IBM

Application System/400™

SC21-8077-2

Programming: Control Language Programmer's Guide

IBM

Application System/400™

SC21-8077-2

Programming: Control Language Programmer's Guide

Third Edition (August 1990)

This edition applies to the licensed program IBM AS/400 Operating System/400 (Program 5728-SS1), Release 3 Modification 0, and to all subsequent releases and modifications until otherwise indicated in new editions. Make sure you are using the proper edition for the level of the product

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

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

IBM Corporation, Department 245, 3605 North Highway 52, Rochester, MN 55901-9986 USA.

Special Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates.

Any reference to an IBM licensed program or other IBM product in this publication is not intended to state or imply that only IBM's program or other product may be used.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Commercial Relations, IBM Corporation, Purchase, NY 10577.

The following terms, denoted by an asterisk (*) in this publication, are trademarks of the IBM Corporation in the United States and/or other countries:

Application System/400	AS/400	IBM
Operating System/400	OS/400	Question and Answer
RPG/400	Total System Package	400

This publication could contain technical inaccuracies or typographical errors.

About This Guide

This guide provides a wide-range discussion of AS/400 programming topics, including:

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

This guide may refer to products that are announced but not yet available.

This guide contains small programs which are furnished by IBM as simple examples to provide an illustration. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. All programs contained herein are provided to you "AS IS". THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE EXPRESSLY DISCLAIMED.

This guide follows the convention that he means he or she.

Who Should Use This Guide

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

What You Should Know

Before using this guide, you should be familiar with general programming concepts and terminology, and have a general understanding of Operating System/400 (OS/400) and the AS/400 system. You should also be familiar with the work stations you are using.

How This Guide Is Organized

This guide is organized as follows:

Chapter 1 provides a brief introduction to each of the topics discussed in detail in the following chapters.

Chapter 2 introduces CL programming, what CL programs can be used for, how they are arranged, and how they are controlled.

Chapter 3 describes the commands that control the flow of processing from one CL program to another, or to programs written in other high-level languages.

Data queues and data areas are discussed as objects that can be used to pass or share data between high-level language programs.

Chapter 4 discusses objects and libraries, including:

- Descriptions of the various types of objects and the operations that can be performed on them.
- The use of libraries and library lists to organize and access objects.

Chapter 5 discusses object qualification and accessing of specific object types in CL programs.

Chapter 6 introduces more advanced programming topics, including:

- Special functions that can be called from high-level language programs (including CL programs).
- Using prompting and the Programmer Menu to enter program source.

Several sample programs are included at the end of the chapter.

Chapters 7 and 8 discuss AS/400 message handling, including creating message files, changing message descriptions, retrieving and monitoring for messages in programs.

Chapter 9 is a detailed discussion of user-defined commands, including several examples of defining and creating commands.

Chapter 10 provides a general description of the IBM-supplied menu function and a more detailed description of two types of user-defined menus: display file menus and program menus. Examples of creating the user-defined menu types are provided.

Chapter 11 provides an overview of the testing functions available on the AS/400 system, including discussions of debug mode, breakpoints, traces, and display functions.

In the back of this manual is a glossary and index. Use the glossary to find the meaning of an unfamiliar term. Use the index to look up a topic, and to see on which pages the topic covered.

How This Guide Has Changed

Changes since the previous edition of this guide are indicated by a vertical line and a plus sign to the left of the change.

More messages for QSYSMSG have been added for this release. REXX support for creating CL commands, the use of the prompt override program, and an explanation of usage data for deleting objects are also covered in this manual.

Related Online Information

The following online information is available on the AS/400 system. After pressing the Help key on any menu, press the Help key a second time to see an explanation of how the online information works, including the index search function. You can press either the Help key or F1 for help.

Help for Displays

You can press the Help key on any display to see information about the display. There are two types of help available:

Contextual
Extended

Contextual help explains the field on which the cursor is positioned when you press the Help key. For example, it describes the choices available for a prompt. If a system message appears at the bottom of the display, position the cursor on the message and press the Help key to see information about the cause of the message and the appropriate action to take.

Extended help explains the purpose of the display. Extended help appears if you press the Help key when the cursor is outside the areas for which contextual help is available, or if you press F2 (Extended help) when you are looking at the contextual help.

To exit the online information, press F3 (Exit). You return to the display where you pressed the Help key.

Index Search

Index search allows you to specify words or phrases that identify the information that you want to see. To use index search, press the Help key, then press F11 (Search index). You can also use index search by entering the Start Index Search (STRIDXSCH) command on any command line or by selecting option 2 on the User Support and Education menu.

