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.

Related tasks

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.

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.

Operations that temporarily remove breakpoints

If you use certain CL commands to specify your library or program, you can temporarily remove breakpoints or statement traces from a program while the debug function is running.

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 CPYLIB	CPROBJ CRTDUPOBJ	RSTLIB RSTOBJ	SAVLIB 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.

Related information

CL command finder

Objects and libraries

Tasks and concepts specific to objects and libraries include performing functions on objects, creating libraries, and specifying object authority.

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 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.

Note: Objects can reside in both libraries and directories. (Previously, an object could reside only in a library.) This topic contains information only about objects residing in libraries.

Related concepts

IBM i objects

An *object* is a named unit that exists (occupies space) in storage, and on which operations are performed by the operating system. IBM i objects provide the means through which all data processing information is stored and processed by the IBM i operating system.

Related tasks

Data area locking and allocation

Locking and allocating a data area helps to ensure that the data area is not accessed by more than one job at a time.

Related information

Integrated fie system

Objects

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.

Object types and common attributes

Each type of object on the system has a unique purpose within the system, and each object type has a common set of attributes that describes the object.

Each type of object has an associated set of commands that process this type of object.

Related concepts

External object types

Many types of external objects are stored in libraries. Some types of external objects can only be stored in the integrated file system in directories.

Related reference

Displaying object descriptions

The **Display Object Description (DSPOBJD)** or **Work with Objects (WRKOBJ)** command displays descriptions of objects.

Related information

Display Object Description (DSPOBJD) command

Functions performed on objects

Many functions can be performed on objects. The system performs some functions automatically and you request the others 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 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 and common functions.

- 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.
- Common functions. Some common functions that apply to objects in general are explained in this guide.

Table 27. Common functions for objects

“Searching for multiple objects or a single object” on page 444
“ Authority for libraries specification” on page 446
“Placing objects in libraries” on page 449
“Describing objects” on page 454
“Displaying object descriptions” on page 454
“Retrieving object descriptions” on page 458
“Detecting unused objects on the system” on page 461
“Moving objects from one library to another” on page 466
“Creating duplicate objects” on page 468
“Renaming objects” on page 470
“Deleting objects” on page 474
“Allocating resources” on page 474
“Displaying the lock states for objects” on page 477
“Checking for the existence of an object” on page 480

Libraries

On the IBM i operating system, 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.

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. 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.

Related concepts

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.

Related tasks

Authority for libraries specification
Certain authorities can be given to users for libraries.

Related information

Independent disk pool examples

Library lists

Library lists are used by the system to locate objects.

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 the IBM i operating system 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.

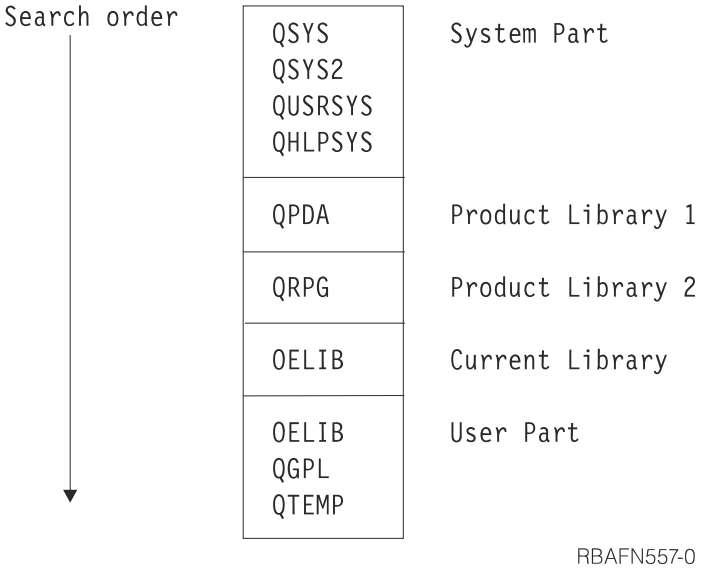
The product libraries are managed by the system, which automatically places product libraries (such as QRPG) into the reserved product library position in the library list when needed. A product library may be a duplicate of the current library or of a library in the user part of the library list.

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.

The following figure shows an example of the structure of the library list.



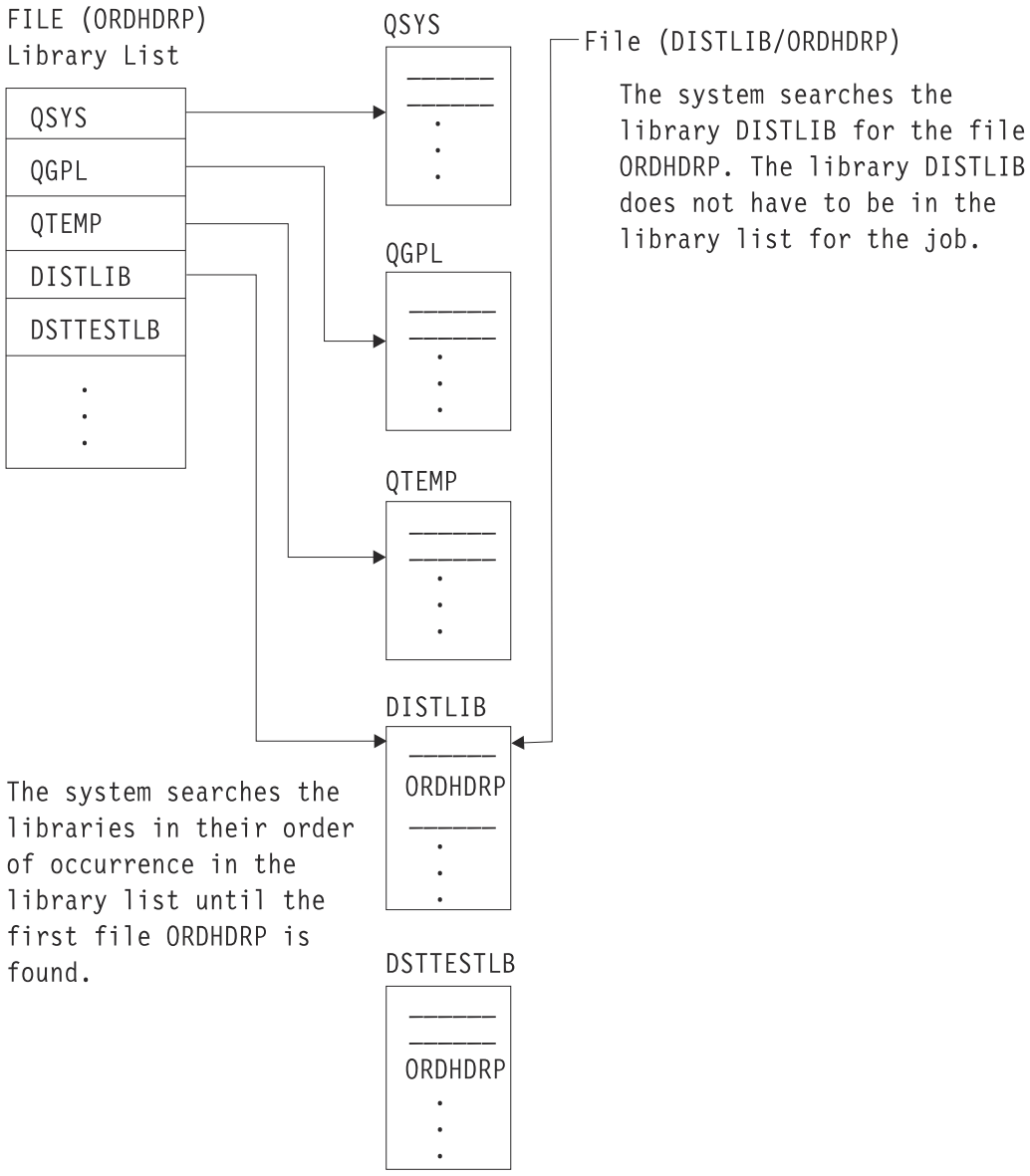
Note: The system places library QPDA in product library 1 when the source entry utility (SEU) is used. When SEU is being used to syntax check source code, a second product library can be added to product library 2. For example, if you are syntax checking RPG source, then QPDA is product library 1 and QRPG is product library 2. In most other system functions, product library 2 is not used.

Related information
Create Menu (CRTMNU) command
Change Library List (CHGLIBL) command
Create Command (CRTCMD) command
Change Current Library (CHGCURLIB) command
Add Library List Entry (ADDLIBLE) command
Edit Library List (EDTLIBL) command
Remove Library List Entry (RMVLIBLE) command

Functions of using a library list
Using a library list simplifies finding objects on the system.

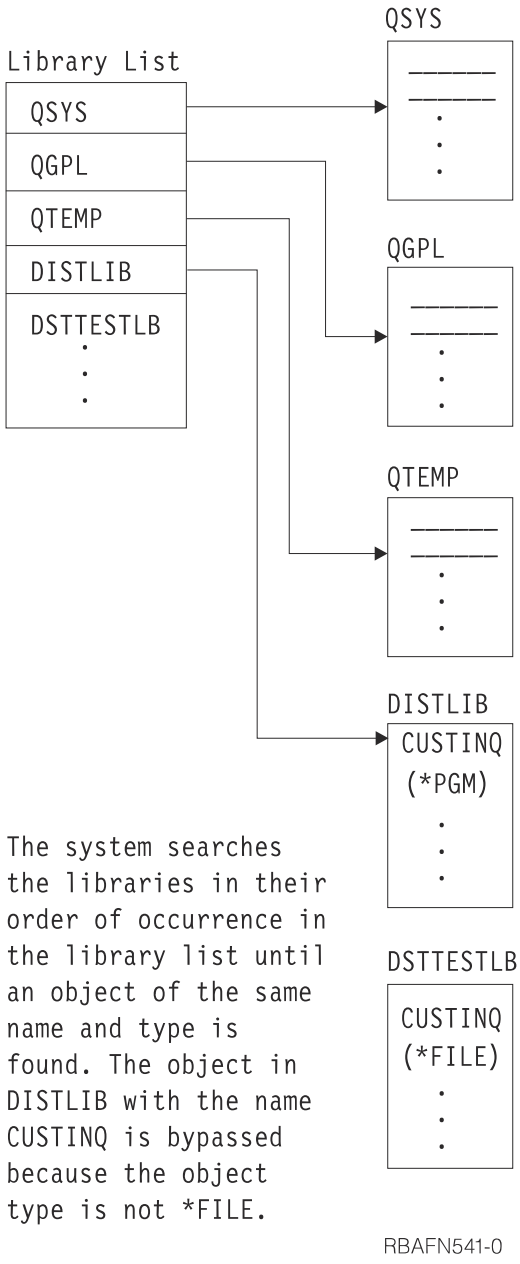
Each job has a library list associated with it. When a library list is used to find an object, each library in the list is searched in the order of its occurrence in the list until an object of the specified name and type is found. If two or more objects of the same type and name exist in the list, you get the object from the library that appears first in the library list. The following figure 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.

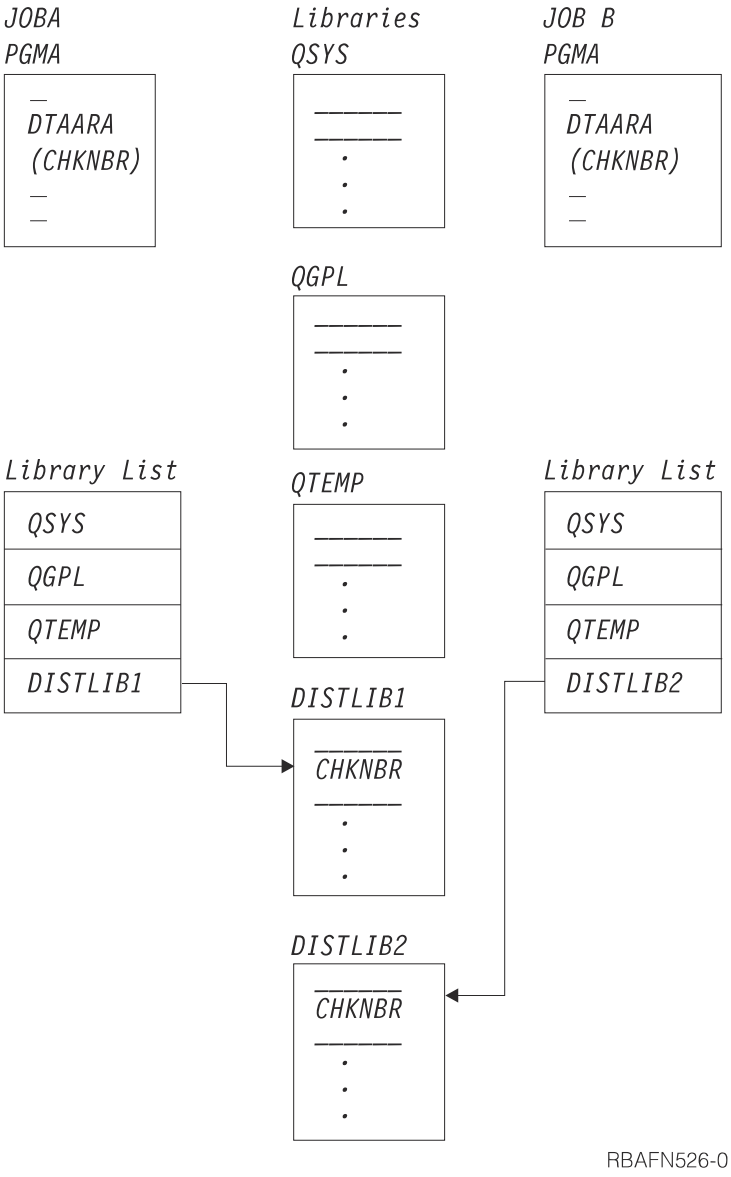


The following figure shows what happens when two objects of the same name but different types are in the library list. The system will search for CUSTINQ *FILE in the library list by specifying:

```
DSPOBJD OBJ(*LIBL/CUSTINQ) OBJTYPE(*FILE)
```



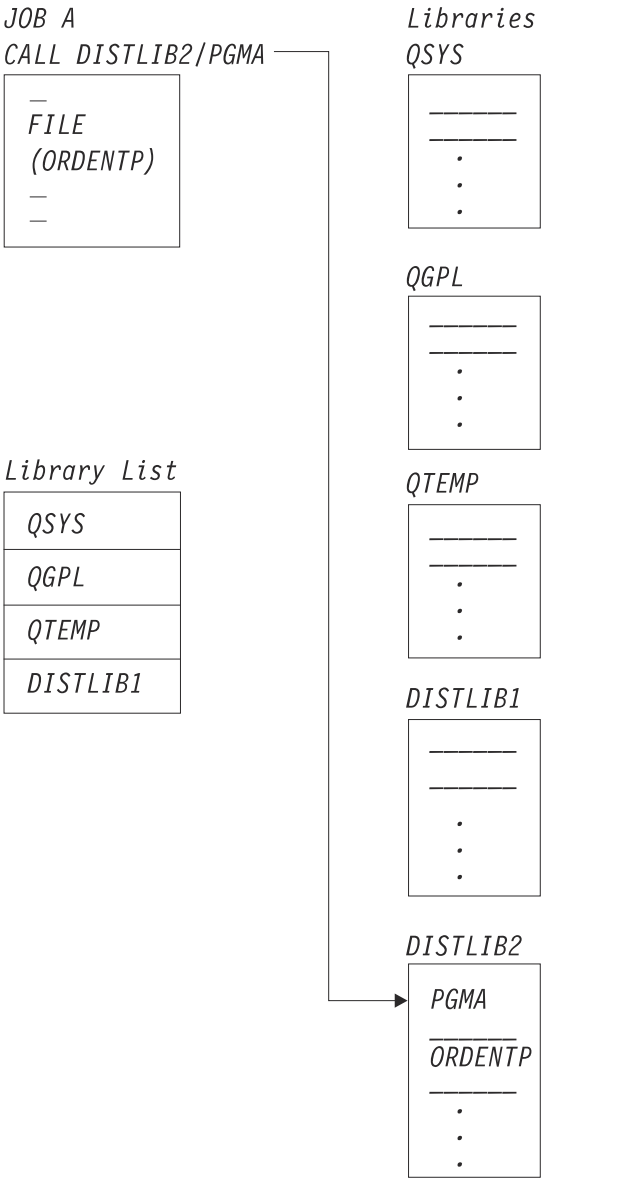
Generally, a library list is more flexible and easier to use than qualified names. More important than the advantage of not entering the library name, is the advantage of performing functions in an application on different data by using a different library list without having to change the application. 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.



RBAFN527-0

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.

Related information

Object signing and signature verification

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.

Related information

Change System Value (CHGSYSVAL) command
Change System Library List (CHGSYSLIB) command

Changing the library list

You can change the library list in a variety of ways.

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 **Change Library List (CHGLIBL)** command or the **Edit Library List (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 **Submit Job (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 **Submit Job (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.
- When a job is submitted either through a **Batch Job (BCHJOB)** command or a **Submit Job (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 **Submit Job (SBMJOB)** command, *CURRENT (the default) can be specified for the library list. *CURRENT indicates that the library list of the job issuing the **Submit Job (SBMJOB)** command is used.
- Within a job, an ADDLIBLE, a RMVLIBLE, or a 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.

Related information

Job description in Work Management
CL command finder
Change Current Library (CHGCURLIB) command
Submit Job (SBMJOB) command
Add Library List Entry (ADDLIBLE) command
Remove Library List Entry (RMVLIBLE) command

Considerations for using a library list

When you set up a library list and use it, consider this information.

- The libraries in a library list must exist on the system. The system values QSYSLIBL and QUSRLIBL are accessed when the IBM i operating system 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 the IBM i operating system is started without the library. After IBM i operating system 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 **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.

Related information

CL command finder
Submit Job (SBMJOB) command
Batch Job (BCHJOB) command
Add Library List Entry (ADDLIBLE) command
Change Library List (CHGLIBL) command

Displaying a library list

The **Display Library List (DSPLIBL)** command displays the library list for a job that is 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.

Related information

Display Job (DSPJOB) command
Display Library List (DSPLIBL) command

Using library lists to search for objects

By using library lists, you can either search for multiple objects or a single object. A generic search can be used for multiple object searches.

Generic object names searching

A generic search can be used to search for more than one object.

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.
- 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.

Related tasks

Creating a library
To create a library, use the **Create Library (CRTLIB)** command.

Deleting objects
To delete an object, you can use the **Delete Object (DLTOBJ)** command, a delete (DLTxxx) command for that type of object, or the delete option on the Work with Objects display (shown from the **Work with Libraries (WRKLIB)** display).

Related information

Generic library names

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 perform the following tasks:

- 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.
- 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.

Related concepts

Library objects
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.

Related tasks

Debugging original program model programs
To debug your original program model (OPM) programs, use testing functions. These functions are available through a set of commands that can be used interactively or in a batch job.

Creating a library

To create a library, use the **Create Library (CRTLIB)** command.

For example, the following **Create Library (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.

Related concepts

Generic object names searching
A generic search can be used to search for more than one object.

Related information

Create Library (CRTLIB) command

Authority for libraries specification

Certain authorities can be given to users for libraries.

Related concepts

Libraries
On the IBM i operating system, objects are grouped in special objects called libraries.

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.

Related information

Security reference

Object authority

The types of object authority include object operational authority, object management authority, and object existence 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

The types of data authority include add authority, read authority, update authority, execute authority, and delete 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

The types of combined authority include *USE authority, *CHANGE authority, *ALL authority, and *EXCLUDE 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.

Special considerations apply when writing a program that must be secure (for example, a program that adopts the security officer's user profile).

Related concepts

Libraries
On the IBM i operating system, objects are grouped in special objects called libraries.

Related tasks

Authority for libraries specification
Certain authorities can be given to users for libraries.

Related information

Security reference

The Display Audit Journal Entries command to generate security journal audit reports

The **Display Audit Journal Entries (DSPAUDJRNE)** command allows you to generate security journal audit reports.

The reports generated by the **Display Audit Journal Entries (DSPAUDJRNE)** command 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.

Related information

Display Audit Journal Entries (DSPAUDJRNE) command

Setting default public authority

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 the following command:

```
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 the following commands:

```
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 the following commands:

```
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 the following commands:

```
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 **Create Physical File (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 as follows:

***ALL**
All public authorities

***CHANGE**
Change authority

***EXCLUDE**
Exclude authority

***SYSVAL**
The public authority for the object being created is the value specified in system value QCRTAUT

***USE**
Use authority

authorization list name
The authorization list secures the object

Setting default auditing attribute

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 the following command, all objects created into the payroll library are audited for both read and change access:

```
CRTLIB LIB(PAYROLL) AUT(*EXCLUDE) CRTAUT(*EXCLUDE) CRTOBJAUD(*ALL)
```

Related information
Security reference

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 the IBM i operating system.

Related information
Create Library (CRTLIB) command
Create Physical File (CRTPF) command

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)
```

```
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 **Delete Library (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.

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.

Related tasks

Deleting objects

To delete an object, you can use the **Delete Object (DLTOBJ)** command, a delete (DLTxxx) command for that type of object, or the delete option on the Work with Objects display (shown from the **Work with Libraries (WRKLIB)** display).

Related information

CL command finder

Clear Library (CLRLIB) command

Delete Library (DLTLIB) command

Change Library List (CHGLIBL) command
Change System Value (CHGSYSVAL) command
Remove Library List Entry (RMVLIBLE) command

Displaying library names and contents

The **Display Library (DSPLIB)** and **Work with Libraries (WRKLIB)** commands display or print all the libraries you have authority to and show basic information about 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 **Display Library (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 **Display Library (DSPLIB)** command, the libraries are displayed in the order they are specified in the display command.
- If *LIBL or *USRLIBL is specified on the **Display Library (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 **Display Library (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 **Display Library (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 *.

Related information
Display Library (DSPLIB) command
Work with Libraries (WRKLIB) command

Displaying and retrieving library descriptions

The **Display Library Description (DSPLIBD)** and **Retrieve Library Description (RTVLIBD)** commands 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
- Auxiliary storage pool group device name of the library
- Create authority of the library
- Create object auditing of the library
- Text description of the library
- Current journaling status of the library

- Name of the current or last journal to which the library is journaled
- Name of the library that contains the journal
- Images written to the journal receiver
- Omitted journal entries
- Whether new objects inherit journaling
- Whether the journal inherit rules are overridden
- Date and time that journaling was last started
- Name of the oldest journal receiver
- Name of the library that contains the starting journal receiver
- Auxiliary storage pool (ASP) device for the starting journal receiver
- ASP group for the starting journal receiver

Related information

Display Library Description (DSPLIBD) command

Retrieve Library Description (RTVLIBD) command

Changing national language versions

The IBM i 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.

You can use either the static prompt message or the dynamic prompt message to change national language versions.

Static prompt message for control language

The static prompt message is stored in the command definition object. Different national language versions of information are displayed when you add a national language library to the library list.

When you use the static prompt message, 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 IBM i) 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 required 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 required 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 required 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 required 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 required 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.

Note: By using the code example, you agree to the terms of the "Code license and disclaimer information" on page 610.

```
PGM
  CHGSYSLIBL  LIB(QSYS2928)  /* Use French information */
ENDPGM
```

Dynamic prompt message for control language

By using the message identifiers that are stored in the command (*CMD) object when the command was created, prompt messages can be dynamically retrieved from the message file. This function enables a single command to have prompt messages in more than one national language.

To use dynamic prompt messages, you need to create a command definition object using the **Create Command (CRTCMD)** command, specify message identifiers for the PROMPT and CHOICE parameters in the *CMD object, and specify a message file for the first element of the prompt message file (PMTFILE) parameter and *DYNAMIC for the second element. By having a copy of the prompt message file in the national language that you need in the library list at prompt time, the same command can be prompted in any national languages.

The message ID specified for the PROMPT or CHOICE parameter on a CMD, PARM, QUAL, or ELEM command definition statement must be found in the prompt message file both when the command is being created and when the command is being prompted.

If an error in locating the message file occurs at the time the command is prompted, all prompt text is retrieved from the static copies of prompt messages stored in the *CMD object. If the message file is found, but an individual prompt text is not found in the message file, the static copy of the prompt text stored in the *CMD object is used for the message.

Related tasks

Defining message descriptions
Predefined messages are stored in a message file.

Defining a CL command
To create a command, you must first define the command through command definition statements.

Related information

Create Command (CRTCMD) command

Describing objects

Whenever you create an object, you can describe the object in a 50-character field on the TEXT parameter of a 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

The **Display Object Description (DSPOBJD)** or **Work with Objects (WRKOBJ)** command displays 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 28. Attributes displayed for object descriptions

Basic attributes	Full attributes	Service attributes (see Notes)
• Object name • Library name • Library ASP device • Library ASP group device • Object type • Extended attribute • Object size • Text description (partial)	• Object name • Library name • Library ASP device • Object type • Owner • Library ASP group device • 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 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 • Object ASP group device • Journaling status • Current or last journal • Journal images • Journal entries omitted • Remote journal filter • Journal start date and time • Starting journal receiver for apply • Library name for starting journal • Library ASP device for starting journal • Library ASP group device for starting journal • Save operation date and time • Restore operation date and time • Save command • Device type • Sequence number • File label ID • Save format	• Object name • Library name • Library ASP device • Library ASP group 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 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 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 (QLIRLIBD)` API or the command DSPLIB OUTPUT(*PRINT) to find the total size of the library.

Using the **Display Object Description (DSPOBJD)** or **Work with Objects (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 **Display Object Description (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
                                       Library ASP group. . :    *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         Library ASP group. . :   *SYSBAS
Type . . . . . :   *FILE           Primary group  . . . :   *NONE

User-defined information:
  Attribute  . . . . . . . . . . . . :
  Text . . . . . . . . . . . . . . . :

Creation information:
  Creation date/time . . . . . :   06/08/05  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         Library ASP group. . :   *SYSBAS
Type . . . . . :   *FILE           Primary group  . . . :   *NONE

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         Library ASP group. . :   *SYSBAS
Type . . . . . :   *FILE           Primary group  . . . :   *NONE

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       Library ASP group. . :   *SYSBAS
Type . . . . . :   *FILE           Primary group  . . . :   *NONE

Save/Restore information:
  Save date/time . . . . . :
  Restore date/time  . . . . . :
  Save command . . . . . :
  Device type  . . . . . :
                                                                       Bottom
Press Enter to continue.

F3=Exit   F12=Cancel
```

Related concepts

Object types and common attributes

Each type of object on the system has a unique purpose within the system, and each object type has a common set of attributes that describes the object.

Retrieving object descriptions

The **Retrieve Object Description (RTVOBJD)** command returns the descriptions of a specific object to a CL program or procedure.

Variables are used to return the descriptions. You can use these descriptions to help you detect unused objects on the system.

You can also use the Retrieve Object Description (QUSROBJD) API to return the description of a specific object to a program or procedure. The system uses a variable to return the descriptions.

The **Retrieve Object Description (RTVOBJD)** 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
- Object ASP group device
- Library ASP device
- Library ASP group device
- Indication of whether 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 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 primary associated space of object in bytes
- Indication of whether the space associated with the object has been optimally aligned
- 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 the object can be changed by the `Change Object Description (QLICOBJDD) API`
- Indication of whether the object has been modified with the `Change Object Description (QLICOBJD) API`
- Information about whether 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 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
- Remote journal filter
- The date and time that journaling was last started

- Name of the oldest journal receiver needed to successfully use the **Apply Journaled Changes (APYJRNCHG)** or **Remove Journaled Changes (RMVJRNCHG)** command
- Name of the library that contains the starting journal receiver
- Name of the auxiliary storage pool (ASP) device where storage is allocated for the library that contains the starting journal receiver
- Name of the auxiliary storage pool (ASP) group where storage is allocated for the library that contains the starting journal receiver

Related information

Retrieve Object Description (QUSROBJD) API

Retrieve Object Description (RTVOBJD) command

Example: Using the Retrieve Object Description command

In this sample CL source, the **Retrieve Object Description (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(7)
DCL  &USECNT   TYPE(*DEC)  LEN(5 0)
DCL  &RESET    TYPE(*CHAR) LEN(7)
.
.
.
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

Information about the creator of an object and the system on which it is created is saved in the object description when an object is created.

This information 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 is created

  - When an object is restored, the system on which the object will be created is the system that the object on the media was originally created on.
  - For IBM-supplied objects, the system created-on field is `00000000`.
  - For objects that already existed on the system before Version 1 Release 3.0, the system created-on field 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 **Create Duplicate Object (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.
  - Objects that are deleted and re-created during the rename operation lose the date of last use.
- Date of last activity
  - Date of last activity is the last date for which data transfer, session or conversation establishment, or use of the hardware associated with a device description occurred.
  - Because the *date of last use* is updated when a device, controller status, or both go beyond vary-on pending, the date of last use does not accurately reflect the use of the configuration description. For example, if you set virtual devices, controllers, or both to ONLINE(*YES), although no data transfer or communications are established on the device, the date of last use is updated when an initial program load (IPL) is processing and the device is varied on. However, *date of last activity* is not updated, which can reflect the accurate use of the configuration description.

Table 29. 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.
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`

Table 29. Updating usage information (continued)

Type of object	Commands and operations
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
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

Table 29. Updating usage information (continued)

Type of object	Commands and operations
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
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
SQL XML schema repository	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 29. Updating usage information (continued)

Type of object	Commands and operations
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 IBM i operating system 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 Retrieve Object Description (QUSROBJD) API provides the same information as the Retrieve Object Description command.

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 Configuration (*NWSCFG)
- 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)

Related information
CL command finder

Moving objects from one library to another

The **Move Object (MOVOBJ)** command moves 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.

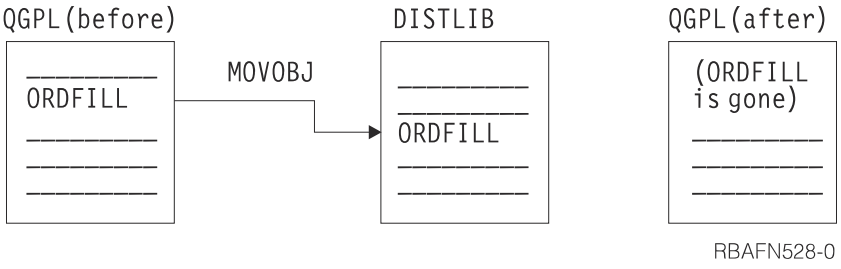
You can move an object out of the temporary library, QTEMP, but you cannot move an object into QTEMP. Also, you cannot move an output queue unless it is empty.

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)
- Network Server Configuration (*NWSCFG)
- Structured Query Language (SQL) packages (*SQLPKG)
- Structured Query Language (SQL) XML schema repository (*SQLXSR)
- 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 **Move Object (MOVOBJ)** command moves only one object at a time.

Related tasks

Renaming objects

The **Rename Object (RNMOBJ)** command renames objects. 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.

Related information

Move Object (MOVOBJ) command

Creating duplicate objects

The **Create Duplicate Object (CRTDUPOBJ)** command creates 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.

Note: 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 Configuration (*NWSCFG)
- Network server descriptions (*NWSD)
- Reference code translation tables (*RCT)
- Server storage space (*SVRSTG)
- Spelling aid dictionaries (*SPADCT)
- SQL packages (*SQLPKG)
- SQL XML schema repository (*SQLXSR)
- 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 **Create Duplicate Object (CRTDUPOBJ)** command with a **Copy File (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.

Related information

Create Duplicate Object (CRTDUPOBJ) command

Renaming objects

The **Rename Object (RNMOBJ)** command renames objects. 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 Configuration (*NWSCFG)
- Network Server Description (*NWSD)
- SQL packages (*SQLPKG)
- SQL XML schema repository (*SQLXSR)

- 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 **Rename Object (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.

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.

Related reference

Moving objects from one library to another
The **Move Object (MOVOBJ)** command moves objects between libraries.

Related information

Rename Object (RNMOBJ) command

Object compression or decompression

The **Compress Object (CPROBJ)** command compresses selected objects in order to save disk space on the system. The **Decompress Object (DCPOBJ)** command decompresses 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 IBM i-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.

Related information

Compress Object (CPROBJ) command
Decompress Object (DCPOBJ) command
Retrieve Object Description (RTVOBJD) command
Display Object Description (DSPOBJD) command

Restrictions for 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 30. 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 IBM i 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 be decompressed (that is 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 (that is ENDSYS or ENDSBS *ALL was issued)
  - 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 before 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 the **Delete Object (DLTOBJ)** command, a delete (DLTxxx) command for that type of object, or 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 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 can specify the generic name with the **Display Object Description (DSPOBJD)** command to verify that the generic delete will delete only the objects you want to delete.

Related concepts

Generic object names searching
A generic search can be used to search for more than one object.

Related reference

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.

Related information

Allocate Object (ALCOBJ) command

Display Object Description (DSPOBJD) command

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.

The IBM i operating system 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.

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. 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.

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.

Related tasks

Defining message descriptions
Predefined messages are stored in a message file.

Messages
Messages are used to communicate between users and programs.

Related information

Allocate Object (ALCOBJ) command

Deallocate Object (DLCOBJ) command

Database Programming: Lock records

Lock states for objects

A *lock state* identifies the use of the object and whether it is shared.

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. 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 31. 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 32. 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			

Table 32. Valid lock states for specific object types (continued)					
Object type	***EXCL**	***EXCLRD**	***SHRUPD**	***SHRNUP**	***SHRRD**
Program	x	x			x
Subsystem description	x				

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.

Note: By using the code example, you agree to the terms of the "Code license and disclaimer information" on page 610.

```
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 (CRTF)** command to specify how long your program is to wait for that file before a time-out occurs.

Displaying the lock states for objects

The **Work with Object Locks (WRKOBJLCK)** command or the **Work with Job (WRKJOB)** command displays the lock states for objects.

The **Work with Object Locks (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 **Work with Object Locks (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 **Work with Object Locks (WRKOBJLCK)** command.

If you use the **Work with Job (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
```

Related information

Work with Object Locks (WRKOBJLCK) command

Work with Job (WRKJOB) command

Accessing objects in CL programs

To access an object from a CL program, the object must be in the specified library when the command that refers to it runs.

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 based only on their runtime location.

You can avoid this runtime 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 (CALL**) command. This creates the called program within the calling program immediately prior to the **Call (CALL** command:

```
PGM /*TEST*/
DCL...
.
.
.
MONMSG
.
.
.
CRTCLPGM PGM(QGPL/PAYROLL)
CALL PGM(QGPL/PAYROLL)
.
.
.
ENDPGM
```

Note that for create commands, such as **Create CL Program (CRTCLPGM)** or **Create Data Area (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.)

Related concepts

Parts of a CL source program

Although each source statement that is entered as part of a CL source program is actually a CL command, the source can be divided into the basic parts that are used in many typical CL source programs.

Accessing command definitions, files, and procedures

To access command definitions or files from a CL program, the command definitions or files must exist at creation time of the program and must exist when the command that refers to them runs.

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.

Related information

Change Command (CHGCMD) command

Accessing files

A file must exist when compiling a CL module or original program model (OPM) program that uses it.

The compiler accesses files when compiling a program module that has a **Declare File (DCLF)** command. 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.

Related tasks

Working with files in CL programs or procedures

Two types of files are supported in CL procedures and programs: display files and database files.

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 **Create Service Program (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 **Check Object (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 want.

In the following example, the **Monitor Message (MONMSG)** command checks only for the object-not-found escape message. For a list of all the messages which the **Check Object (CHKOBJ)** command may send see the online help information for the **Check Object (CHKOBJ)** command.

The **Check Object (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.

```
           CHKOBJ  OBJ(OELIB/PGMA)  OBJTYPE(*PGM)
           MONMSG  MSGID(CPF9801)  EXEC(GOTO NOTFOUND)
           CALL OELIB/PGMA
           .
           .
           .
NOTFOUND:  CALL FIX001 /*PGMA Not Found Routine*/
           ENDPGM
```

Related tasks

Defining message descriptions
Predefined messages are stored in a message file.

Messages
Messages are used to communicate between users and programs.

Related reference

Monitor Message command
The **Monitor Message (MONMSG)** command is used to monitor for escape, notify, or status messages that are sent to the call stack of the CL program or procedure in which the **MONMSG** command is used.

Related information

Check Object (CHKOBJ) command

Working with files in CL programs or 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 **Receive File (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.

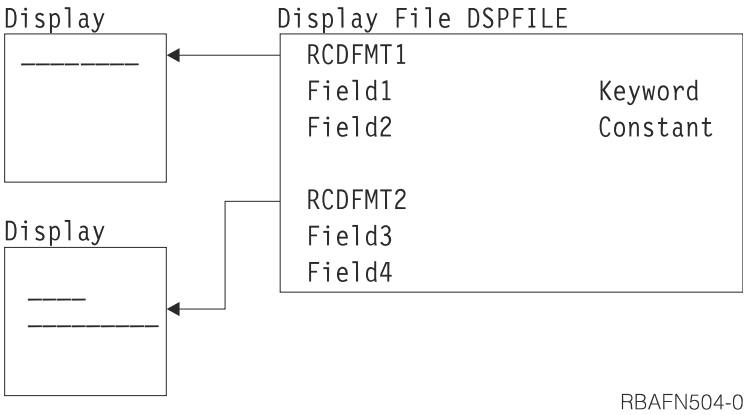
Up to five display or database files can be referred to in a CL procedure. The support for database files and display files is similar as the same commands are used. However, there are a few differences, which are described here.

- 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 **Receive File (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 **Receive File (RCVF)** command is processed. The file must exist and have a member at that time.
  - The file remains open until the procedure or original program model (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 high-level languages (HLLs) may require it, even though CL procedures and programs do not.



Related concepts

Parts of a CL source program

Although each source statement that is entered as part of a CL source program is actually a CL command, the source can be divided into the basic parts that are used in many typical CL source programs.

Related tasks

Accessing files

A file must exist when compiling a CL module or original program model (OPM) program that uses it.

Related information

CL command finder

Application Display Programming

Data manipulation commands

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.

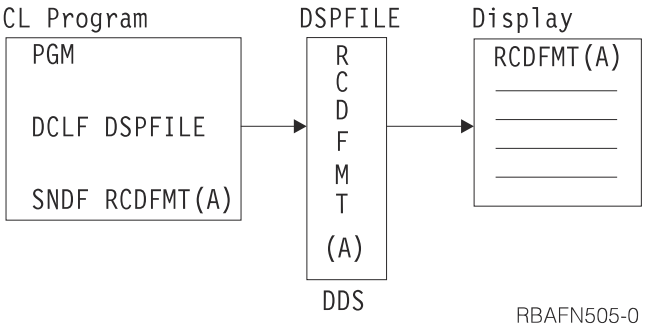
Receive File (RCVF)
Receives data from the display or database.

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 runtime override of a file used by a procedure or program with a display file.

Override with Database File (OVRDBF)
Allows a runtime 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 **Declare File (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 **Declare File (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 **Declare File (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.

Related information

CL command finder

Files in a CL program or procedure

Files are accessed during compiling of **Declare File (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.

Related information

Declare File (DCLF) command

Opening and closing files in a CL program or procedure

If you understand when and how files can be opened and closed, you can share open data paths between running programs or procedures.

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 original program model (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. After 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 **Override Database File (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 **Create Display File (CRTDSPF)**, **Create Physical File (CRTPF)**, and **Create Logical File (CRTLF)** commands online. 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 **Declare File (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 original program model (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 **Declare File (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 **Declare File (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
  .
  .
  .
```

Related information

Declare File (DCLF) command

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 **Send File (SNDF)**, **Receive File (RCVF)**, and **Send/Receive File (SNDRCVF)** commands.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 610.