To access index search for AS/400 languages that have it available, you must be using source entry utility (SEU) to edit source statements. When the cursor is in the data entry area, press Help and then F11 (Search index). To access index search for Application Development Tools when you are using SEU, make sure your cursor is not in the data area, press Help, and then F11 (Search index).

Online Education

AS/400 online education provides training on a wide variety of topics. To use the online education, select option 10 (User support and education) from the main menu or press F13 (User support) on any system menu to show the User Support and Education menu. Then select the option to use online education.

Question-and-Answer Function

The question-and-answer (Q & A) function provides answers to questions you may have about using the AS/400 system. To use the Q & A function, press F13 (User support) on any system menu to show the User Support and Education menu. Then select the option to use the Q & A function. You can also use the Q & A function by entering the Start Question and Answer (STRQST) command on any command line.

Help for Control Language Commands

To see prompts for parameters for a control language command, type the command, then press F4 (Prompt) instead of the Enter key. To see extended help for the command, type the command and press the Help key. You can see contextual help for CL commands also.

Related Printed Information

You may want to refer to other AS/400 manuals for more specific information about a particular topic. The *System Introduction*, GC21-9766, provides an introduction to the system characteristics and licensed programs available for the AS/400 system.

The following is a list of the additional manuals and the information to be found in them.

For information about operating the AS/400 system and its display stations, see:

- *System Operations: New User's Guide*, GC21-8211

  This guide provides general information about using display stations, including function keys, display, command, and help information.

- *System Operations: Operator's Guide*, SC21-8082

  This guide provides general information about how to run the system, how to send and receive messages and use the display station function keys.

For more information about AS/400 programming, see:

- *Programming: Backup and Recovery Guide*, SC21-8079

  This guide provides information about the different media available to save and protect system data.

- *Programming: Database Guide*, SC21-9659

  This guide provides a detailed discussion of the AS/400 database structure, including information on how to create, describe, and manipulate database files.

- *Programming: Data Management Guide*, SC21-9658

  This guide provides information about file management, device files, spooling, job and output queues.

- *Programming: Security Concepts and Planning*, SC21-8083

  This manual discusses general security concepts and planning for security on the system. It also includes information for all users about resource security.

- *Programming: Work Management Guide*, SC21-8078

  This guide provides information about creating and changing the work management environment, working with system values, collecting and using performance data to improve system performance.

- *National Language Support: User's Guide*, GC21-9877

  This guide provides the data processing manager, system operator and manager, application programmer, end user, IBM marketing representative, and system engineer with information required to understand and use the national language support function on the AS/400 system. This manual prepares the AS/400 user for planning, installing, configuring, and using AS/400 national language support (NLS) and multilingual system. It also provides an explanation of the database management of multilingual data and application considerations for a multilingual system.

For detailed information about CL commands, see:

- *Programming: Command Reference Summary*, SC21-8076

  This manual provides quick reference information about the structure of the AS/400 commands, including syntax diagrams and error messages that can be monitored.

- *Programming: Control Language Reference*, SBOF-0481

  This manual provides a description of the AS/400 control language (CL) commands. Each command is described, including its syntax diagram, parameters, default values, and keywords.

- *Programming: System Reference Summary*, SC21-8104

  This manual provides reference information, including summary charts, system values, and DDS keywords for the AS/400 system.

For more information about AS/400 utilities mentioned in this manual, see:

- *Programming: Data Description Specifications Reference*, SC21-9620

  This manual provides a detailed description of the entries and keywords needed to externally describe database files and certain device files.

- *Application Development Tools: Character Generator Utility User's Guide*, SC09-1170

  This guide provides information about using the character generator utility (CGU) to create and maintain a double-byte character set on the AS/400 system.

- *Application Development Tools: Programming Development Manager User's Guide and Reference*, SC09-1173

  This manual provides information about using the programming development manager (PDM) to work with lists of libraries, objects, members, and user-defined options.

- *Application Development Tools: Screen Design Aid User's Guide and Reference*, SC09-1171

  This manual provides information about using the screen design aid (SDA) to design, create, and maintain display formats and menus.

- *Application Development Tools: Source Entry Utility User's Guide and Reference*, SC09-1172

  This manual provides information about using the source entry utility (SEU) to create and edit source members.

For more information about alerts and problem reporting, see:

Communications: Communications and Systems Management User's Guide, SC21-9661

This guide provides information about using the remote management support (DHCF), change management support (DSNX), and problem management support (alerts).