The system formats the content of the variables associated with the output or output/input fields in the record format when you run a **Send File (SNDF)** command. Additionally the system sends it to the display device. This is similar to when you run a **Receive File (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 **Send/Receive File (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 **Send/Receive File (SNDRCVF)** or **Receive File (RCVF)** command for a workstation indicates WAIT(*NO), then the system uses the WAIT command to receive data. The same is true if a **Send File (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 runtime errors. Most other functions specified for display files in DDS are available; some functions (such as using variable starting line numbers) are not.

The following example shows the steps required to create a typical operator menu and to send and receive data using the **Send/Receive File (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 IBM i.

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 **Send/Receive File (SNDRCVF)** command defaults to WAIT(*YES); that is, the program waits until input is received by the program.

Related tasks

Messages
Messages are used to communicate between users and programs.

Related information

Send File (SNDF) command
Receive File (RCVF) command
Send/Receive File (SNDRCVF) command

Example: Writing a CL program or procedure to control a menu

This example shows how a CL program or procedure can be written to display and control a menu.

This example shows a CL procedure, ORD040C, that controls the displaying of the order department general menu and determines which high-level language (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.

Note: By using the code example, you agree to the terms of the “Code license and disclaimer information” on page 610.

```
       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 **Send/Receive File (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 **Send File (SNDF)**, **Receive File (RCVF)**, or **Send/Receive File (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 **Send/Receive File (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 (CALL)** command. See the Application Display Programming book for a discussion of those menus run using the **Go (GO)** command.

Related information

Send File (SNDF) command

Receive File (RCVF) command

Send/Receive File (SNDRCVF) command

Application Display Programming

Overriding display files in a CL procedure or program (OVRDSPF command)

The **Override with Display File (OVRDSPF)** command can be used to replace the display file named in a CL procedure or program or to change certain parameters of the existing display file. This can 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:

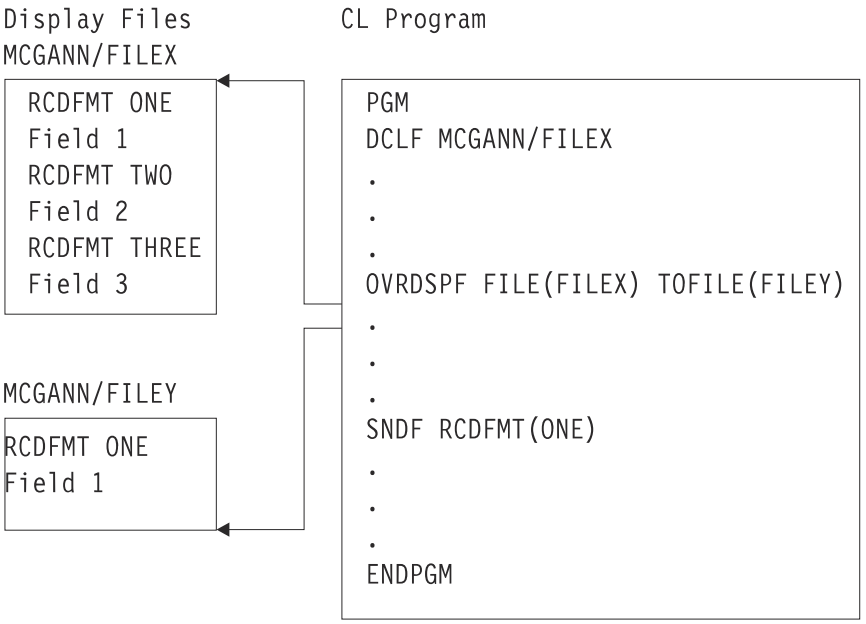
```
OVRDSPF   FILE(overridden-file-name) TOFILE(new-file-name)
          DEV(device-name)
```

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 **Send File (SNDF)**, **Receive File (RCVF)**, or **Send/Receive File (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.

Related information

Override with Display File (OVRDSPF) command

Send File (SNDF) command

Receive File (RCVF) command

Send/Receive File (SNDRCVF) command

Working with multiple device display files

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 program or procedure, the display file, or different record formats within it, can be sent to several device displays.

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 program or procedure, including the display file in the program or procedure. Each requester calls a separate job in this kind of use. This is not considered to be multiple device display use.

Commands used primarily with multiple device display files are as follows:

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 **Receive File (RCVF)** or SNDRCVF commands when WAIT(*NO) was specified on the command, or by one or more previous **Send File (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 **Send/Receive File (SNDRCVF)** and the RCVF commands and you may need to use the **Wait (WAIT)** or **End Receive (ENDRCV)** commands. When an **Receive File (RCVF)** or **Send/Receive File (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.

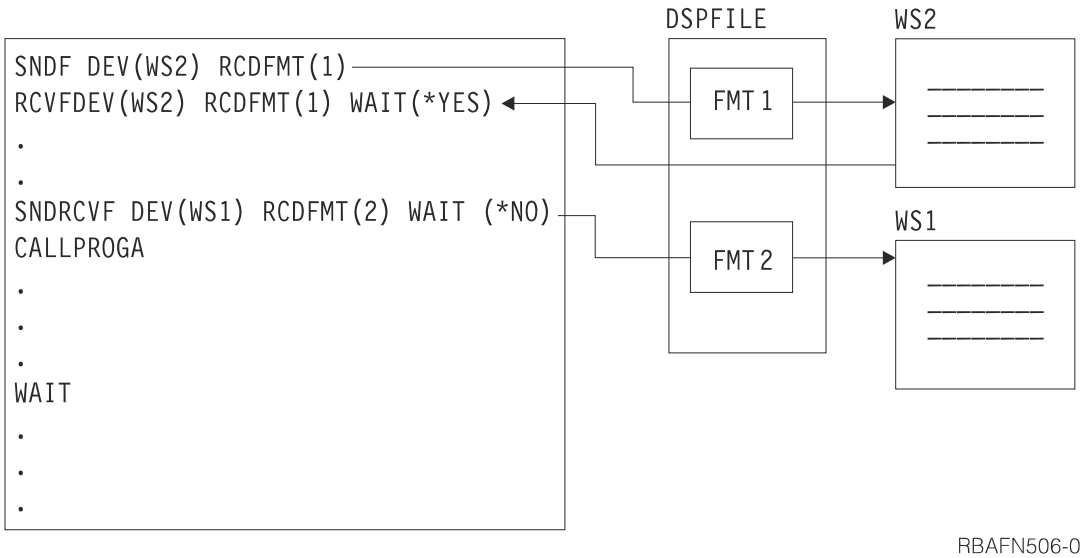
If you use an RCVF or **Send/Receive File (SNDRCVF)** command and specify WAIT(*NO), the CL procedure or program continues running until a **Wait (WAIT)** command is processed.

Using a **Send File (SNDF)** command with a record format which has the DDS INVITE keyword is equivalent to using a **Send/Receive File (SNDRCVF)** command with WAIT(*NO) specified. The DDS INVITE keyword is ignored for **Send/Receive File (SNDRCVF)** and **Receive File (RCVF)** commands.

The **Wait (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 (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.



In the previous 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 (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 **End Receive (ENDRCV)** command, which lets you cancel a request for input that has not yet been satisfied. A **Send File (SNDF)** or **Send/Receive File (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 **Send File (SNDF)** or **Send/Receive File (SNDRCVF)** command was processed, message CPF0887 is sent. In this case the data must be received with the WAIT command or **Receive File (RCVF)** command, or the request must be explicitly canceled with a **End Receive (ENDRCV)** command before the **Send File (SNDF)** or **Send/Receive File (SNDRCVF)** command can be processed.

Related concepts

Variables in CL commands
A *variable* is a named changeable value that can be accessed or changed by referring to its name.

Related information

Wait (WAIT) command
End Receive (ENDRCV) command
Send File (SNDF) command
Receive File (RCVF) command

Send/Receive File (SNDRCVF) command

Receiving data from a database file (RCVF command)

To receive data from a database file, use the **Receive File (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 original program model (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.

When the **RCVF** command reaches the end of the file, you can use the **Close Database File (CLOSE)** command to close the file. When the file is closed, the next **RCVF** command implicitly reopens the file and reads a record.

Related information

Receive File (RCVF) command

Close Database File (CLOSE) command

Reading a file multiple times (CLOSE command)

The **Close Database File (CLOSE)** command can be used to close a database file that is implicitly opened by the **Receive File (RCVF)** command.

You can use the **CLOSE** command to process the records in the database file more than once. When the file is closed, the next **RCVF** command implicitly reopens the file and reads a record. You can also use the **CLOSE** command for a database file that is already closed or was never opened; no error message is sent. When the CL program or procedure ends, all database and display files that are implicitly opened within the CL program or procedure were implicitly closed.

Related information

Receive File (RCVF) command

Close Database File (CLOSE) command

Overriding database files in a CL procedure or program (OVRDBF command)

The **Override with Database File (OVRDBF)** command overrides the database file named in a CL procedure or program, or changes certain parameters of the existing database file.

This can be especially useful for files that have been renamed or moved since the procedure or program is created. It can also be used to access a file member other than the first member.

The initial parameters of the OVRDBF command are as follows:

```
OVRDBF FILE(overridden-file-name) TOFILE(new-file-name)
       MBR(member-name)
```

The **Override with Database File (OVRDBF)** command is valid for a file referred to by a CL procedure or program only if the file specified in the **Declare File (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 **Override with Database File (OVRDBF)** command must be processed before the file to be overridden is opened for use (an open occurs by the first use of the **Receive File (RCVF)** command).

The file is overridden if it is opened in the procedure or original program model (OPM) program containing the **Override with Database File (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).

Related information

Override with Data Base File (OVRDBF) command

Output files from display commands

A number of the IBM display commands 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.

Note: By using the code example, you agree to the terms of the "Code license and disclaimer information" on page 610.

```
     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 **Display Object Description (DSPOBJD)** command. This file is the primary file that 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 **Override with Database File (OVRDBF)** command overrides the declared file (QADSPOBJ) to the file created by the **DSPOBJD** command.

(4)

The **Receive File (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 **Declare File (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 **Grant Object Authority (GRTOBJAUT)** command is processed, using the variables for object name, library name and object type, which were read by the **RCVF** command.

Messages

Messages are used to communicate between users and programs.

On the IBM i operating system, communication between procedures or programs, between jobs, between users, and between users and procedures or programs occurs through messages. 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 original program model (OPM) program or Integrated Language Environment (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

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      06/12/05   10:50:54
System going down at 11:00; please sign off
```

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

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.

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.

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.

Related concepts

Types of message queues
The system has different types of message queues: workstation message queue, user profile message queue, job message queue, system operator message queue, and history log message queue.

Related tasks

Allocating resources
Objects are allocated on the system to guarantee integrity and to promote the highest possible degree of concurrency.

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.

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 **Send File (SNDF)**, **Receive File (RCVF)**, and **Send/Receive File (SNDRCVF)** commands.

Related reference

Monitor Message command
The **Monitor Message (MONMSG)** command is used to monitor for escape, notify, or status messages that are sent to the call stack of the CL program or procedure in which the **MONMSG** command is used.

Related information

Add Message Description (ADDMSGD) command

Defining message descriptions

Predefined messages are stored in a message file.

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:

- 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 in a message queue (**Display Messages (DSPMSG)** and **Work with Messages (WRKMSG)**)
- Display message descriptions in a message file (**Display Message Description (DSPMSGD)** 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, but the status message will not appear in a job log. 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.

Related concepts

Dynamic prompt message for control language
By using the message identifiers that are stored in the command (*CMD) object when the command was created, prompt messages can be dynamically retrieved from the message file. This function enables a single command to have prompt messages in more than one national language.

Related tasks

Allocating resources
Objects are allocated on the system to guarantee integrity and to promote the highest possible degree of concurrency.

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.

Related reference

Monitor Message command
The **Monitor Message (MONMSG)** command is used to monitor for escape, notify, or status messages that are sent to the call stack of the CL program or procedure in which the **MONMSG** command is used.

Related information

CL command finder

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 **Create Message File (CRTMSGF)** command, you can specify the maximum size in kilobytes on the SIZE parameter. The following formula can be used to determine the maximum:

```
S + (I x N)
```

where the following variables are true:

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 default for S is 10, for I is 2, and for N is *NOMAX.

For example, you specify S as 5, I as 1, and N as 2. When the file reaches the initial storage amount of 5KB, the system automatically adds another 1KB to the initial storage. The amount added (1KB) can be added to the storage two times to make the total maximum of 7KB. If you specify *NOMAX as N, the maximum size of the message file is 16MB.

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. Because 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.

Related information
CL command finder
Add Message Description (ADDMSGD) command
Create Message File (CRTMSGF) command

Message files in independent ASPs

Message files can be created in an independent auxiliary storage pool (ASP), but this 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

To determine the size of a message file, you need to use this formula. The **Add Message Description (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 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 (DFTPGM, 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.

Table 33. The largest possible entry

Attribute	**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 **Create Message File (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 IBM i, the message file must be named QUSERMSG.

Related information

Add Message Description (ADDMSGD) command

Create Message File (CRTMSGF) command

Adding messages to a file

The **Add Message Description (ADDMSGD)** command describes your predefined messages and adds them to the message file that 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 Identifier (CCSID)

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 **Add Message Description (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, the following message is 1234 of CPF:

```
CPF1234
```

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 be careful when 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.

Related tasks

Monitoring for messages in a CL program or procedure

Messages that can be monitored are *ESCAPE, *STATUS, and *NOTIFY messages that are issued by each CL command used in the program or procedure.

Related information

Add Message Description (ADDMSGD) command

Defining messages and message help

You can define two levels of messages on the **Add Message Description (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 (typically 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.

Defining substitution variables

On the FMT parameter on the **Add Message Description (ADDMSGD)** command, you can specify substitution variables for either first-level 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 **Add Message Description (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 the data type and, optionally, the length on the **Add Message Description (ADDMSGD)** command. The valid data types for message data fields are:

- 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.

- Character string (*CHAR). A string of character data not to be enclosed in single quotation marks. 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 single quotation marks. 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 CCSID support for messages for more information about the use of message handlers with CCSIDs.
- Coordinated Universal Time date stamp (*UTCD). An 8-byte field that contains a system date time stamp in Coordinated Universal Time (UTC). The date in the output message is adjusted from UTC by the time zone specified for the job. The date is formatted by using the job definition attributes for date format QDATFMT and date separator QDATSEP. When the 8-byte field is passed as hexadecimal zero (X'0000000000000000'), the value is formatted as *N.
- Coordinated Universal Time date and time stamp (*UTC). An 8-byte field that contains a system date time stamp in UTC. The output-formatted date time stamp contains the date followed by one blank separator and the time. The date and time in the output message are adjusted from UTC by the time zone specified for the job. The date is formatted by using the job definition attributes for the date format, QDATFMT, and the date separator, QDATSEP. The time is formatted by using the job definition attribute for time separator QTIMSEP. When the 8-byte field is passed as hexadecimal zero (X'0000000000000000'), the value is formatted as *N.
- Coordinated Universal Time time stamp (*UTCT). An 8-byte field that contains a system date time stamp in UTC. The time in the output message is adjusted from UTC by the time zone specified for the job and is formatted by using the job definition attribute for the time separator, QTIMSEP. When the 8-byte field is passed as hexadecimal zero (X'0000000000000000'), the value is formatted as *N.
- 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.
- 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.
- Hexadecimal (*HEX). A string to be preceded by the character X and enclosed in single quotation marks; 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.
- Quoted character string (*QTDCHAR). A string of character data to be enclosed in single quotation marks. Preceding and trailing blanks are not deleted. If length is not specified in the message description, the sender determines the length of the field.
- Time interval (*ITV). An 8-byte time interval that contains the time to the nearest whole second for various wait timeout conditions.
- 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 assumes that the binary integer is 2.

The following data types are valid only in IBM-supplied message descriptions and should not be used for other messages:

- 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 **Add Message Description (ADDMSGD)** command.
- 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.

Assigning a severity code

The severity code you assign to a message on the **Add Message Description (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.

Related reference

SEV parameter
The severity (SEV) parameter specifies a severity code.

Related information

Add Message Description (ADDMSGD) command

Specifying validity checking for replies

On the **Add Message Description (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 things:

- 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.

Example: Sending an immediate message and handling a reply

This example shows how a program or procedure sends an inquiry message and handles the reply.

In this example, the procedure completes the following tasks:

- Sending an immediate inquiry message to QSYSOPR
- Requesting a reply of yes or no (Y or N)
- Ensuring that a valid reply has been entered
- Doing a timeout 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 timeout 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 program or procedure logic checks for an uppercase or lowercase value of Y or N. Normally, the value is entered by the operator as a lowercase value. If the operator enters a nonblank value other than Y or N, the program or procedure sends a different message and then repeats the inquiry message.

If the operator enters a blank, no reply is sent to the program or procedure. If a blank is returned to the program or procedure, a timeout occurs (the operator does not reply). The program or procedure sends a message to the system operator, stating that a reply was not received and the default is 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 can be answered or has already timed out. The program or procedure does not remove a message from the message queue after it has been sent. The second message minimizes this concern and provides an audit trail for what has occurred.

If the timeout 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.

Related tasks

Using a sender copy message to obtain a reply
When an inquiry message is sent, it expects a reply. To allow the sender of an inquiry message to obtain a reply, a sender copy message is issued and associated internally with the inquiry message.

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 **Add Message Description (ADDMSGD)** command specifies 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 the *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.
  - 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 items:
  - 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 list 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.

Related tasks

CL handling for unmonitored messages
The system provides default monitoring and handling of any messages you do not monitor.

Related information

Message Handling APIs
Exit programs

Example: Sending the last diagnostic message as an escape message

This example is a program that sends the last diagnostic message as an escape message.

The following program is an example default program that could be used when a diagnostic message is sent followed by an escape message. This program could be an original program model (OPM) CL program or an Integrated Language Environment (ILE) program that has this single CL procedure.

Note: By using the code example, you agree to the terms of the "Code license and disclaimer information" on page 610.

```
            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 **Add Message Description (ADDMSGD)** command, you can specify an alert option to allow an alert to be created for a message.

A message with an alert option specified 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.

Related information
DSNX Support PDF

Example: Describing a message

This example shows how to create a message to be used in an application.

In the following example, the **Add Message Description (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 **Add Message Description (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, you can write a CL procedure or program using the **Add Message Description (ADDMSGD)** command. The defined message is put into a message file and then sent normally.

When writing the program, follow these steps:

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 **Monitor Message (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.

Viewing messages

The **Work with Message Descriptions (WRKMSGD)** or **Display Message Description (DSPMSGD)** command views or prints single messages or a range of message descriptions in a message file.

Here is an example of using the **Display Message Description (DSPMSGD)** command:

```
DSPMSGD   RANGE(*FIRST  IDU0571)  MSGF(QIDU/QIDUMSG)
          FMTTXT(*NO)  OUTPUT(*PRINT)
```

To see the exception messages which can be generated with a command, see the description of the command.

Message file searching

The system searches for the message file from which the message description is retrieved.

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.
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.

System message file searches

The message file name and library specified at the time the message file is sent are used to search for the message file from which the message description is retrieved.

This is true when a message file has not been overridden and when a message file name is overridden but the message identifier is not contained in the overridden 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 because of damage or an authorization problem.