Contents

Chapter 1. Introduction

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

System operation is controlled by the following:

- CL commands. CL commands are used singly in batch and interactive jobs, (such as from the Command Entry display) and in CL programs.
- Menu options. System operation can be controlled by selecting menu options. Interactive users can use the AS/400* menus to perform many system tasks.
- System messages. System messages are used to communicate between programs and to communicate between programs and users. Messages can report both status information and error conditions.

Control Language

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

- Individually from a work station
- As part of batch jobs
- As source statements to create a CL program.

Commands can be entered individually from any command line or the Command Entry display. For information about the Command Entry display, see "Interactive Entry" on page 2-3.

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.

Command Syntax

Each command is made up of a command name and parameters. A command name usually consists of a verb, or action, followed by a noun or phrase that identifies the receiver of the action. The words that make up the command name are abbreviated, usually to three letters, to reduce the amount of typing required to enter the command. For example, one of the CL commands is the Send Message command. The command name is SNDMSG, used to send a message from a user to a message queue.

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

The following display shows the SNDMSG command entered on the Command Entry display and identifies the parts of the command.

```
                               Command Entry                       SYSTEM1
                                                           Request level:   1
Previous commands and messages:

  (No previous commands or messages)










                                                                     Bottom
Type command, press Enter.
===> SNDMSG      MSG('Good morning')    TOUSR(BOB2)
        1          2        3                4


F3=Exit   F4=Prompt   F9=Retrieve   F10=Include detailed messages
 F11=Display full     F12=Cancel    F13=User support   F16=System main menu
```

1 Command name. The SNDMSG command is constructed like most CL commands where the first part tells what action is taken and the second part tells what thing receives the action. Here, SND is the abbreviation used for *send*; MSG is used to abbreviate *message*, the thing that is sent.

At least one blank must separate the command name from the following parameter name or value.

2 Keyword. The MSG keyword states that the MSG (message) parameter is being specified.

3 Parameter value. The parameter value specified for the MSG parameter is 'Good morning'. Blanks may not be used between the keyword (or parameter name) and the left parenthesis of the parameter value.

4 The last specified parameter, TOUSR, instructs the system to send the message (Good morning) to a user profile named BOB2. BOB2 is specified as a value for the TOUSR (To user) parameter.

Note: To see the full-screen Command Entry display, which allows more room for entering commands, press F11=Display full. To see all detailed messages from previously run commands, press F10=Include detailed messages.

For more information about entering commands interactively from the Command Entry display, see "Interactive Entry" on page 2-3. The *CL Reference* discusses CL command syntax in detail, and contains descriptions of all CL commands.

Command Prompting

The operating system provides interactive command prompting for any command supplied with the system or created by the user. Type the command name only, then press a function key to prompt for the command. The display provides one of the following:

- A list of all required or often-used parameters
- The name of the object being operated on and its qualifiers (such as the library for a file) that uniquely identify the object.

Based on the values specified for the parameters, the user may be prompted for additional information. This conditional prompting does the following:

- Limits the information requested to those parameters required to perform the function
- Allows you to tailor the values returned for the remaining prompts based on the key parameter values specified.

Note: If you specify the values for the key parameter when you type the command, the values that appear for the remaining prompts are tailored for the key parameter values.

The display may show either a list of possible values or keyword names; F11 can be used to switch between the two display options.

Online help information is available for IBM-supplied commands by pressing the Help key.

The following display shows the prompts for parameters on the SNDMSG command.

```
                         Send Message (SNDMSG) 1

 Type choices, press Enter.

 Message text . . . . . . . . . .   ____________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
____________________________________________________________
 To user profile . . . . . . . .   ________   Name, *SYSOPR, *ALLACT... 2






                                                                  Bottom
F3=Exit   F4=Prompt   F5=Refresh   F10=Additional parameters   F12=Cancel
F13=How to use this display        F24=More keys 3
```

1 Command name

2 Possible values for the To User (TOUSR) parameter

3 Function keys.

If some command parameters are specified before prompting, those values already specified are shown after prompting.

Most of the parameters used on commands have default values that are used if no other value is specified. The default value can be explicitly entered if the user desires.

CL Programs

CL programs are made up of CL commands. The commands are compiled into a program that can be called whenever the functions provided by the program are needed. Advantages of using CL programs include:

- Because the commands are compiled and stored in a form that can be run immediately, using CL programs is faster than entering and running the commands individually.
- Some functions require CL commands that cannot be entered individually and must be part of a CL program.
- CL programs can be tested and debugged like other high-level language (HLL) programs.
- Parameters can be passed to CL programs to adapt the operations performed by the program to the particular requirements of that use.