Overriding message files

The creation (Override Message File command), deletion (Delete Override command), and display (Display Override command) of message file override operations is similar to other types of override operations. With these commands, however, only the name of the message file, not the attributes, is overridden, and the rules for applying the override operations 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 can be used:

```
OVRMSGF MSGF(QCPFMSG) TOMSGF(USRMSGF)
```

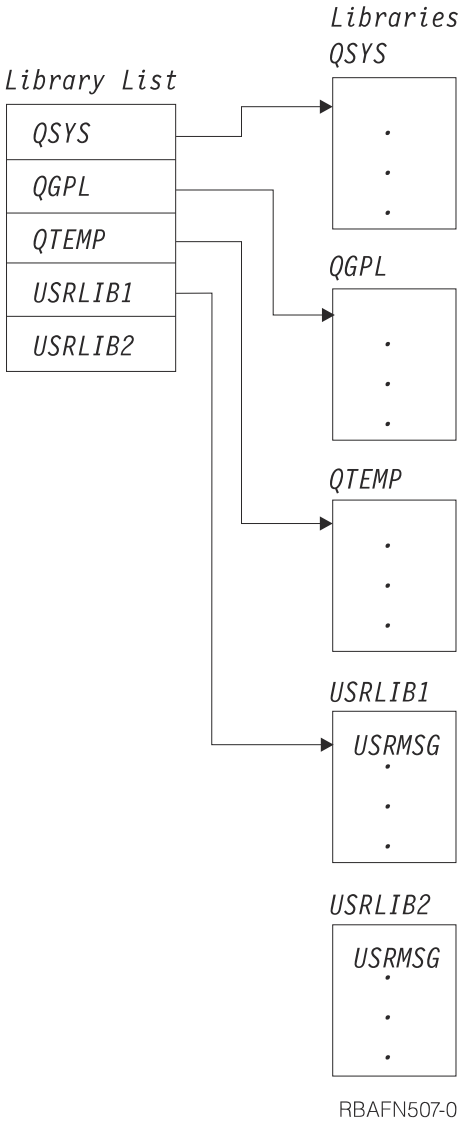
When a predefined message is retrieved or displayed, the overriding file is searched for a message description. If the message description is not found in that file, the overridden file is searched.

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: Overriding a message file
This example shows how to change a message used in a job.

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:

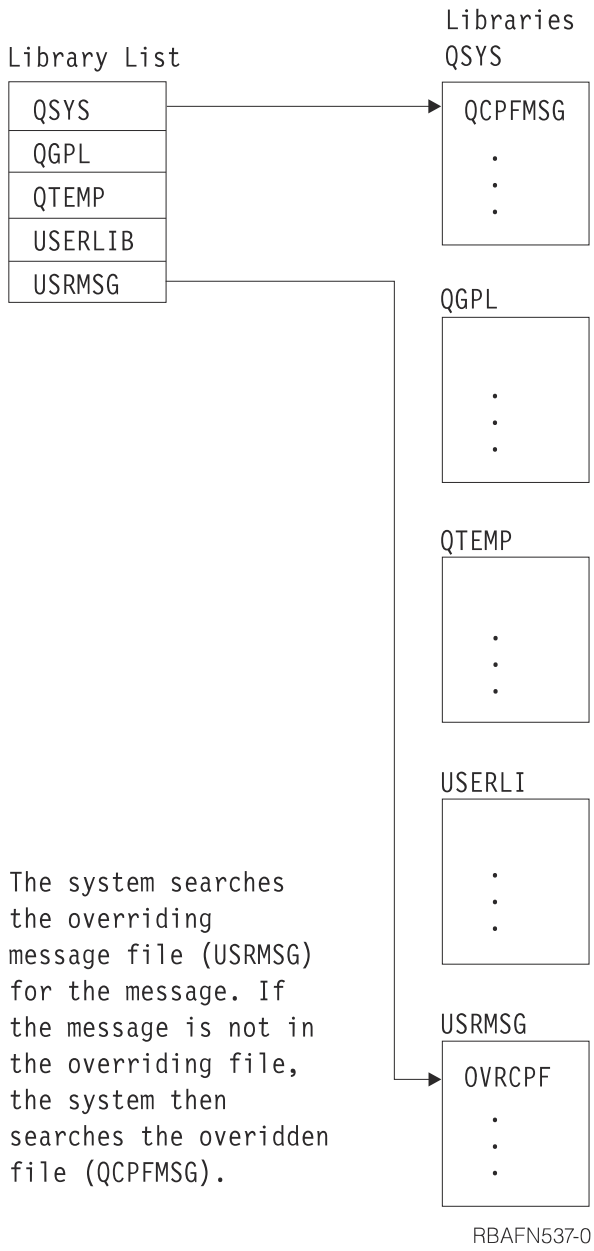
```
CRTMSGF MSGF(USRMSG/OVRCPF)
```

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 **Override Message File (OVRMSGF)** command to override the message file when you run the job:

```
OVRMSGF MSGF(QCPFMSG) TOMSGF(USRMSG/OVRCPF)
```



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 **Override Message File (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.

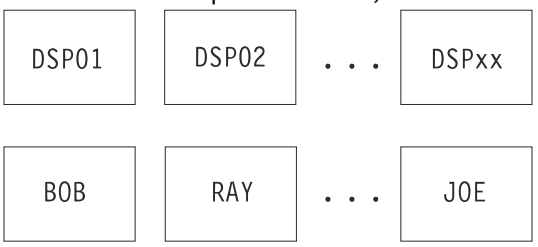
Message queues

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 that requested the reply.

Types of message queues

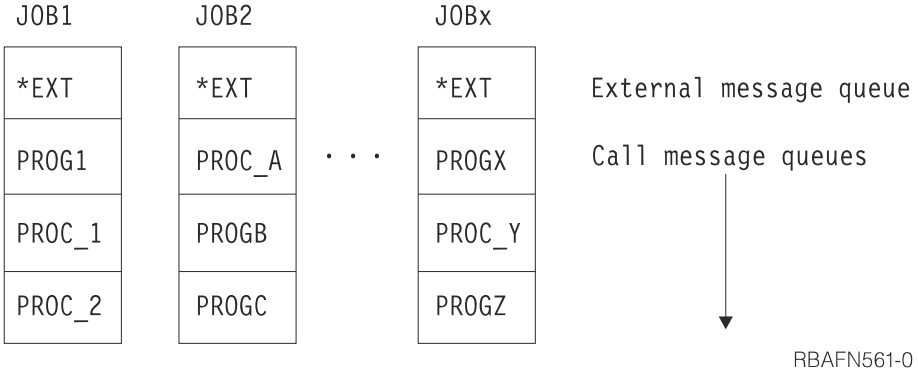
The system has different types of message queues: workstation message queue, user profile message queue, job message queue, system operator message queue, and history log message queue.

The following figures 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).



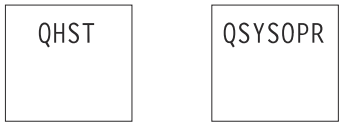
RBAFN560-0

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 original program model (OPM) program or Integrated Language Environment (ILE) procedure within the job has its own call message queue.



RBAFN561-0

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.
- 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 any job in the system to have a record of high-level system activities.

In addition to these message queues, you can create your own user message queues for sending messages to system users and between application programs.

Related concepts

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.

Related tasks

Messages
Messages are used to communicate between users and programs.

Creating or changing a message queue

To create your own user message queues, use the **Create Message Queue (CRTMSGQ)** command. In addition, you can use the **Change Message Queue (CHGMSGQ)** command to change attributes of your message queue. To view the contents of the message queue, use the **Display Messages (DSPMSG)** or **Work with Messages (WRKMSG)** command.

The attributes of a message queue are as follows:

- 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 **DSPMSG** command automatically displays the message. Break messages for a job can be controlled with the BRKMSG parameter on the **Change Job (CHGJOB)** command.
  - Notify delivery. A display 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** or **WRKMSG** command.
  - Hold delivery. The message queue holds the messages until the display station user requests them with the **DSPMSG** or **WRKMSG** 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.
  - 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.

  You cannot change this attribute for message queue QHST; QHST sends CPF2460 when it is full. IBM ships QSYSOPR with this attribute set to wrap.

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 a secondary 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 a secondary 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 secondary 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 previous situation can occur for a system operator as well.

Related information

Create Message Queue (CRTMSGQ) command
Change Message Queue (CHGMSGQ) command
Display Message (DSPMSG) command
Work with Message (WRKMSG) command

Message queues in independent ASPs

IBM suggests that message queues in independent auxiliary storage pools (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-handling program will not be able to change the library name space for the thread.

Related information

Independent disk pool examples

Message queues in break mode

A break-handling program can be 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 **Change Message Queue (CHGMSGQ)** command. The break-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. There is an IBM-supplied break-handling program that you can use by default on the **CHGMSGQ** command. For example:

```
CHGMSGQ MSGQ(name) DLVRY(*break)
```

Note: This program cannot open a display file if the interrupted program is waiting for input data from the device display.

Related tasks

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.

Placing a message queue in break mode automatically

By placing a message queue in break mode automatically, you can monitor the QSYSOPR message queue.

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.

Note: By using the code example, you agree to the terms of the "Code license and disclaimer information" on page 610.

```
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 that another user has QSYSOPR in break mode already.

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.

Related tasks

Using the system reply list
By using the system reply list, you can specify that the system automatically issues the reply to specified predefined inquiry messages.

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 Integrated Language Environment (ILE) procedure and original program model (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.

Related concepts

Types of message queues
The system has different types of message queues: workstation message queue, user profile message queue, job message queue, system operator message queue, and history log message queue.

External message queue

The external message queue (*EXT) is used to communicate with the external requester (such as a display station user) of the job.

Messages sent to *EXT are displayed in the following ways:

- Display Program Messages screen

  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 program or procedure waits for a reply to inquiry or notify messages from the display station user. If the user does 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.

- Message line of the display station

  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
```

Messages (except status messages) sent to the external message queue of a job are also placed on the job log.

Related concepts

Job log
Each job has an associated job log.

Status messages
To send status messages from your CL procedure or program to the external message queue (*EXT) for the job or to a call message queue, use the **Send Program Message (SNDPGMMSG)** command.

Call message queue

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. After the program or procedure returns, 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) 4096 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.

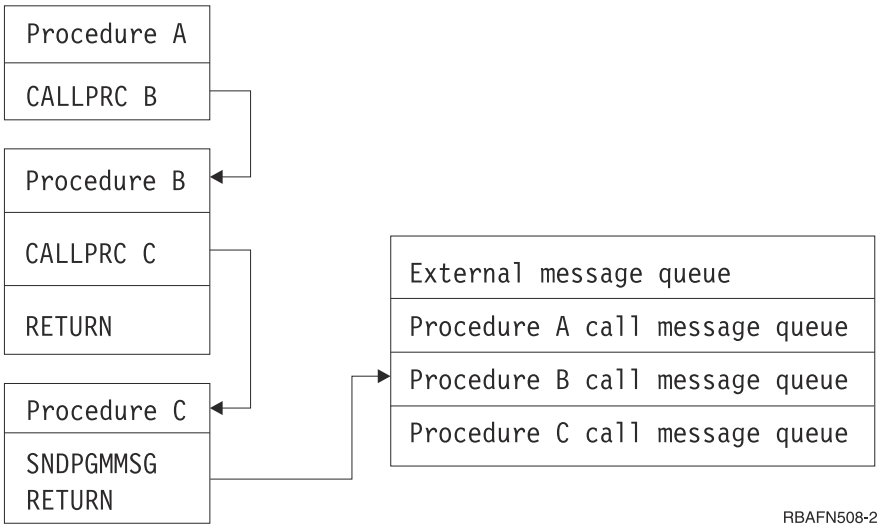
When identifying the call stack entry for an ILE procedure, it is sufficient to specify only the procedure name. If the procedure name by itself does not uniquely identify the call stack entry, the module name or the ILE program name can also be specified. If, at the time a message is sent, a program 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.

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 resolve 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 than 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.

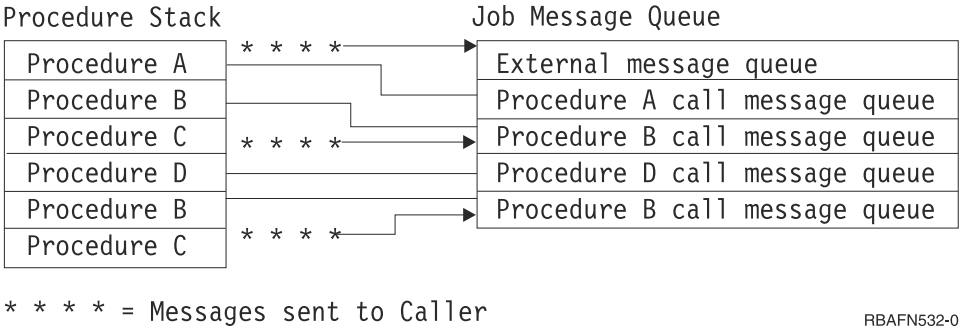
For example, assume that procedure A calls procedure B which calls procedure C. Procedure C sends a message to procedure B and ends. The message is available to procedure B. However, when procedure B ends, its call message queue is no longer available. As a result, you cannot access procedure B by using 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.



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 move or 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. Procedure A sends a message to itself and to *EXT, and then calls procedure B. Procedure B calls Procedure C. Procedure C sends a message to its caller (Procedure B) and then calls Procedure D. Procedure D sends a message to itself and calls Procedure B. A connecting line (-----) indicates which message queue is associated with which call of a procedure.



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). After 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.

Related tasks

Receiving a message from a program or procedure that has ended
Occasionally, there is a need to receive messages from the job log that are sent to a call message queue that is no longer active.

Identifying a call stack entry
If a CL procedure is to send a message to an original program model (OPM) program or another Integrated Language Environment (ILE) procedure, you must identify the call stack entry to which the message is sent.

Commands used to send 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)**

Send Program Message (SNDPGMMSG) and **Send User Message (SNDUSRMSG)** can only be used in batch or interactive original program model (OPM) programs or Integrated Language Environment (ILE) procedures. These commands cannot be entered on a command line. The **Send Message (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 **Send Message (SNDMSG)** command is sent by a display station user to the system operator:

```
SNDMSG   MSG('Mount tape on device TAP1') TOUSR(*SYSOPR)
```

The **Send Break Message (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 **Send Break Message (SNDBRKMSG)** command

when sending any message that requires the immediate attention of a display station user. You cannot ensure the 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 **Send Break Message (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.

Related reference

Commands used to send messages from a CL program
The **Send Program Message (SNDPGMMSG)** command or the **Send User Message (SNDUSRMSG)** command is used to send a message from a CL procedure or program.

Commands used to send messages from a CL program

The **Send Program Message (SNDPGMMSG)** command or the **Send User Message (SNDUSRMSG)** command is used 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
- Call message queue of a program or procedure called by the job
- 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 34. 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: Describing a message" on page 508, 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 (TOMSGQ parameter); therefore, you specify the external message queue *EXT on the TOPGMQ parameter.

Related concepts

Identifying the base entry by name
To identify the base call stack entry, provide the name of the original program model (OPM) program or Integrated Language Environment (ILE) procedure that runs in that entry.

Related tasks

Receiving request messages
Receiving request messages is a method for your CL procedure or program to process CL commands.

Related reference

Commands used to send messages to a system user

Several commands can be used to send messages to system users.

Related information

Types of message queues

Inquiry and informational messages

To send an inquiry message or an informational message to a display station user, to the system operator, or to a user-defined message queue, use the **Send User Message (SNDUSRMSG)** command.

If you use the **Send User Message (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 **Send User Message (SNDUSRMSG)** command, you can specify the following on the **Send User Message (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

To send diagnostic and completion messages, use the **Send User Message (SNDUSRMSG)** command.

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 typically not sent because the requested function was performed.

For an example of sending a completion message, assume that the system operator uses the command entry display 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 completion message.

Note: By using the code example, you agree to the terms of the “Code license and disclaimer information” on page 610.

```
PGM
SAVOBJ OBJ(PAY1 PAY2) LIB(PAYROLL) DEV(TAP01)
SNDPGMMSG MSG('Payroll objects have been saved') MSGTYPE(*COMP)
ENDPGM
```

If the **Save Object (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. If the SAVOBJ command completes successfully, the completion message is sent to the call message queue associated with the program that displays the command entry display.

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

To send status messages from your CL procedure or program to the external message queue (*EXT) for the job or to a call message queue, use the **Send Program Message (SNDPGMMSG)** command.

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.

Status messages are not included in the job log for a job.

Related concepts

External message queue
The external message queue (*EXT) is used to communicate with the external requester (such as a display station user) of the job.

Escape and notify messages

You can send both escape and notify messages from your CL program or procedure to message queues.

You can send escape messages from your CL program or procedure to the call message queue of the calling program or procedure with the **Send Program Message (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, the default system action is taken.

You can send notify messages from a CL program or procedure 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 original program model (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.

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)
```

Related tasks

Monitoring for messages in a CL program or procedure
Messages that can be monitored are *ESCAPE, *STATUS, and *NOTIFY messages that are issued by each CL command used in the program or procedure.

Examples: Sending messages

These examples show how to send different kinds of messages.

Example 1: Sending a completion message

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 610.

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: Sending a completion message with variable text

The following CL procedure sends a message based on a parameter received from a program that is called from within this procedure. The message is 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: Sending an inquiry message and receive its reply

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 contain the remainder of the code that would use the reply value.

```
PGM
DCL SNDRCOPY TYPE(*CHAR) LEN(4)
SNDPGMMSG   MSG('Load special form') TOUSR(*SYSOPR) +
            KEYVAR(SNDRCOPY) MSGTYPE(*INQ)
RCVMSG MSGTYPE(*RPY) MSGKEY(SNDRCOPY) WAIT(120)
.
.
.
ENDPGM
```

Notes:

1. The WAIT parameter must be specified on the **Receive Message (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 **Receive Message (RCVMSG)** command, without receiving the reply.
2. The variable SNDRCOPY in the **Send Program Message (SNDPGMMSG)** command returns to the procedure the message key of the sender copy message associated with the inquiry message. This is needed to get the reply. When this key is specified on a **Receive Message (RCVMSG)**

command with a message type of *RPY, the reply that was entered for the associated inquiry message is returned. The SNDRCOPY message key is specified in the MSGKEY parameter of the **Receive Message (RCVMSG)** command.

Example 4: Sending an inquiry message and receive a reply with SNDUSRMSG

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 (the default of the TRNTBL parameter of the **Send User Message (SNDUSRMSG)** command) to make program logic easier.) If the value entered is not one of these four, the operator is issued a message from **Send User Message (SNDUSRMSG)** 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: Sending an escape message

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: Sending an informational message to multiple users

The following procedure allows the system operator to send a message to several display stations. When the system operator calls the program containing this procedure, it 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 /* Change from military time */
            CHGVAR &WORKHR &HOUR
            CHGVAR &WORKHR (&WORKHR - 12)
            CHGVAR &HOUR &WORKHR
            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
```

Related tasks

Using a sender copy message to obtain a reply

When an inquiry message is sent, it expects a reply. To allow the sender of an inquiry message to obtain a reply, a sender copy message is issued and associated internally with the inquiry message.

Identifying a call stack entry

If a CL procedure is to send a message to an original program model (OPM) program or another Integrated Language Environment (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 **Send Program Message (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

This specification identifies a program or procedure within the call stack (this is element 2 of TOPGMQ).

- 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).

The specification TOPGMQ(*PRV *) identifies the base entry as being the one in which the procedure using the **Send Program Message (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.

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. Because program CLPGM2 is ILE, it can consist of several procedures, such as: CLPROC1, CLPROC2, CLPROC3, and CLPROC4. At run time, the following calls take place:

- CLPGM1 is called first.
- CLPGM1 calls CLPGM2.
- CLPGM2 calls CLPROC1.
- CLPROC1 calls CLPROC2.
- CLPROC2 calls CLPROC3 or CLPROC4.

The following figure illustrates the considerations about the structure of the call stack when CLPROC2 calls CLPROC4:

- 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.