CL programs can be used for many kinds of applications. For example, CL programs 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. This makes the work station user's job easier and reduces the chances of errors occurring when commands are entered.
- Control the operation of a batch 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 program is run.
- Provide predefined procedures for the system operator, such as procedures to start a subsystem, to provide backup copies of files, or to perform any other procedural operating functions. The use of CL programs to perform these 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. In addition, some functions designed for use in CL programs are not available when commands are entered individually. These functions include:

- Logic control functions that can be used to control which operations are performed by the program according to conditions that exist when the program 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.
- Data operations that provide a way for the program to communicate with a work station user. Data operations let the program send formatted data to

and receive data from the work station, and allow limited access to the data-base.

- Functions that allow the program to send messages to the display station user.
- Functions that receive messages sent by other programs. These messages can provide normal communication between programs, or indicate that errors or other exceptional conditions exist.
- The use of variables and parameters for passing information between commands in the program and between programs.

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

- Determine which programs 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 provide the flexibility needed to let the application user select what operations to perform and run the necessary programs.

CL programming is discussed in detail in Chapter 2, "CL Programming." More advanced uses of CL are discussed in Chapter 3, "Controlling Flow and Communicating between Programs," and Chapter 6, "Advanced Programming Topics."

Command Definition

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

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

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

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

The command definition functions can be used to:

- Create unique commands needed by system users while keeping a consistent interface for CL command users.
- Define alternative versions of CL commands to meet the requirements of system users. This function might include having different defaults for parameter values, or simplifying the commands so that some parameters

would not need to be entered. Constant values can be defined for those parameters. The IBM-supplied commands should not be changed.

See Chapter 9, "Defining Commands," for a detailed discussion of command definition.

Menus

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

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

The following example shows the AS/400 Main Menu.

```
MAIN 1                           AS/400 Main Menu
                                                                System   XXXXXXXX
Select one of the following:

      1. User tasks
      2. Office tasks
      3. General system tasks
      4. Files, libraries, and folders
  2   5. Programming
      6. Communications
      7. Define or change the system
      8. Problem handling
      9. Display a menu
     10. User support and education
     11. PC Support tasks

     90. Sign off

Selection or Command
===> 3 ____________________________________________________________________

F3=Exit   F4=Prompt   F9=Retrieve   F12=Cancel   F13=User support
F23=Set initial menu  4
(C) COPYRIGHT IBM CORP. 1980, 1990.
```

1 The menu name. Here, the menu name is MAIN. All system menus can be run using the Go to Menu (GO) command. For example, to see the Programming menu (menu name PROGRAM), type:

```
GO PROGRAM
```

on the command line, then press the Enter key.

2 Menu options. You can select any displayed menu options by typing the number of the option on the command line and pressing the Enter key.

3 Selection or command line. You can select menu options or enter commands by typing them on this line and pressing the Enter key. (The second line may be used to continue typing long commands.)

4 Function keys. These keys provide an easy way for you to perform such functions as returning to the previously shown display (F12=Cancel) or retrieving a command typed on the command line (F9=Retrieve).

The *New User's Guide* provides detailed descriptions of system menus and information about how to use these menus.

Just as users can define commands to meet their special application requirements, users can also create menus to simplify their tasks.

Menus can be used to group jobs by application; for example, all accounts receivable jobs could be listed on one menu, all order entry jobs on another menu, and all inventory management jobs on a third menu. Such grouping keeps related jobs together and allows users to run several related jobs one after the other.

Procedures for creating menus that can be used like the system-supplied menus are described in Chapter 10, "User-Defined Menus."

Objects and Libraries

An **object** is a named element made up of a set of attributes that describe the object and a value. The attributes of an object include its name, type, size, the date it was created, and a description provided by the user who created the object. The value of an object is the collection of information stored in the object. The value of a program, for example, is the code that makes up the program. The value of a file is the collection of records that makes up the file. The concept of an object simply provides a term that can be used to refer to a number of different items that can be stored in the system, regardless of what the items are.

Objects

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

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

- Files
- Programs
- Commands
- Libraries
- Queues.

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.