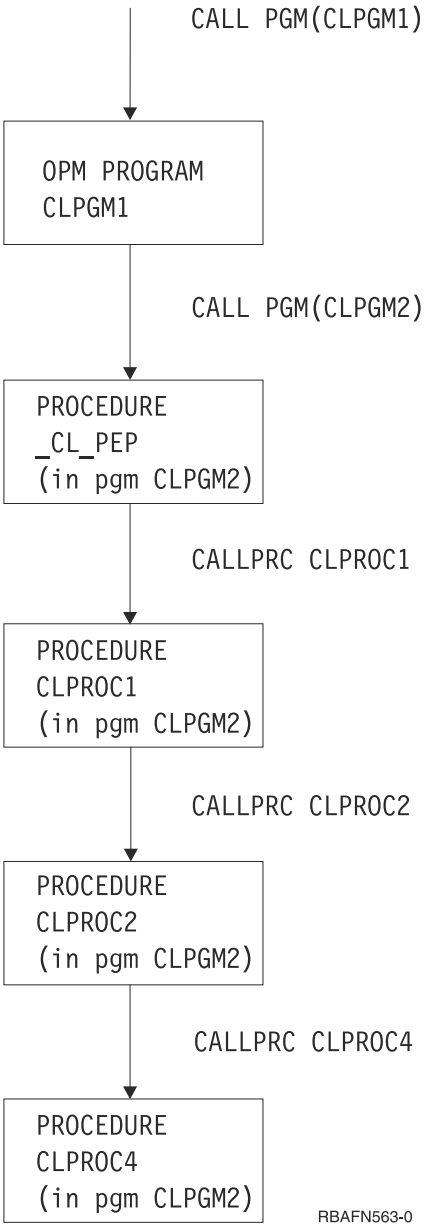


Figure 16. Example of runtime call stack with an OPM program and an ILE program with multiple procedures

Related concepts

Call message queue

A call message queue is used to send messages between one program or procedure and another program or procedure.

Using the Send Program Message command as the base

To use the program or procedure that runs the **Send Program Message (SNDPGMMSG)** command as the base call stack entry, use the TOPGMQ parameter.

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 program or procedure will send a message to its 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 CLPGM1 which is the caller of the CLPGM2, 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.

The following figure illustrates the results when CLPROC1, CLPROC2, and CLPROC4 each sends a message previous of their procedure.

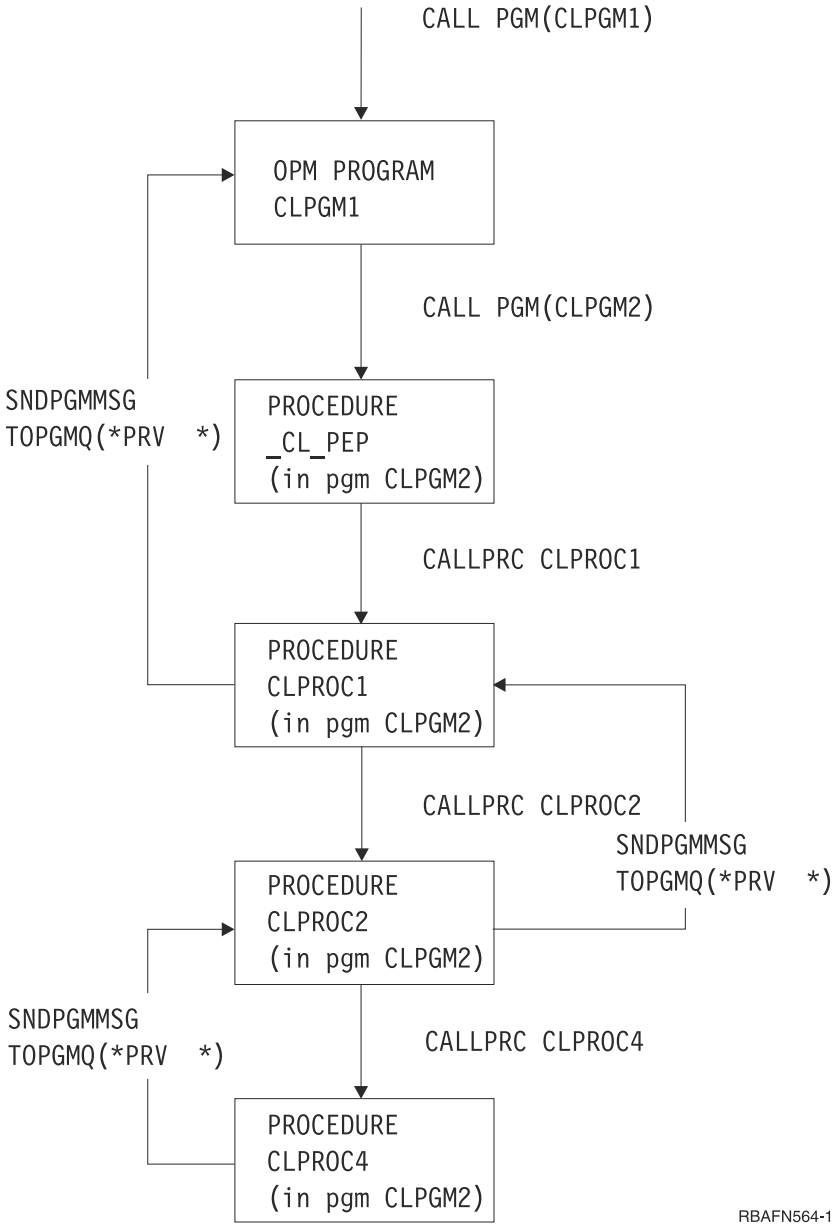


*Figure 17. Example of TOPGMQ(*PRV *)*

Note: The PEP is not visible to (*PRV *), so the message from CLPROC1 will be sent to CLPGM1.

Identifying the base entry by name

To identify the base call stack entry, provide the name of the original program model (OPM) program or Integrated Language Environment (ILE) procedure that runs 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 18 on page 533 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 19 on page 534 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 can use partial names. The online command help text provides information about how to specify partial call stack names. For example, see the Send Program Message (SNDPGMMSG) command.

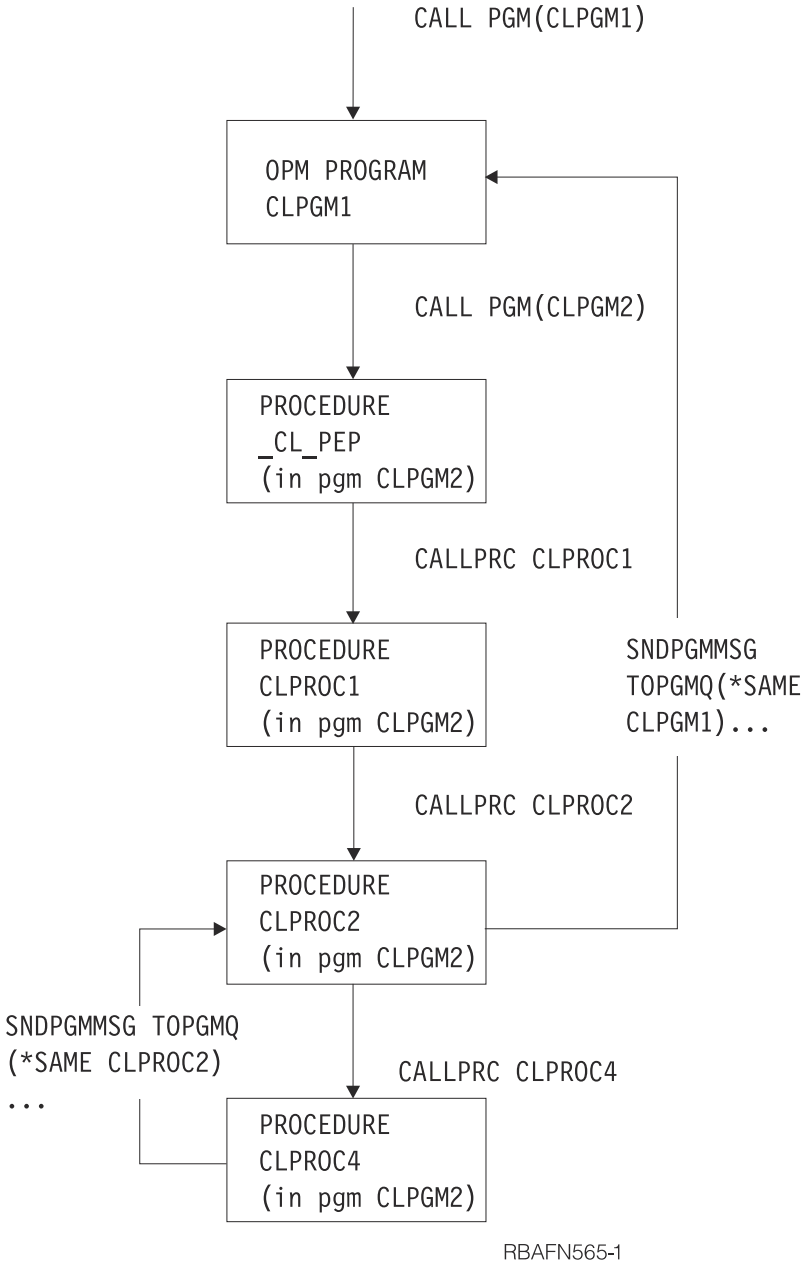


Figure 18. Example of using a simple name as a base

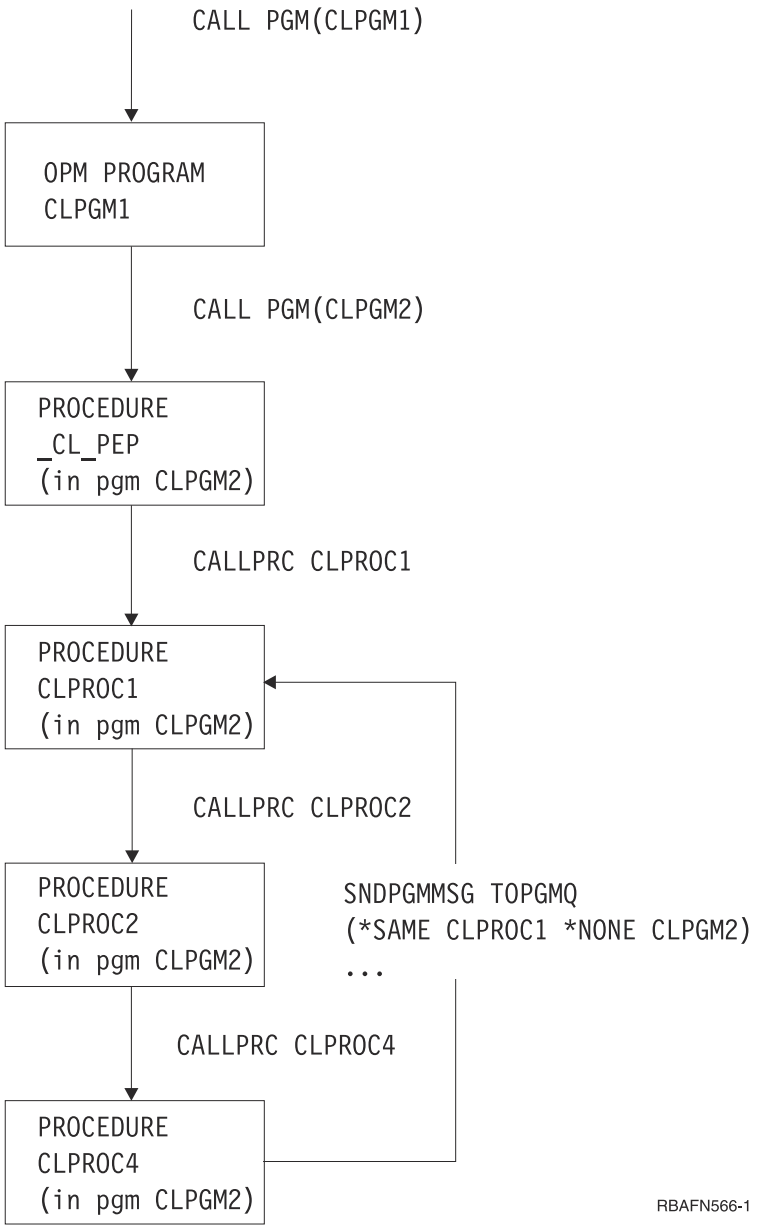


Figure 19. Example of using a complex name as a base

Related reference

Commands used to send messages from a CL program

The **Send Program Message (SNDPGMMSG)** command or the **Send User Message (SNDUSRMSG)** command is used to send a message from a CL procedure or program.

Related information

Send Program Message (SNDPGMMSG) command

*Using the program boundary as a base (*PGMBDY)*

To specify the program boundary as the base call stack entry, use the *PGMBDY special value.

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 the first figure for an example of sending a message using the special value *PGMBDY. In this example, CLPROC4 is sending a message directly 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 the second figure for the example of sending a message using the special value *PGMBDY and a program name. The following programs and procedures are used in the second figure:

- CLPGM1 and CLPGM2. These are defined as in the previous examples.
- CLPGM3. This is another Integrated Language Environment (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.

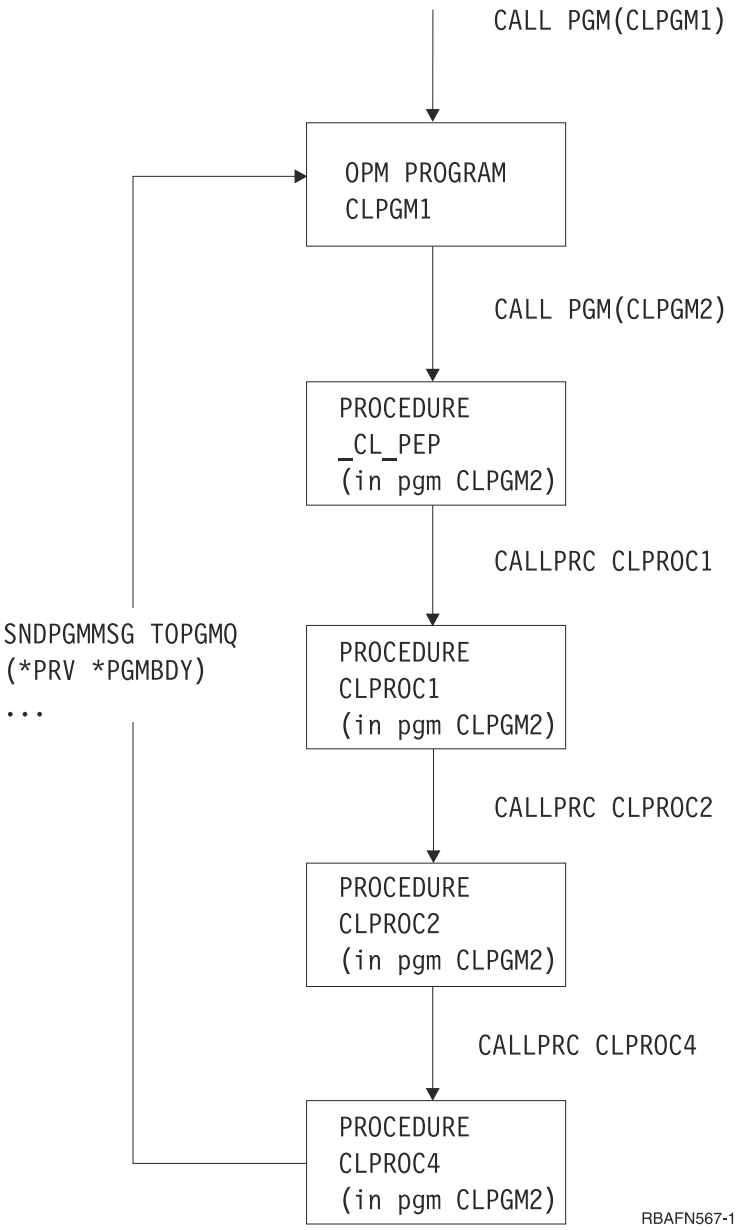


*Figure 20. Example of using *PGMBDY as a simple name*

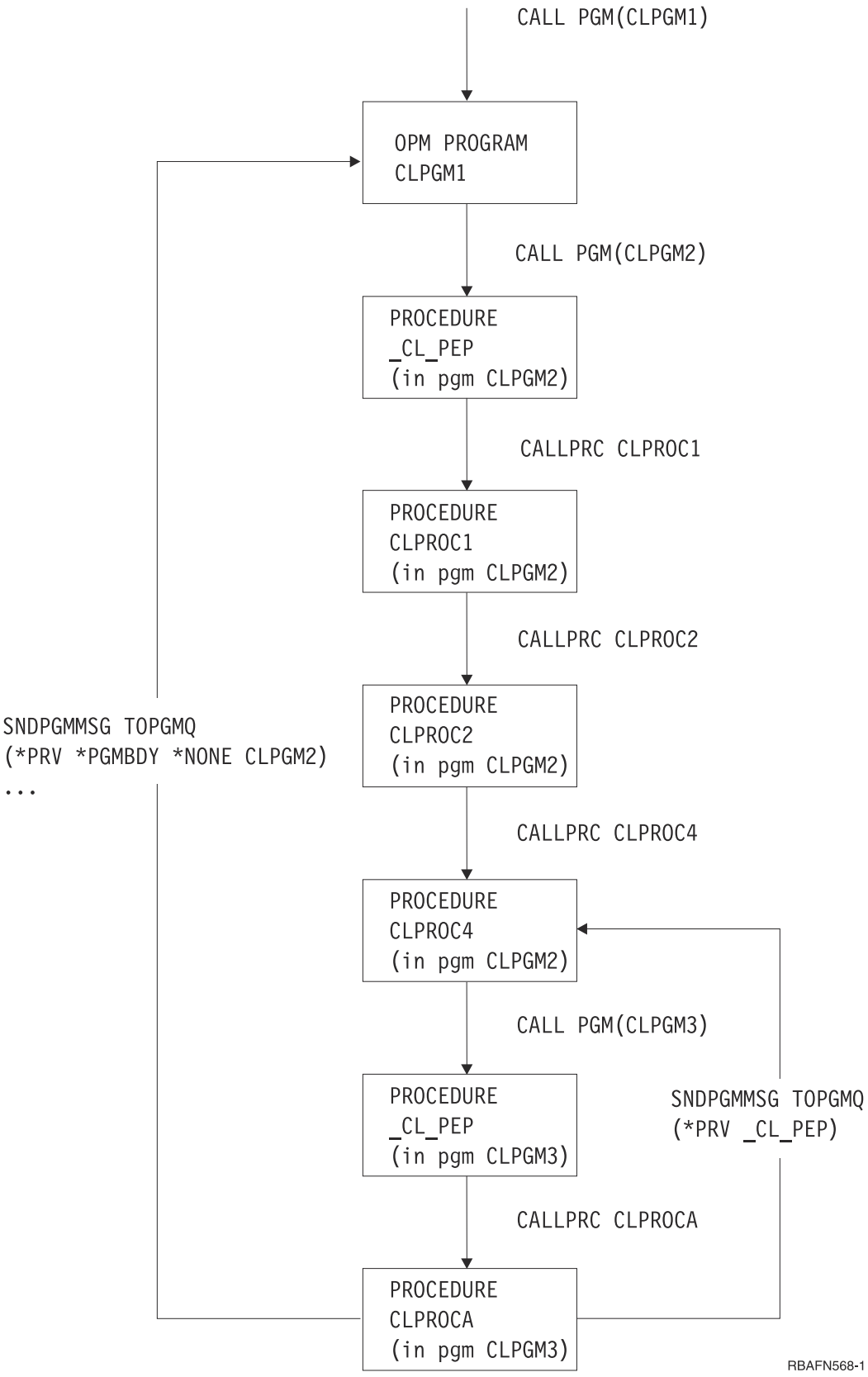


*Figure 21. Example of using *PGMBDY in a complex name*

The special value *PGMBDY can also be used with an original program model (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. The following figure 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.