Libraries

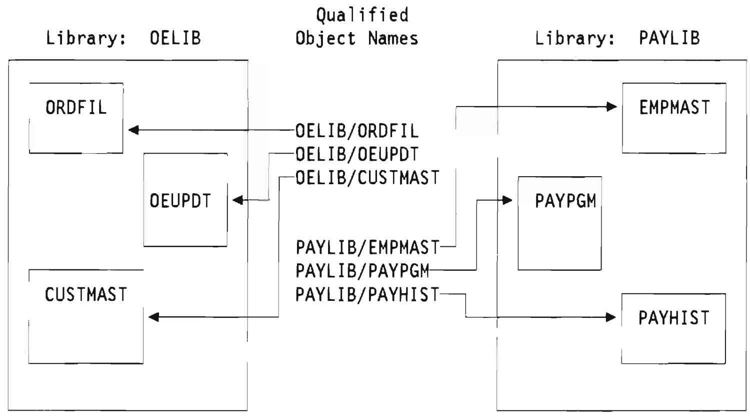
A **library** is an object used to group related objects, and to find objects by name when they are used. Thus, a library is a directory to a group of objects. Libraries can be used to group the objects into any meaningful collection. For example, objects can be grouped according to security requirements, backup requirements, or processing requirements. The number of objects contained in a library and the number of libraries on the system are limited only by the amount of storage available, but should be limited to less than 8000 to ensure that save operations can be performed. The number of libraries on the system can be tailored to the way the objects are used.

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) 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. If an object increases in size, the system automatically allocates additional storage to the object.

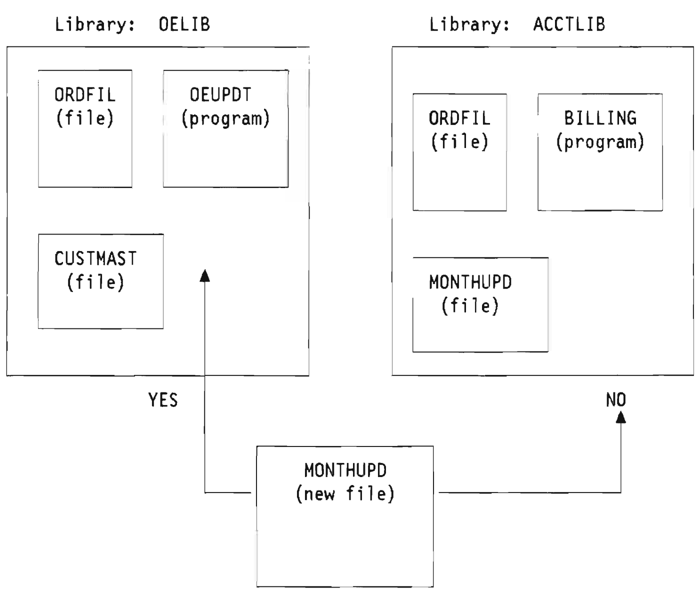
Most types of objects are placed in a library when they are created. Most object types can be moved from one library to another, but a single object cannot be in more than one library at the same time. When an object is moved to a different library, the object is not moved in storage, but it is located through the new library. Most object types can also be renamed and copied from one library into another.

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

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



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



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

Using Libraries to Find Objects

An object name can be specified as a qualified name (where both the library name and object name are specified) or only the object name may be specified. If a qualified name is specified, the system tries to find the object in the specified library. If only the object name is specified, the system searches a list of libraries, called the *library list*, until it finds the first occurrence of the object of the correct name and type or until it has searched all the libraries on the list without finding the object. The libraries that are searched, and the order in which they are searched, is determined by a search list called the library list. The system creates an initial library list for each job when the job is started.

A library list has four parts. The first part is the *system part* of the library list. This part specifies the libraries used for all the jobs that run on the system. When the system is installed, the system part of the library list consists of the system library (QSYS), the system library for user data (QUSRSYS), and the system help information (QHLPSYS). Libraries in the system part of the library list are searched before the remaining libraries in the list.

The second part of the library list is the *product library* part. The product library part is changed by the system as users run commands or menus to include the library where the product objects are stored. The product library will vary while a job is running, based on the function being performed.

The third part of the library list is the *current library*. The current library is the default library used for creating objects. Users can specify the current library for a job using CL commands.

The fourth part of the library list is the *user part*. The user part contains the libraries used by application programs to perform their functions. When the system is installed, this part contains the general-purpose library (QGPL) and the job's temporary library (QTEMP). Each job has its own QTEMP, which is not visible to other jobs. QTEMP is cleared when a job ends.

When a system has a number of user-defined libraries, the user part of the library list may vary between different jobs. For example, for an order entry job, the user part of the library list might include:

- OELIB (order entry library)
- QGPL (general-purpose library)
- QTEMP (job's temporary library)

The following diagram shows an example of a library list:

Search order

QSYS QUSRSYS QHLPSYS	System Part
QPDA	Product Library 1
QRPG	Product Library 2
OELIB	Current Library
OELIB QGPL QTEMP	User Part

Specifying only the object name and using the library list to search for the object can make using the AS/400 system easier and more flexible. A library list can be designed for each job to ensure that the correct objects will be located without using the qualified names. This approach provides advantages such as:

- Easier testing of application programs. Libraries can be created to contain sample data when programs are tested. The object names used in the library are the same as those used in the normal production library. The library with the testing objects is placed before the normal production library in the library list. When the program has been fully tested, that library can be removed from the library list. The program then operates on objects contained in the production libraries, and the object names do not need to be changed in the program.
- Flexible use of the libraries on the system. As processing needs change, existing libraries may need to be divided into more than one library to help simplify the organization of objects on the system. This change does not require the names of objects in the programs to be changed. Only the library lists used by the jobs need to be changed.
- The ability to let different system users operate on different objects using the same application program. Separate libraries can be created for each user or group of users. The library list for each user's job ensures that the correct objects are used by the program for each system user.

Because of these advantages, qualified names are not usually specified when existing objects are used. However, the qualified name can be specified in situations where it is more efficient than changing the library list, or where specific objects should be specified to ensure that the correct object is used.

See Chapter 4, "Objects and Libraries," for more information about objects, libraries, and library lists. Chapter 5, "Working with Objects in CL Programs," contains more information about using objects.

Messages

A **message** is a communication sent from one user or program to another. Most data processing systems provide communications between the system and the system operator to handle errors and other conditions that occur during processing. In addition to this type of support, OS/400 provides message handling functions that support two-way communications between programs and system users, between programs, 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 and between programs and users, using the OS/400 message handling functions should be considered when developing applications. The following concepts of message handling are important to application development:

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

Because replies can be returned by a user or a program that receives a message, the message handling function allows for two-way communications.

Message Descriptions

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

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

Message Queues

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

OS/400 provides message queues for:

- Each work station 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 retained, so the receiver of the message does not need to process the message immediately. A message queue can be used as a mailbox to hold the messages until the program or user decides to receive them.

See Chapter 7, "Defining Messages," and Chapter 8, "Working with Messages," for more information about messages.

Testing Functions

The system includes functions that let a programmer observe operations performed as a program runs. These functions can be used to locate operations in the program that are not performing as intended. Testing functions can be used in either batch or interactive jobs from a work station. 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 program's source statements. Often, an error is apparent only because the output produced is not what is expected. To find those errors, a programmer needs 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. The programmer might want to make changes to those variables before letting the program continue running.

The programmer does not need to know machine language instructions, nor is there a need to include special instructions in the program to use the testing functions. The OS/400 testing functions let the programmer:

- Stop a running program at any named point in the program's source statements.
- Display information about program variables at any point where the program can be stopped. The programmer can also change the variable information before continuing program processing.
- Trace the use of variables in the program by recording the steps in the program that change the variables, and what those changes are. This operation produces a list or display tracing the sequence of program statements run and the values of variables at any point in the program.

See Chapter 11, "Testing Functions," for more information about testing and debugging programs.

Chapter 2. CL Programming

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

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, the source remains for later modification if you wish, and the commands are compiled together as a program. This program remains as a permanent system object that can be rerun whenever you call it. Thus, CL is actually a high-level programming language for system functions. You can perform functions with a CL program that you cannot perform by entering commands individually, and the CL program provides better performance at run time than the processing of several separate commands. CL programs also ensure consistent processing of groups of commands.

CL programs can be used in batch or interactive processing, depending on the commands used in the CL.

CL source statements consist of CL commands. Not all CL can be used as CL source statements, and some of them can be used only in CL programs. You can determine what restrictions are placed on the use of CL commands by checking the box in the lower right-hand corner of the syntax diagram of a command in the *CL Reference*, such as the Program (PGM) command shown below:

```
                                                        Optional
PGM──PARM─┬CL-variable-name┬────────────────────────────────
          ↑                │
          └──40 maximum────┘

                                               ┌─────────┐
                                               │Pgm: B,I │
                                               └─────────┘
```

The syntax diagram for the PGM command shows that this command can be used in either batch or interactive jobs, but can only be used within a CL program.

CL source statements can be entered in a database source member either interactively from a work station or in a batch job input stream from a device. For either method, you must enter the source and then create a permanent object before you can run the CL program.