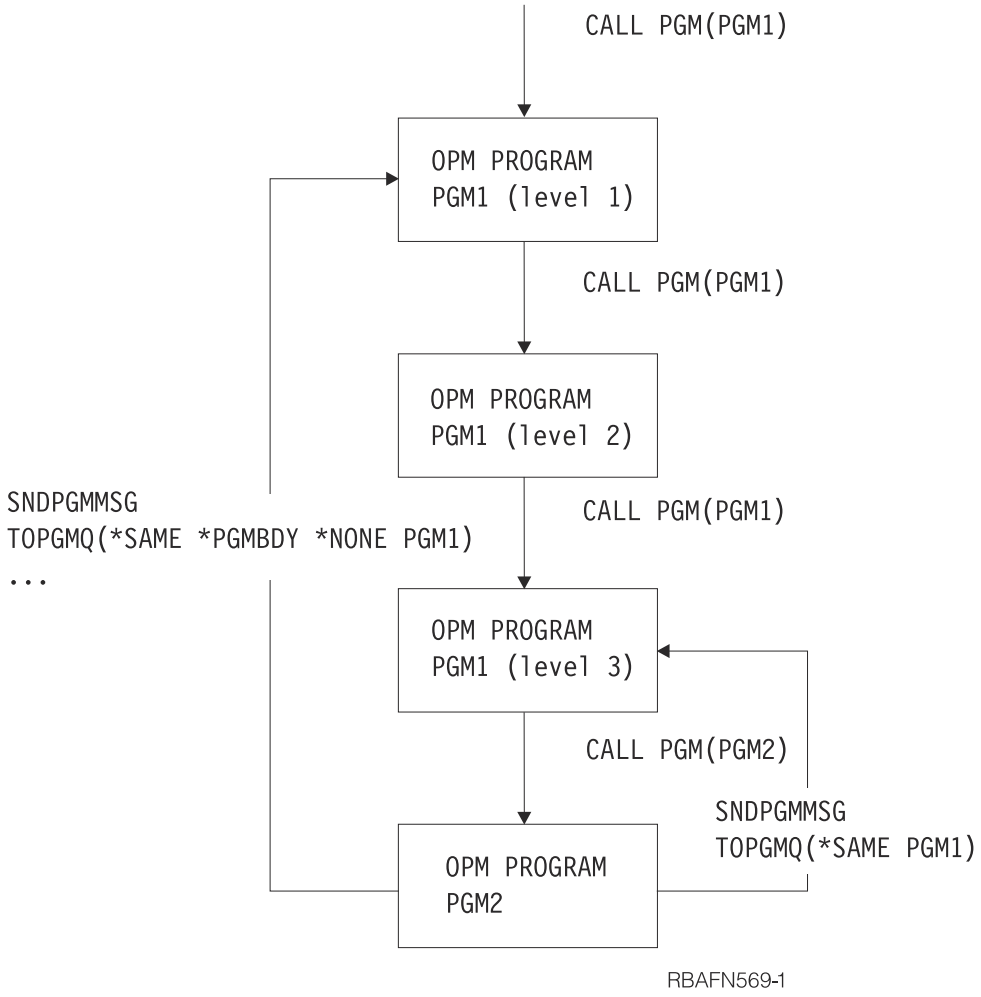


*Figure 22. Example of using *PGMBDY with an OPM name with recursion*

*Using the most recently called procedure as a base (*PGMNAME)*

To specify the most recently called procedure as the base call stack entry, use the *PGMNAME special value.

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 Integrated Language Environment (ILE) program. The special value *PGMNAME is used with an 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:

- CLPGM51 is an ILE program with procedures PROCA and PROCB.
- CLPGM52 and CLPGM53 are both original program model (OPM) programs.
- CLPGM53 is to send a message to CLPGM51 and does not know which procedure is the most recently called.

The send operation is accomplished using the special value *PGMNAME and the program name CLPGM51.

The special value *PGMNAME is useful if you convert some CL programs, but not all CL programs, to ILE programs. For example, CLPGM71 is an OPM CL program; CLPGM73 sent messages to CLPGM71 and specifies TOPGMQ(*SAME CLPGM71). If CLPGM71 is converted to ILE, only the **Send Program Message (SNDPGMMSG)** command with the *PGM name in CLPGM73 (OPM) works. Specifying CLPGM71 as a simple name does not work because there was no entry in the call stack for CLPGM71. If you change the command to TOPGMQ(*SAME *PGMNAME *NONE CLPGM71), CLPGM73 sends messages successfully to CLPGM71 regardless of the names you may have used for procedure names.

The special value *PGMNAME can also be used 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.

Related concepts

Using a control boundary as a base (*CTLBDY)
To identify the base call stack entry as the one at the nearest control boundary, use the special value *CTLBDY.

*Using a control boundary as a base (*CTLBDY)*

To identify the base call stack entry as the one at the nearest control boundary, use 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 the second figure for an example of sending a message using the special value *CTLBDY. The three programs in this example (CLPGM91, CLPGM92, and CLPGM93) are all Integrated Language Environment (ILE) programs. CLPGM91 runs in activation group AG91 while both CLPGM92 and CLPGM93 run in activation group AG92. In this example, PROC93A sends a message back to the entry that immediately precedes the boundary for AG92.

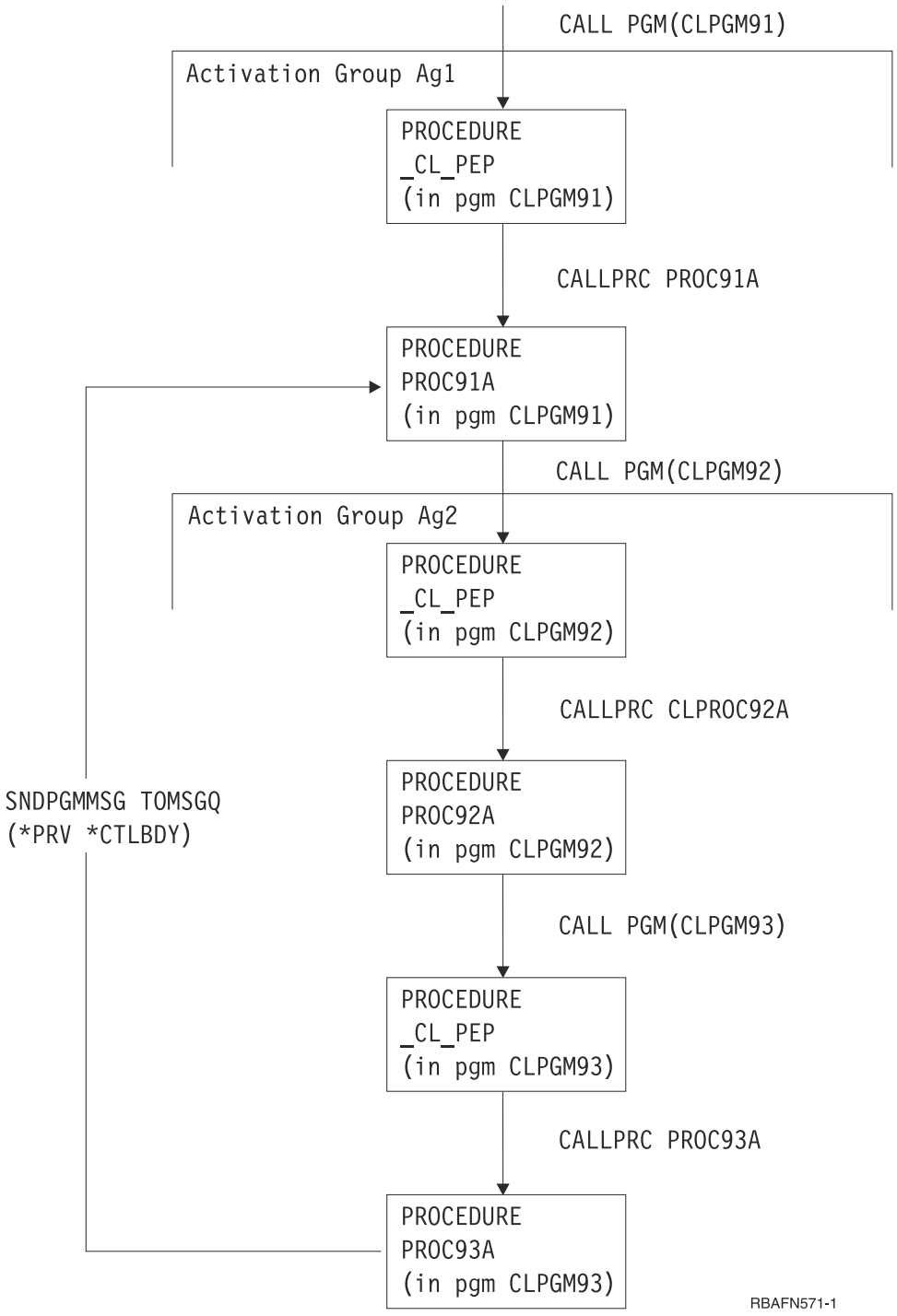


*Figure 23. Example of using *CTLBDY*

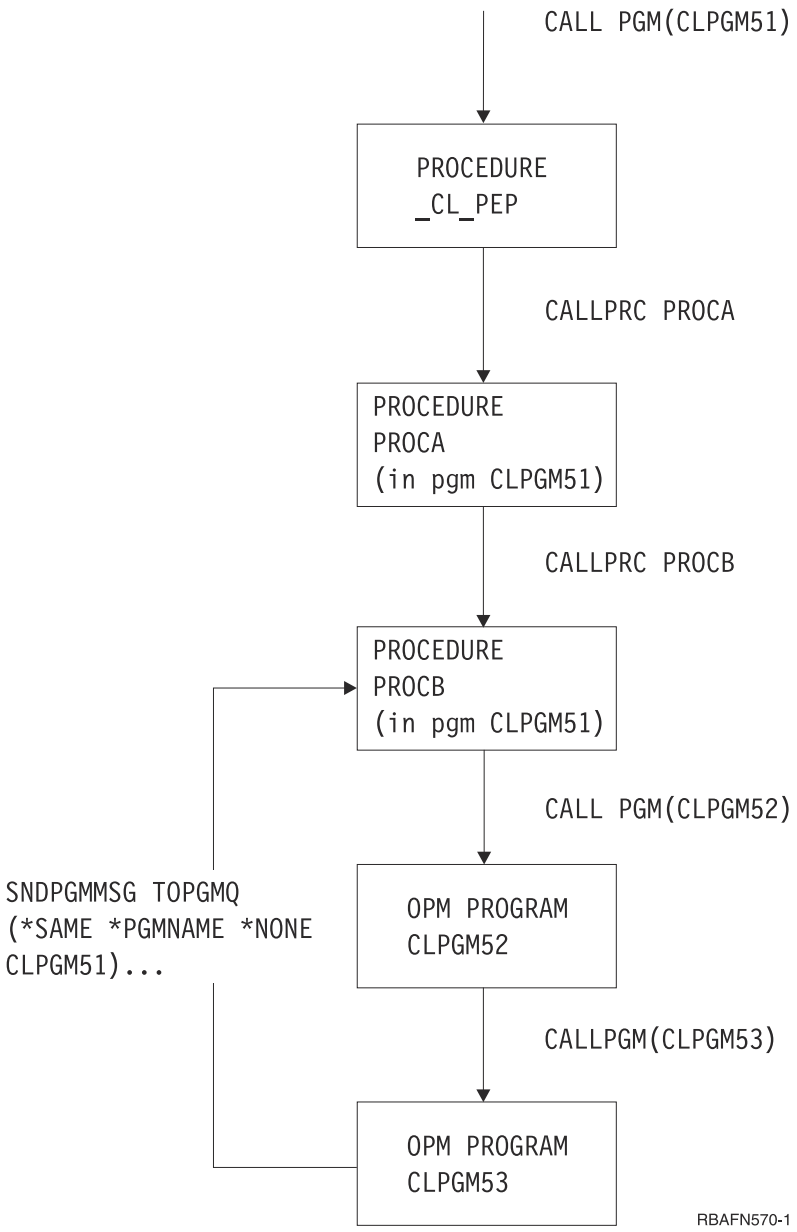


*Figure 24. Example of *PGMNAME*

Related concepts

Using the most recently called procedure as a base (*PGMNAME)

To specify the most recently called procedure as the base call stack entry, use the *PGMNAME special value.

Considerations for service programs

Integrated Language Environment (ILE) programs differ from ILE service programs in many ways.

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 into a CL procedure or program

To obtain a message from a message queue for your procedure or program, use the **Receive Message (RCVMSG)** command.

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 the MSGKEY parameter on the **Receive Message (RCVMSG)** command to identify the CL variable through which the key is to be passed. This will indicate the specific message to be received.
  - 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. Specify either a specific message type, or a value of *ANY to represent any type of message.
- 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 **Receive Message (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 **Change Message Queue (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.
- 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 on the **Receive Message (RCVMSG)** command). One or more of the following return variables can be specified:
  - 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, but a minimum of 87 characters are needed to obtain the current user information)
- 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)

In the following sample **Receive Message (RCVMSG)** command, a message is obtained from message queue INVN in the QGPL library. The message text received is placed in the variable &MSG. *ANY is the default value on the MSGTYPE parameter.

```
RCVMSG MSGQ(QGPL/INVN) MSGTYPE(*ANY) MSG(&MSG)
```

When working with the call stack entry message queue of an Integrated Language Environment (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 **Receive Message (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.

Related tasks

Removing messages from a message queue
Messages are held on a message queue until they are removed.

Receiving request messages

Receiving request messages is a method for your CL procedure or program to process CL commands.

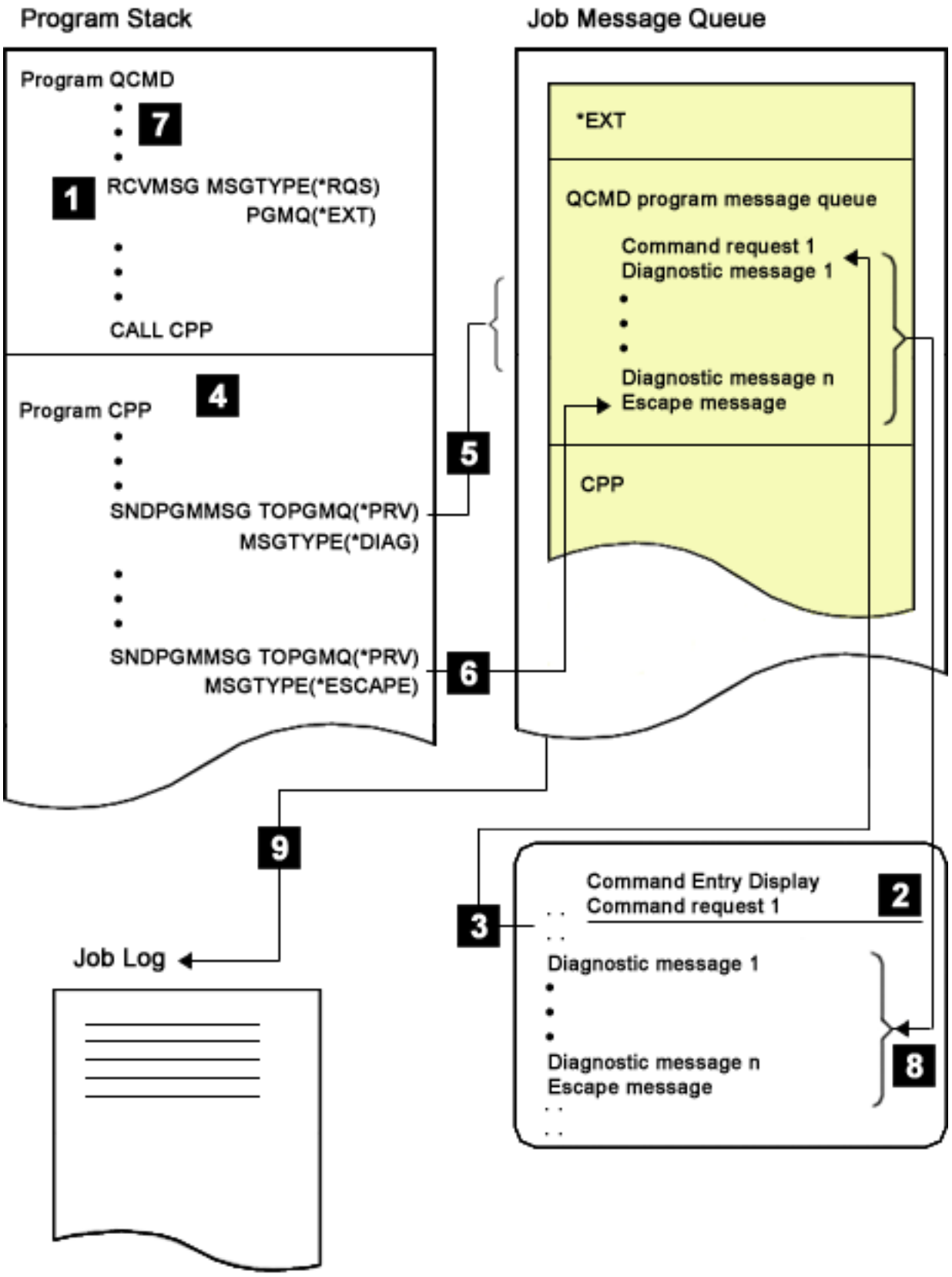
For example, your procedure or program can obtain input (commands) from a display station and handle the messages that result from the analysis and processing of the input. The commands are sent as request messages so they can be processed. Typically, request messages are received from the external message 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 QCMD.

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 original program model (OPM) program or Integrated Language Environment (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 then the nesting occurs within the same ILE program and the nesting levels remain independent.

The following figure 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

Since step 2 placed a request on *EXT, now the RCVMSG in step 1 can continue. The command is then moved to the QCMD call message queue and is passed from there to QCMD to complete step 1.

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.

Note: 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.

Related concepts

Log messages
There are two types of logs for messages: job logs and history logs.

Related reference

Commands used to send messages from a CL program
The **Send Program Message (SNDPGMMSG)** command or the **Send User Message (SNDUSRMSG)** command is used to send a message from a CL procedure or program.

Writing request-processor procedures and programs

To become a request-processing procedure or program, your procedure or program must send and receive a request message and not remove the request message. The **Send Program Message (SNDPGMMSG)** and **Receive Message (RCVMSG)** commands can make your procedure or program become a request processor.

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 “Receiving request messages” on page 543.
- 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.

For example, the following commands make a procedure or program 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 later 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 because the request is moved to a different call message queue.) 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 (**Monitor Message (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 **RCVMSG** 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 reference key to have the IBM i command analyzer to process a request message for an IBM i command. You can get the message reference 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 reference key to QCAPCMD.

Determining if a request processor exists

To determine if a job has a request processor, you need to 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 Integrated Language Environment (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 **Remove Message (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 **Remove Message (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 **Receive Message (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 message descriptions from a message file

To retrieve the text of a message from a message file into a variable in a CL program or procedure, use the **Retrieve Message (RTVMSG)** command. The **RTVMSG** command operates on predefined message descriptions.

You can specify the message identifier and message file name in addition to the following items:

- CCSID to convert to. It specifies the coded character set identifier that you want your message text and data returned in.
- Message data fields. The message data to be inserted for the substitution variables.
- Message data CCSID. It 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 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 substitution variable &1 and &2 is contained in the procedure variable &A, in which the values of the procedure variables &FILE and &LIB have been concatenated. When the retrieve command is done, 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 **Retrieve Message (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 can perform the following tasks:

- Send the message using the **Send Program Message (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

Removing messages from a message queue

Messages are held on a message queue until they are removed.

You can remove them using the **Remove Message (RMVMSG)** command, **Clear Message Queue (CLRMSGQ)** command, the RMV parameter on the **Retrieve Message (RTVMSG)** 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 **Remove Message (RMVMSG)** command or a single old message using the **Retrieve Message (RTVMSG)** 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 message and its reply are removed. If you remove an inquiry message that you have already answered, both the message and its reply are removed.

To remove all messages for all inactive programs or procedures from a user's job message queue, specify *ALLINACT for the PGMQ parameter and *ALL for the CLEAR parameter on the **Remove Message (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 Integrated Language Environment (ILE) procedure, it is possible that an exception message for unhandled exceptions is on the queue at the time the **Remove Message (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 **Remove Message (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 **Remove Message (RMVMSG)** command removes all messages from the call message queue for the procedure containing the command.

```
RMVMSG CLEAR(*ALL)
```

Related tasks

Receiving messages into a CL procedure or program

To obtain a message from a message queue for your procedure or program, use the **Receive Message (RCVMSG)** command.

Monitoring for messages in a CL program or procedure

Messages that can be monitored are *ESCAPE, *STATUS, and *NOTIFY messages that are issued by each CL command used in the program or procedure.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 610.

Each IBM-supplied command identifies in its help documentation which exception messages it generates. You can use this information to determine which messages you want to monitor in your program or procedure.

Exception messages include escape, notify, and status messages that are sent to the call message queue of your CL procedure or program by the commands in your procedure or program or by commands in another procedure or program. Diagnostic messages cannot be monitored.

Using the **Monitor Message (MONMSG)** command, you can monitor for one or more messages sent to the call message queue for the conditions specified in the command. You can then use the **MONMSG** command to specify that 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 program or 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 original program model (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. 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 message 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 specified as message data when the CPF5101 message is sent by a program.

```
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, and 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 program or 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 (CALL)** commands and a procedure-level MONMSG command for CPF0001. This escape message occurs if a **Call (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 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 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 situations happen:

- 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. Online help and documentation for each command contains a list of the escape, notify, and status messages that are issued for the command. You should also keep a list of all messages that you have defined.

Note: The previous 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.

Related concepts

Escape and notify messages

You can send both escape and notify messages from your CL program or procedure to message queues.

Related tasks

Assigning a message identifier

The message identifier you specify on the **Add Message Description (ADDMSGD)** command is used to refer to the message and is the name of the message description.

Watching for messages

The system provides a watch-for-event function that allows you to watch for messages.

The watch-for-event function allows you to specify messages for which you want the system to watch. When these messages occur, a user exit program is called to take any necessary action. You must specify the message queue or job log where you expect the message to be sent. You can specify some text string to be compared against the message data, the from-program, or the to-program. You can also select watches by message type or severity. Using this watch selection capability allows you to be informed of only certain messages.

Note: Be careful when starting message watches. Do not create never-ending loops of watch notifications. For example, do not start a message watch in *ALL jobs for the job-start message CPF1124.

Use the **Start Watch (STRWCH)** command to start a watch session and be notified when a specified message occurs. When the watched-for message is added to the specified message queue or log, the watch exit program is called. The watch exit program runs in a job in the QUSRWRK subsystem.

The **Work with Watches (WRKWCH)** command can be used to show a panel with the list of active watches on the system. The **End Watch (ENDWCH)** command can be used to end watch sessions.

The Start Watch (QSCSWCH) and End Watch (QSCEWCH) APIs have a similar usage to the STRWCH and ENDWCH commands.

The Retrieve Watch Information (QSCRWCHI) and Retrieve Watch List (QSCRWCHL) APIs can be used to obtain information about watches.

The watch for event function is also incorporated into some trace commands:

- **Start Trace (STRTRC)**
- **Start Communications Trace (STRCMNTRC)**
- **Trace Internal (TRCINT)**
- **Trace Connection (TRCCNN)**
- **Trace TCP Application (TRCTCPAPP)**

Watch support enhances the trace functions by automatically monitoring and ending traces when certain predetermined criteria are met. Using watch support can prevent the loss of valuable trace data and reduce the amount of time you need to spend monitoring traces.

Related information

Watch for event function
Monitoring APIs
Start Watch (STRWCH) command
End Watch (ENDWCH) command
Work with Watches (WRKWCH) command
Start Watch (QSCSWCH) API
End Watch (QSCEWCH) API
Retrieve Watch Information (QSCRWCHI) API
Retrieve Watch List (QSCRWCHL) API

CL handling for unmonitored messages

The system provides default monitoring and handling of any messages you do not monitor.

Many escape messages can be sent to a program or procedure that calls commands, programs, and procedures. You might want to monitor and handle the escape messages that pertain to the function of your program or procedure. However, you might not want to monitor and handle all of the messages.

Default handling assumes that an error has been detected in a program or procedure. If you are debugging the program or 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 program or procedure, the system performs a message percolation function.

Message percolation is a two-step function that completes the following tasks:

- Moving the escape message one step earlier in the call stack.
- Determining whether the program or procedure has a **Monitor Message (MONMSG)** command for the escape message.

If the program or procedure has a **Monitor Message (MONMSG)** command for the escape message, the message percolation action stops, and the system takes the action that is specified by the **Monitor Message (MONMSG)** command. Message percolation continues until either finding a **Monitor Message (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 program or procedure with a **Monitor Message (MONMSG)** command that applies to the message. The system considers the action on the original escape message complete. The system then sends the function check message (CPF9999) to the program or procedure that is the target of the original escape message. If that program or procedure has a **Monitor Message (MONMSG)** command for the function check message, 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 operator can reply with one of the following replies:

R
Retry the failing command in the program or procedure.

I
Ignore the message. Continue processing at the next command in the program or procedure.

C
Cancel the program or procedure and percolate the function check to the previous program or procedure on the call stack.

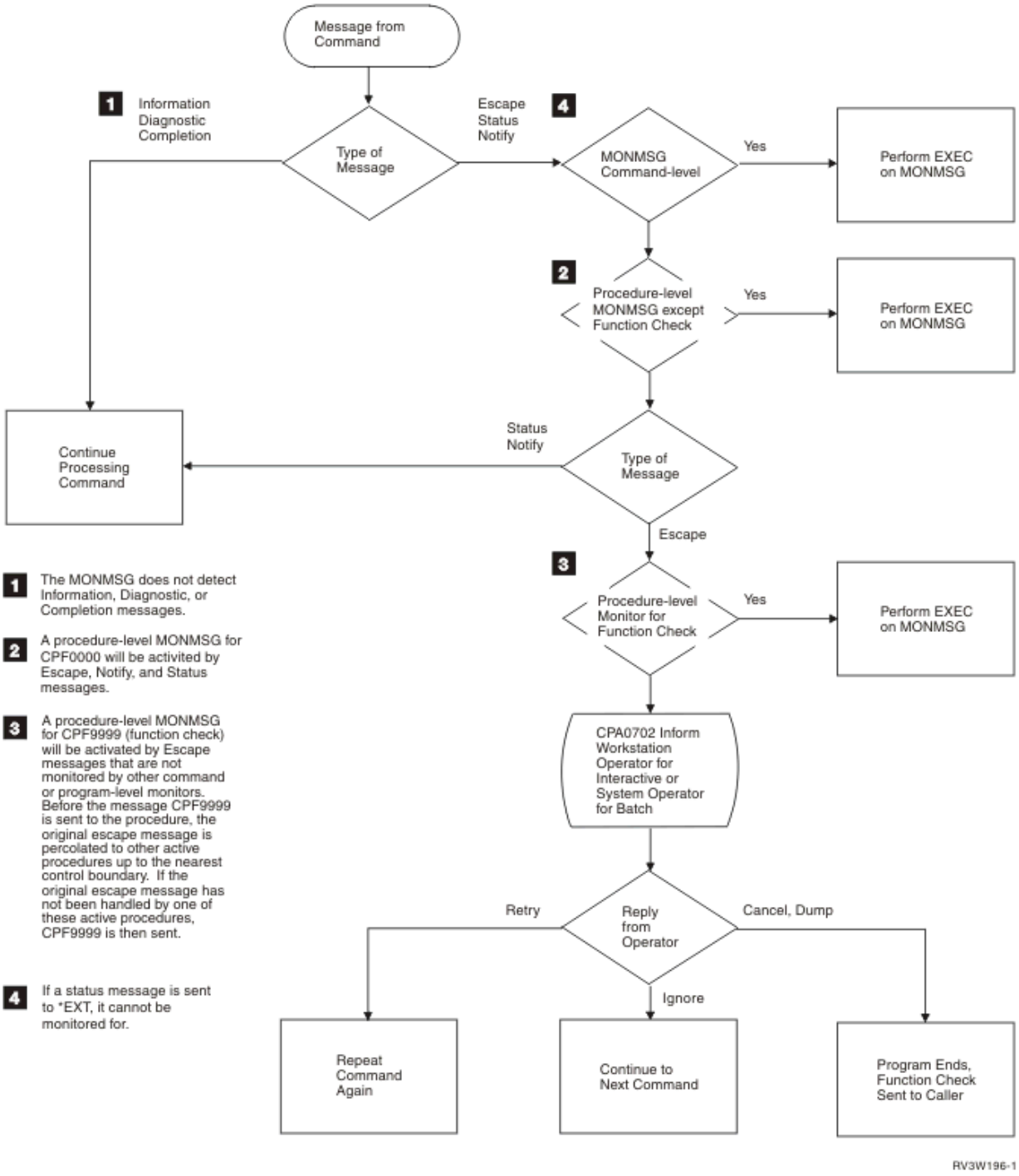
D
Dump the call stack entry for the failing program or procedure, cancel the program or procedure, and percolate the function check to the previous program or procedure on the call stack. This is the default action if the workstation operator enters no reply, or if the job is a batch job.

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 program or 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 program or procedure, or continue with your program or 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.

The following figure shows how CL monitors for messages and notifies the workstation user of a function check for default handling.



Related tasks

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.

Monitoring for notify messages

You can monitor for notify messages that are sent to the call message queue of your CL procedure or program 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 **Monitor Message (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.

Monitoring for status messages

You can monitor for status messages that are sent by the commands in your CL procedure or by the programs and 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 **Monitor Message (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 IBM i 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.

Receiving a message from a program or procedure that has ended

Occasionally, there is a need to receive messages from the job log that are sent to a call message queue that is no longer active.

For example, PGMA calls PGMB and PGMB sends a diagnostic message to itself and an escape message to PGMA. PGMA can easily receive the escape message that was sent to it, but sometimes it is necessary to also receive the diagnostic message. Because the diagnostic is on a call message queue that is no longer active, the message must be received by using a message reference key. This can be done, but it takes more work than one simple receive message command.

The following considerations are important in understanding how this works:

- If the **Receive Message (RCVMSG)** command (or QMHRCVPM API) is coded to receive a message by key from its own call message queue, it will receive the message with that key (if it still exists) from the job log, regardless of what call message queue the message was sent to or whether that procedure is active.
- The message reference key for messages in the job log can be treated as a 4-byte unsigned integer that is incremented for each message sent (even though it is documented externally as a 4-character value).

The following steps are required to obtain a message from a program or procedure that has ended:

1. Send a message (and optionally remove it) to obtain a starting message reference key.
2. Run the function that will send the messages you are interested in.
3. Send another message to obtain an ending message reference key.
4. In a loop, increment the message reference key between the starting and ending message keys identified in steps 1 and 3 and receive the message for each key. There can be gaps in the message reference key so the CPF2410 exception from the receive message function must be handled. The gaps can occur from a message that have been deleted or from internal exception handling.

Note: This topic provides a procedure and example for one way to receive a message from a program or procedure that has ended. There are other techniques that can be used to access some or all of the messages in the job log. These are summarized in the following list, but are not explained in detail, nor are sample programs provided:

- Write the job log to an outfile. This can be done either by specifying OUTPUT(*OUTFILE) on the **Display Job Log (DSPJOBLOG)** command, or using the QMHCTLJL API to define the job log outfile specifications and either ending the job or specifying OUTPUT(*APIDFN) on the **Display Job Log (DSPJOBLOG)** command. When you do this all messages in the job log will be written to the outfile. So messages that are left on a call message queue when the program or procedure ends are included.
- Use the List Job Log Messages (QMHLJOBL) API to copy messages from the job log into a user space. You can select all of the messages in the job log or a subset, and you can select which fields are returned for each message. There are APIs that can be used to access the contents of the user space. The QUSPTRUS API can be used to get a pointer to the data in the user space. Then the pointer can be used with languages that support pointers. API QUSRTVUS can be used to retrieve data from the user space in order to use languages that do not support pointers.

A sample program of the procedure described previously is provided as follows using the RCVMSG command and the QMHRCVPM API.

Note: By using the code example, you agree to the terms of the "Code license and disclaimer information" on page 610.

```
*****************************************************************************
 *
```

```
 * CL program example using the RCVMSG
command
 *
 *
 ****************************************************************************

 This has the advantage of being easier to code and is easier to follow.
 It has the disadvantage of requiring the program to monitor for and handle
 the CPF2410 exception, which is not a good thing if performance is important.


PGM
DCL &LOWKEY *CHAR 4
DCL &HIKEY *CHAR 4
DCL &MSGKEY *CHAR 4
DCL &MSG *CHAR 256
/*-------------------------------------------------------------------*/
/* OBTAIN STARTING MESSAGE REFERENCE KEY                             */
/*-------------------------------------------------------------------*/
SNDPGMMSG  MSG(TEST) TOPGMQ(*SAME) KEYVAR(&LOWKEY)
RMVMSG     MSGKEY(&LOWKEY)
/*-------------------------------------------------------------------*/
/* EXECUTE FUNCTION                                                  */
/*-------------------------------------------------------------------*/

  ---- Insert whatever command(s) or program call(s) you want ----
  ---- to handle messages for here                             ----

/*-------------------------------------------------------------------*/
/* OBTAIN ENDING MESSAGE REFERENCE KEY                               */
/*-------------------------------------------------------------------*/
SNDPGMMSG  MSG(TEST) TOPGMQ(*SAME) KEYVAR(&HIKEY)
RMVMSG     MSGKEY(&HIKEY)
/*-------------------------------------------------------------------*/
/* LOOP TO RECEIVE MESSAGES ON INACTIVE INVOCATION                   */
/*-------------------------------------------------------------------*/
CHGVAR %BIN(&MSGKEY 1 4)  (%BIN(&LOWKEY 1 4) + 1)
LOOP:
RCVMSG     PGMQ(*SAME (*)) MSGKEY(&MSGKEY) RMV(*NO) MSG(&MSG)
MONMSG CPF2410 EXEC(DO) /* HANDLE MSGKEY NOT FOUND                  */
  RCVMSG     MSGTYPE(*EXCP) RMV(*YES) /* REMOVE UNWANTED EXCEPTION */
  GOTO       SKIP
ENDDO

    ---- Insert code here to do whatever processing is needed  ----
    ---- for the message.  You may need to add additional      ----
    ---- values, such as message ID, message type, etc., to    ----
    ---- the RCVMSG command.                                   ----

SKIP:
CHGVAR %BIN(&MSGKEY 1 4)  (%BIN(&MSGKEY 1 4) + 1)
IF (&MSGKEY *LT &HIKEY) GOTO LOOP
ENDPGM


 ****************************************************************************
 *
 * CL program using the QMHRCVPM API.
 *
 ****************************************************************************

This sample is similar to the first sample.  The only difference is that
it uses the QMHRCVPM API instead of the RCVMSG CL command.  By using the error
code structure, it eliminates the need to handle the CPF2410 exception and the
overhead required to send the exception in the case of the message key not
being found.  This example shows how to extract the message text from the
structure returned by the API.  If the message is an immediate message (i.e. no
message ID), the message is in the 'Replacement data or impromptu message text'
area; otherwise it is in the 'Message' area which follows the 'Replacement
data' field.  The example checks the message length to determine which of these
fields to use for the message.

PGM
DCL &LOWKEY *CHAR 4
DCL &HIKEY *CHAR 4
DCL &MSGKEY *CHAR 4
DCL &MSG *CHAR 256
/*-------------------------------------------------------------------*/
/* Some messages have a large amount of message data, in which case */
/* the size of the &MSGINFO variable will not be adequate.  If you  */
/* expect to receive messages with a large amount of message data,  */
/* you will need to increase the size of this variable accordingly. */
```

```
/* Be sure to also change the value that is put into &MSGINFOL to   */
/* reflect the size of the variable.                                */
/*------------------------------------------------------------------*/
DCL &MSGINFO *CHAR 1000
DCL &MSGINFOL *CHAR 4
DCL &ERRCODE *CHAR 16
DCL &MSGOFFS *DEC (4 0)
DCL &MSGLEN *DEC (4 0)
/*------------------------------------------------------------------*/
/* OBTAIN STARTING MESSAGE REFERENCE KEY                            */
/*------------------------------------------------------------------*/
SNDPGMMSG  MSG(TEST) TOPGMQ(*SAME) KEYVAR(&LOWKEY)
RMVMSG     MSGKEY(&LOWKEY)
/*------------------------------------------------------------------*/
/* EXECUTE FUNCTION                                                 */
/*------------------------------------------------------------------*/

  ---- Insert whatever command(s) or program call(s) you want ----
  ---- to handle messages for here                             ----

/*------------------------------------------------------------------*/
/* OBTAIN ENDING MESSAGE REFERENCE KEY                              */
/*------------------------------------------------------------------*/
SNDPGMMSG  MSG(TEST) TOPGMQ(*SAME) KEYVAR(&HIKEY)
RMVMSG     MSGKEY(&HIKEY)
/*------------------------------------------------------------------*/
/* LOOP TO RECEIVE MESSAGES WITH QMHRCVPM API                       */
/*------------------------------------------------------------------*/
CHGVAR %BIN(&MSGKEY 1 4)  (%BIN(&LOWKEY 1 4) + 1)
CHGVAR %BIN(&MSGINFOL 1 4) 1000
CHGVAR %BIN(&ERRCODE 1 4) 16
LOOP2:
CALL QMHRCVPM (&MSGINFO &MSGINFOL RCVM0200 '*          ' +
               X'00000000' '*ANY      ' &MSGKEY X'00000000' +
               '*SAME     ' &ERRCODE)
IF ((%BIN(&MSGINFO 5 4) *GT 0) *AND (%BIN(&ERRCODE 5 4) *EQ 0)) +
  DO /* IF A MESSAGE WAS RECEIVED */
    IF (%BIN(&MSGINFO 161 4) *EQ 0) +
      DO /* IMPROMPTU MESSAGE */
        CHGVAR &MSGLEN %BIN(&MSGINFO 153 4)
        CHGVAR &MSGOFFS 177
      ENDDO
    ELSE +
      DO /* STORED MESSAGE */
        CHGVAR &MSGLEN %BIN(&MSGINFO 161 4)
        CHGVAR &MSGOFFS (177 + %BIN(&MSGINFO 153 4))
      ENDDO
    CHGVAR &MSG %SST(&MSGINFO &MSGOFFS &MSGLEN)

    ---- Insert code here to do whatever processing is needed  ----
    ---- for the message.  You can extract additional          ----
    ---- values, such as message ID, message type, etc.,       ----
    ---- from the message information structure if necessary.  ----