CL programs can be written for many purposes, including:

- To control the sequence of processing within a program and the calling of other programs.
- To display a menu and run commands based on options selected from that menu. This makes the work station user's job easier and reduces errors.
- To read a single-format database file.

- To handle error conditions issued from commands or programs 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 procedures 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.

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

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

You cannot use CL programs to:

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

Creating a CL Program

All CL programs are created in two steps:

1. Source creation. CL programs consist of CL commands. In most cases, source statements are entered into a database file in the logical sequence determined by your application design.
2. Program creation. Using the Create Control Language Program (CRTCLPGM) command, this source is used to create a system object. The created CL program can be run immediately using the CALL command.

Batch Entry

You can perform both source creation and program creation in one batch input stream from diskette. The following example shows the basic parts of the input stream from a diskette device. The input is submitted to a job queue using the Submit Diskette Job (SBMDKTJOB) command. The input stream should follow this format:

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

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

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

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

```
OVRDKTF FILE(QDKTSRC) LABEL(PGMA)
```

Now you can consider the QDKTSRC file as the source file on the CRTCLPGM command. To create the CL program based on the source input from the diskette, enter the following command:

```
CRTCLPGM PGM(QGPL/PGMA) SRCFILE(QDKTSRC)
```

When the CRTCLPGM command is processed, it treats the QDKTSRC source file like any database source file. Using the override, the source is located on diskette. PGMA is created in QGPL, and the source for that program remains on diskette. See the *Data Management Guide* for more information on device files.

Interactive Entry

The AS/400 system provides many menus and displays to assist the programmer, including the Programmer Menu, the Command Entry display, and command prompt displays. If your AS/400 system uses the security functions described in *Security Concepts and Planning*, 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.

The Command Entry Display

The Command Entry display has a partial-screen version and a full-screen version. F11 acts as a toggle key between the two versions.

To see the Command Entry display, enter CALL QCMD on any command line or specify QCMD as the initial program in a user profile. The following partial-screen version shows the parts of the Command Entry display:

```
                            Command Entry                          SYSTEM1
                                                     1  Request level:   1
Previous commands and messages:

  (No previous commands or messages)
  2








                                                                    Bottom
Type command, press Enter.
===> 3 ____________________________________________________________________
___________________________________________________________________________
___________________________________________________________________________

 F3=Exit   F4=Prompt   F9=Retrieve   F10=Include detailed messages
 F11=Display full      F12=Cancel    F13=Support menu    F16=System main menu
```

1 Request level. This number specifies the current level of the request processing program. A request processing program processes commands. QCMD is the request processing program for the Command Entry display. Each time a request processing program is called, the request level increases. When the program is ended the request level decreases.

2 History area. This area, available only on the partial-screen version, shows messages and previously run commands. The most recent messages and commands are shown at the bottom of the history area.

You can show detailed messages in the history area or omit them. The command line is displayed and usable whether or not detailed messages are shown in the history area. To see detailed messages, press F10=Include detailed messages. To remove detailed messages from the history area, press F10 again.

Commands can be retrieved from the history area. You can also prompt for commands directly from the history area. The online help information for the Command Entry display describes these functions.

Note: Comment requests (text preceded by a /* and followed by a */) can only be retrieved by positioning the cursor to the comment request and then pressing F9.

3 Command line. The partial-screen version provides four lines for command entry; the full-screen version has 17 lines.

Note: Ellipses (...) are shown at the end of the command line when you retrieve a command string that is longer than the command line. The command can be run or prompted with the ellipses, but if changes are made, you may receive an error.

Command Prompt Displays

When you type a command name on a command line and press F4=Prompt, a command prompt display is shown. To display help information about how to enter parameter values on a command prompt display, press F13=Prompter help.

Parts of a CL Program

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

PGM command

`PGM PARM(&A)`

Optional PGM command beginning the program and identifying any parameters received. The PGM command is valid only within a CL program.

Declare commands

`(DCL, DCLF)`

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

CL processing commands

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

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

Logic control commands

`IF, THEN, ELSE, DO, ENDDO, GOTO`

Commands used to control processing within the CL program.

Built-in functions

`%SUBSTRING (%SST) and %SWITCH`

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

Program control commands

`CALL, RETURN, TFRCTL`

CL commands used to pass control to other programs.

ENDPGM command

`ENDPGM`

Optional End Program command. The ENDPGM command is valid only within a CL program.