  ENDDO
CHGVAR %BIN(&MSGKEY 1 4)  (%BIN(&MSGKEY 1 4) + 1)
IF (&MSGKEY *LT &HIKEY) GOTO LOOP2
ENDPGM
```

Related concepts

Call message queue

A call message queue is used to send messages between one program or procedure and another program or procedure.

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.

Note: By using the code examples, you agree to the terms of the “Code license and disclaimer information” on page 610.

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. If the system calls a break-handling program, it interrupts the job that has the message queue in break mode. Refer to the Break Handling Exit Program API information for details of parameters passed to a break handling program. When the break-handling program ends, the interrupted 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. During CHGMSGQ, the break program is called only once when a message is found that meets the criteria of the message severity for the queue, regardless of how many messages meeting the message severity criteria are found so the break-handling program should handle all of the messages on the queue. After CHGMSGQ, the break program is called for each message that meets the message severity criteria.
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 **RCVMSG** command. The message arrival event cannot be handled by the system while the job is running a break-handling event.
3. The message queue severity can be set to indicate that the break-handling program should only be called if the severity of the message sent is equal to or greater than the severity of the message queue.

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 previous example of a break-handling program, if a CPA5316 message should arrive at the queue while the **Display Message (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.

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 **Submit Job (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.

Related concepts

Message queues in break mode

A break-handling program can be 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.

Related tasks

Using the system reply list

By using the system reply list, you can specify that the system automatically issues the reply to specified predefined inquiry messages.

Related information

Break Handling Exit Program API

Ways of handling replies to inquiry messages

Handling replies includes using a sender copy message to obtain a reply, finding the job that sent a reply, using the system reply list, and using reply handling exit programs.

Using a sender copy message to obtain a reply

When an inquiry message is sent, it expects a reply. To allow the sender of an inquiry message to obtain a reply, a sender copy message is issued and associated internally with the inquiry message.

A *sender copy message* is a copy of the inquiry message that is used by the sender to obtain the reply that is sent to its associated inquiry message. Sending an inquiry and obtaining the reply can be done rather easily when the **Send User Message (SNDUSRMSG)** command is used in a source program. If the **SNDUSRMSG** command is not used, the function to send an inquiry and obtain the reply can be done by the **Send Program Message (SNDPGMMSG)** and **Receive Message (RCVMSG)** commands in a CL source program. An inquiry can be sent with the **SNDPGMMSG** command and the reply can be obtained with the **RCVMSG** command. For example, when an inquiry message is sent with the **SNDPGMMSG** command, the message reference key (MRK) of the sender copy can be returned on the KEYVAR parameter of the **SNDPGMMSG** command. The sender copy message is placed on the reply message queue that is specified on the **SNDPGMMSG** command. When a reply is sent to the inquiry message, internal message handling also sends the same reply to the sender copy message. Then the program can obtain the reply with the **RCVMSG** command by doing the following tasks:

- Using the MRK of the sender copy from the KEYVAR parameter of **SNDPGMMSG** and specify it on the MSGKEY parameter of the **RCVMSG** command.
- Specifying a message type of reply.
- Providing a wait time to allow time for a reply to be sent to the inquiry message.

Related tasks

Example: Sending an immediate message and handling a reply
This example shows how a program or procedure sends an inquiry message and handles the reply.

Related reference

Examples: Sending messages
These examples show how to send different kinds of messages.

Finding the job that sent the reply

Occasionally you might want to know the job that replied to an inquiry message on various standard message queues.

The information is not available through **Display Message (DSPMSG)** because the reply is associated with the inquiry message (or sender copy), so when the details of the reply are viewed, the information displayed pertains to the inquiry message, not the reply. The sender of a reply can be viewed by doing a DSPMSG msgqname OUTPUT(*PRINT). The spooled file that is generated includes the sending job of the reply message.

If the message queue in question is QSYSOPR, there is another way to obtain the information. If DSPLOG is specified with a time period around the time of the inquiry message, find the reply, position the cursor on the reply message, and press help. Then, press F9 to view details to see the job that sent the reply.

Using the system reply list

By using the system reply list, you can specify that the system automatically issues the reply to specified predefined inquiry messages.

Only inquiry messages can be automatically responded to with the system reply list.

The system reply list contains message identifiers, optional compare data that must match the message data in the inquiry message, a reply value for each message, and a dump attribute that effectively does DSPJOB OUTPUT(*PRINT). 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 inquiry 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 by setting the inquiry message reply job attribute to *SYSRPYL.

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. If the message ID does not identify a default reply value the *N is sent as the reply.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 (sequence number 15) that was added overrides this entry for message RPG1241, so RPG1241 takes the action for sequence number 15, not 18.
- 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 of the inquiry message 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 predefined 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 data that is of type *CCHAR only.

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.

If messages are added to the system reply list, it is helpful to have a CL program created to add the entries. For example, if the reply list is damaged and gets cleared after an IPL, the CL program can be called to re-add the entries, rather than having users re-add entries manually.

If a program has not been created to update system reply list entries, there are some alternatives to re-establish the system reply list after it has been lost:

- After changes have been made to the system replay list, use the **Retrieve System Information (RTVSYSINF)** command to save data to a library. Note that this also saves other system information in addition to the system reply list. Then later, when the system reply list needs to be updated, the **Update System Information (UPDSYSINF)** command can be used. The UPDSYSINF command uses the data that was saved with the RTVSYSINF command.
- Another alternative is to restore the system reply list from a backup. This assumes that a save operation of the system has been done and all the necessary entries were added to the system reply list before the save operation. A slip installation of the base operating system needs to be done with the following steps:
  1. With the key in MANUAL mode, do an IPL.
  2. On the DST IPL/install screen, select option 2 to install.
  3. On the Install Operating System screen (that is the time/date screen), select option 2 to change install options.
  4. On the Specify Install Options screen, select option 1 to restore program and language options from media.
  5. On the Specify Restore Options screen, select option 1 to restore for MESSAGE REPLY LIST. It is about the third item on the screen.

     Note: Usually, the default value of this screen is 2 or 3 to indicate not to restore the item.
  6. Continue the installation and the command entry display should be shown.
  7. Take the key out of manual mode.

Related tasks

Placing a message queue in break mode automatically

By placing a message queue in break mode automatically, you can monitor the QSYSOPR message queue.

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.

Related information
Add Message Description (ADDMSG) command

Using the reply handling exit program

By using the reply handling exit program, a user-supplied exit program can be called when a reply is sent to an inquiry message.

The exit program can accept, reject or replace the reply value.

Related information
Reply Handling Exit Program

Message subfiles in a CL program or procedure

In CL programs and procedures, message subfiles are the only type of subfiles supported. Message subfiles let a controlling program or procedure display one or more error messages.

To use subfile message support, run a **Send File (SNDF)** or **Send/Receive File (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 original program model (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.

Related concepts
DDS

Log messages

There are two types of logs for messages: job logs and history logs.

A job log contains information related to requests entered for a job. The system history log (QHST) contains system data, such as a history of job start and end activity on your system.

Related tasks
Receiving request messages
Receiving request messages is a method for your CL procedure or program to process CL commands.

Job log

Each job has an associated job log.

The job log can contain the following items 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 is run with the LOGCLPGM (*YES) option.
- All messages and message help sent to the requester and not removed from the call message queues.

Related concepts
External message queue

The external message queue (*EXT) is used to communicate with the external requester (such as a display station user) of the job.

Related tasks

Logging CL program or procedure commands
You can specify that most CL commands that are run in a CL program or procedure be written (logged) to the job log.

Writing a job log to a file

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. When a job's job log output job attribute is *PND, no job log is produced when the job ends, but the pending job log is still accessible from the **Work with Job Logs (WRKJOBLOG)** command.

You can use the QMHCTLJL API to write a job log to a database file. 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.

Related reference

Job log output files
When you use job log output files, consider the job log information you want in the files and the model files that describe the output-file record formats.

Related information

Control Job Log Output (QMHCTLJL) API

Job log pending

Controlling information written in a job log

To control what information the system writes in the job log, specify the LOG parameter on the **Create Job Description (CRTJOBD)** command. You can change the 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.

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.

Job log message filtering

Message filtering is the process of removing messages from the job log based on the message logging level set for the job.

Before each new request is received by a request processing program, message filtering occurs.

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.

Example: Controlling information written in a job log

This 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 is run when the next request is received.

Note: Both high-level and detailed message logging levels are included in the examples. High-level messages start with the text of *Message*; detailed messages start with the text of *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 caller or the previous call message queue, for example, *PRV, which would be QCMD in this example using the command entry display. 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, which is a high-level message.

Note in the following display, after PGMA and PGMB are called, that the CHGJOB command no longer appears on the display. According to logging level 2, only requests for which a high-level 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. CHGJOB was removed, or filtered from the job log, when the next request CALL PGMA was received. When a new request is received, the previous request is filtered according to the log level. If such a high-level 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
```

Request CALL PGMA results in a high-level message equal to or greater than the current log severity, so when request CALL PGMB is entered, no messages from PGMA are filtered.

3. When F10=Include detailed messages is pressed from the Command Entry display, all the messages associated with the request CALL PGMA are displayed. All messages for request CALL PGMB are also shown because no filtering for the request has occurred yet.

```
                              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, another 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. Press F10 again to exclude detailed messages.

   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 its caller (*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 (CALL PGME) 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 (filtered from the job log) 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 end the job and 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 last 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.

```
 5770SS1 V7R1M0 100416                          Job Log                              SYSAS727 01/16/11 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   01/16/11  07:13:19.570504  QWTPIIPP     QSYS        0613     *EXT                    *N
                                    Message . . . . :   Job 038518/JOHNDOE/QPADEV000C started on 01/16/11 at
                                      07:13:19 in subsystem QINTER in QSYS. Job entered system on 01/16/11 at
                                      07:13:19.
*NONE      Request                    01/16/11  07:13:24.318144  QMHGSD       QSYS        0010     QCMD        QSYS        0178
                                    Message . . . . :   -CALL PGMA
MSG1001    Information            20   01/16/11  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   01/16/11  07:13:24.361416  PGMA         JOHNDOE     0032     PGMA        JOHNDOE     0032
                                    Message . . . . :   Detailed message sev 50 - PGMA
                                    MSG1002 second level text - PGMA
MSG1003    Information            60   01/16/11  07:13:24.361592  PGMA         JOHNDOE     0036     PGMA        JOHNDOE     0036
                                    Message . . . . :   Detailed message sev 60 - PGMA
                                    MSG1003 second level text - PGMA
MSG1004    Information            20   01/16/11  07:13:24.361776  PGMA         JOHNDOE     003A     QCMD        QSYS        01A6
                                    Message . . . . :   Message sev 20 - PGMA
                                    MSG1004 second level text - PGMA
MSG1005    Information            50   01/16/11  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   01/16/11  07:13:24.362552  PGMA         JOHNDOE     004C     QCMD        QSYS        01A6
                                    Message . . . . :   Message sev 60 - PGMA
                                    MSG1006 second level text - PGMA
*NONE      Request                    01/16/11  07:13:24.370240  QMHGSD       QSYS        0018     QCMD        QSYS        0178
                                    Message . . . . :   -CHGJOB LOG(3 40 *SECLVL)
*NONE      Request                    01/16/11  07:13:24.370672  QMHGSD       QSYS        001C     QCMD        QSYS        0178
                                    Message . . . . :   -CALL PGMC
MSG100F    Information            30   01/16/11  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   01/16/11  07:13:24.379608  PGMC         JOHNDOE     *STMT    QCMD        QSYS        01A6
                                    From module . . . . . . . . . :   PGMC
                                    From procedure  . . . . . . . :   PGMC
                                    Statement . . . . . . . . . . :   8200
 5770SS1 V7R1M0 100416                          Job Log                              LPAR3TLM 01/16/11 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                    01/16/11  07:13:24.379928  QMHGSD       QSYS        0020     QCMD        QSYS        0178
                                    Message . . . . :   -CALL PGMD
*NONE      Request                    01/16/11  07:13:24.383568  QMHGSD       QSYS        0024     QCMD        QSYS        0178
                                    Message . . . . :   -CALL PGME
*NONE      Request                    01/16/11  07:13:35.166408  QMHGSD       QSYS        073E     QCMD        QSYS        0178
                                    Message . . . . :   -signoff *list
CPF1164    Completion             00   01/16/11  07:13:35.173496  QWTMCEOJ     QSYS        00BD     *EXT                    *N
                                    Message . . . . :   Job 038518/JOHNDOE/QPADEV000C ended on 01/16/11 at
                                      07:13:35; 1 seconds used; end code 0 .
                                    Cause . . . . . :   Job 038518/JOHNDOE/QPADEV000C completed on 01/16/11 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 previous sample job log , 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 and auditing was not active so a message was sent to the job to notify it of the change by another user.
- If the sender is an Integrated Language Environment (ILE) procedure, additional lines are printed to identify the module, procedure, and statement number. In the previous sample job log, 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 following the message.

Related concepts

Job log sender or receiver information

The job log contains information about the sending or receiving program or procedure.

Job log sender or receiver information

The job log contains information about the sending or receiving program or procedure.

When the sender or receiver is an Integrated Language Environment (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.

Related concepts

Example: Controlling information written in a job log

This 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.

Displaying a job log

The way to display a job log depends on the status of the job.

- The **Work with Job Logs (WRKJOBLOG)** command can be used to display pending job logs for completed jobs, all job log spooled files, or both. For example, to display the list of pending job logs for all jobs that have ended, enter:

```
WRKJOBLOG JOBLOGSTT(*PENDING)
```

- If the job is active or on a job queue, or if the job log is pending, 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.

- If the job has ended and the job log is written to an output file but 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.

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 **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 a high-level language (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.

Related information

Job log pending

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.

To prevent a job log from being produced when the job is completed, but still remain in the system in a pending state, specify *PND for the LOGOUTPUT parameter on the **Submit Job (SBMJOB)**, **Change Job (CHGJOB)**, **Create Job Description (CRTJOBD)**, or **Change Job Description (CHGJOBD)** command. If you specify *NOLIST for the LOG parameter, no job log will be produced, and there will be no pending job log either. Pending job logs will only be available when a job log would normally be written to an output file or database file when the job ends and the job log output job attribute is *PND. You can use the **Work with Job Logs (WRKJOBLOG)** command to find both pending and written job logs.

Related information

Job log pending

Job log considerations

When you use job logs, consider these suggestions.

- 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.
- 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.

Related information

Operational Assistant APIs

Interactive job log considerations

When you use interactive job logs, consider these suggestions.

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 **Change Job (CHGJOB)** or **Change Job Description (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.

Batch job log considerations

When you use batch job logs, consider these suggestions.

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 **Batch Job (BCHJOB)** or **Submit Job (SBMJOB)** command. See “Job log” on page 567 for a description of logging levels.

If a batch job is running a CL program or procedure, the CL commands are 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.

Message filtering through the Control Job Log Output API

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.

Job log output files

When you use job log output files, consider the job log information you want in the files and the model files that describe the output-file record formats.

Related concepts

Writing a job log to a file

After the job ends, the job log can be written to either the output file QPJOBLOG or a database file.

Job log direction

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 database 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. After the MRK of a primary file record is known, the related secondary records can be readily identified because only these secondary records contain the same MRK value.

Primary job log model

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

Field order	Field name	Data type	Length in bytes	Field description
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
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 original program model (OPM) program or an Integrated Language Environment (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 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; 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 field 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 receiving 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.

QMHRMD
Receiving module name; CHAR(10)

- 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) 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 or immediate message; 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 are 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 receiving 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 portion 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.

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 (QHST) are written by the system to the current log version physical file. The history log contains a high-level trace of system activities such as system, subsystem, and job information, device status, and system operator messages.

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 the following is true:

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. There is also a value of *DAILY to allow a new log version to be created each day instead of creating a new file based only on size.

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 write a program to process the history log files or use IBM-supplied functions to process the files. IBM provides the **Display Log (DSPLOG) CL command** and the Open List of History Log Messages (QMHOLHST) API. Because several versions of each log might 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 **Display Log (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 before 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 **Display Log (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 log version physical file when the message queue reaches a certain size or when the **Display Log (DSPLOG)** command was used. If you want to ensure the current log version is up-to-date, specify a fictitious message identifier, such as ###0000, on the DSPLOG command. No messages are displayed, but the messages in the message queue are copied to the log version physical file to make it current.

Note: If a message is sent to QHST and the **Display Log (DSPLOG)** command is done immediately, the DSPLOG command will show messages from the log version physical file. Depending on the number of messages being copied to the log version, your message may not be shown until a second **Display Log (DSPLOG)** command is run later.

If you print the information in a log using the **Display Log (DSPLOG)** 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 (DSPLOG)** 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 (DSPLOG)** 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 message queue 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

Contents	Type	Length	Positions in record
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

Notes:

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 (includes immediate message text)	Character	Variable[1]
Message data	Character	Variable[2]

Notes:

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 log version file.

QHST file processing

If you use a high-level language (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 **Batch Job (BCHJOB)** or **Submit Job (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 **Transfer Batch Job (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 begins in position 11 of the second record. If the message identifier or the message file could not be found at the time the message entry was copied to the QHST data base file, there will only be one record in the file; there will be no second record containing the message or the message data.
  - 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 hexadecimal 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.

QHST files deletion

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.

QSYSMSG message queue

QSYSMSG is a queue that you can create if you plan to handle the list of serious messages that are sent to the 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')
```

After the QSYSMSG message queue is created, all of the specific messages 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

Messages that are sent to the QSYSMSG message queue can be categorized into these groups: messages sent to the QSYSMSG message queue instead of QSYSOPR; messages sent to QSYSMSG, QSYSOPR, or both; and messages always sent to both QSYSMSG and QSYSOPR.

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 QSYSMSG, to QSYSOPR, or to both QSYSMSG and 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.

CPF097F
Load source disk unit not in valid location.

This message is sent when the load source disk unit is not in a valid location to allow the system to load the initial program.

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. 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 switch 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.

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 about 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 the IBM i operating system. 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).

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

`i5/OS grace period expired.`

The software license grace period for IBM i 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 i5/OS 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 previous disk unit information 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 switch off 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 switch off 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 switch 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.

CPI116F
`Hot spare disk unit not in valid location.`

This message is sent when the load source disk unit fails and a hot spare disk unit that protects the load source disk unit is not in a valid location to allow the system to load the initial program.

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.

CPI1163
No hot spare disk units remain on I/O adapter.

This message is sent when all hot spare disk units on an I/O adapter have been consumed or have failed. No data has been lost.

CPI1164
Hot spare disk unit not operating.

This message is sent when a hot spare disk unit has stopped operating. No data has been lost.

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.

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 before 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 installation, 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 programs have to send 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.

This message is sent when 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.
```

Related information

Backup and recovery

Communications Management PDF

Finance Communications Programming PDF

Example: Receiving messages from QSYSMSG

This example program 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 **End Mode (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 to indicate 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 **End Job (ENDJOB)** command.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 610.

```
/*********************************************************************/
/*                                                                   */
/*  Example 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
```

Code license and disclaimer information

IBM grants you a nonexclusive copyright license to use all programming code examples from which you can generate similar function tailored to your own specific needs.

SUBJECT TO ANY STATUTORY WARRANTIES WHICH CANNOT BE EXCLUDED, IBM, ITS PROGRAM DEVELOPERS AND SUPPLIERS MAKE NO WARRANTIES OR CONDITIONS EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OR CONDITIONS OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT, REGARDING THE PROGRAM OR TECHNICAL SUPPORT, IF ANY.

UNDER NO CIRCUMSTANCES IS IBM, ITS PROGRAM DEVELOPERS OR SUPPLIERS LIABLE FOR ANY OF THE FOLLOWING, EVEN IF INFORMED OF THEIR POSSIBILITY:

1. LOSS OF, OR DAMAGE TO, DATA;
2. DIRECT, SPECIAL, INCIDENTAL, OR INDIRECT DAMAGES, OR FOR ANY ECONOMIC CONSEQUENTIAL DAMAGES; OR
3. LOST PROFITS, BUSINESS, REVENUE, GOODWILL, OR ANTICIPATED SAVINGS.

SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OR LIMITATION OF DIRECT, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, SO SOME OR ALL OF THE ABOVE LIMITATIONS OR EXCLUSIONS MAY NOT APPLY TO YOU.

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
1623-14, Shimotsuruma, Yamato-shi
Kanagawa 242-8502 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.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Software Interoperability Coordinator, Department YBWA
3605 Highway 52 N
Rochester, MN 55901
U.S.A.

© Copyright IBM Corp. 1998, 2013

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

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

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

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_.

Programming interface information

This CL overview and concepts publication documents intended Programming Interfaces that allow the customer to write programs to obtain the services of IBM i.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at www.ibm.com/legal/copytrade.shtml.

Adobe, the Adobe logo, PostScript, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java and all Java-based trademarks and logos are trademarks of Oracle, Inc. in the United States, other countries, or both.

Other product and service names might be trademarks of IBM or other companies.

Terms and conditions

Permissions for the use of these publications is granted subject to the following terms and conditions.

Personal Use: You may reproduce these publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative works of these publications, or any portion thereof, without the express consent of IBM.

Commercial Use: You may reproduce, distribute and display these publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these publications, or reproduce, distribute or display these publications or any portion thereof outside your enterprise, without the express consent of IBM.

Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the publications or any information, data, software or other intellectual property contained therein.

IBM reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the publications is detrimental to its interest or, as determined by IBM, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations.

IBM MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.

Product Number: 5770-SS1