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

A CL program may refer to other objects, which must exist either when the program is created or when the command is processed, or both. This distinction is discussed in "Accessing Objects in CL Programs" on page 5-1, and in the sections discussing various objects. In some circumstances, for your program to run successfully, you may need:

- A display file. Display files are used to format information on a device display. If your program uses a display, the display file and record format must be entered and created using the Create Display File (CRTDSPF) command before the program is created. It must also be declared to the program in the DCL section using the Declare File (DCLF) command. See "Working with Files in CL Programs" on page 5-4 and the *Data Management Guide* for more information.
- A database file. Records in a database file may be read by a CL program. If your program 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 program 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 program in the DCL section using the Declare File (DCLF) command. See "Working with Files in CL Programs" on page 5-4 and the *Database Guide* for more information.
- Other programs. If you use a CALL command, the called program must exist before the CALL command is processed. It does not have to exist when the calling program is compiled. See "Accessing Objects in CL Programs" on page 5-1 and Chapter 3 for more information.

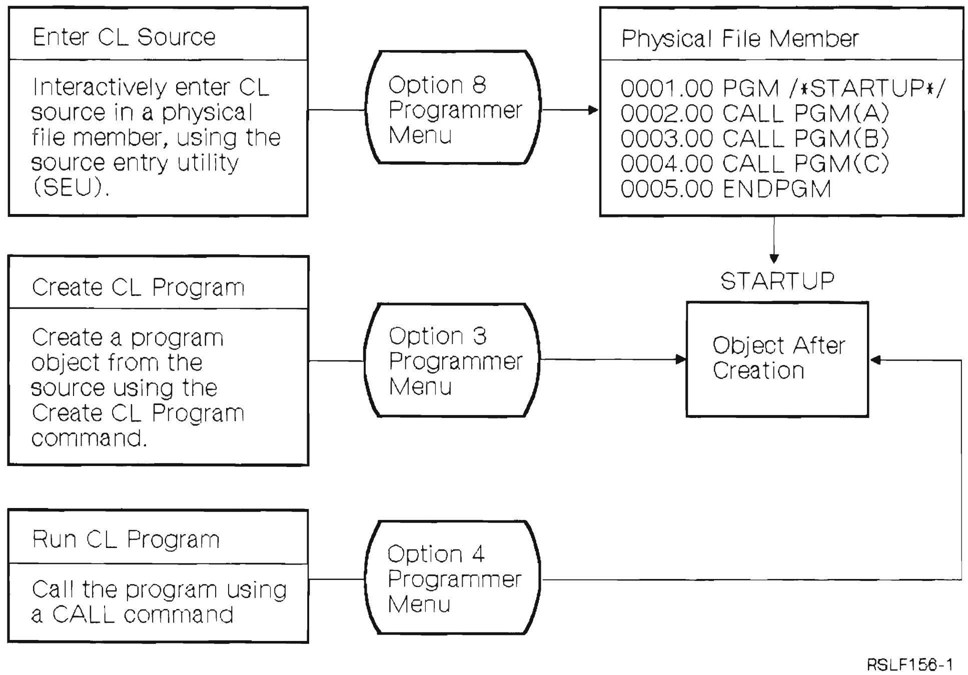
Example of a Simple CL Program

A CL program can be as simple or as complex as you wish. 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 program STARTUP with the following code:

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

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

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



To enter CL source:

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

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

 F3=Exit                F4=Prompt                       F5=Refresh
 F10=Top                F11=Bottom                      F23=More keys
                                              (C) COPYRIGHT IBM CORP. 1988.  +
```

When you have finished entering the source statements:

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

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

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

These are the basic steps required to enter, create, and run a simple CL program.

Commands Used in CL Programs

A CL program can contain only CL commands. These can be IBM-supplied or commands defined by you. All IBM-supplied commands are described in the *CL Reference*.

You can determine what restrictions are placed on CL commands by checking the box in the lower right-hand corner of the syntax diagram of the command in the *CL Reference*. Commands that can only be used as source statements in CL programs will have only `Pgm:` in the box. If the box does not contain this indicator, the command cannot be used as source for a CL program. For specific information about interpreting syntax diagrams, see the *CL Reference*.

Some commands *cannot* be used in CL programs. Refer to the *CL Reference* for the individual command descriptions and their applicability in CL programs.

Commands Entered on the RQSDTA and CMD Parameters

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

CL Commands

The following is a list of commands frequently used in CL programs. You can use this list to select the appropriate command for the function you want, and to determine which command you might need to refer to in the *CL Reference*. Familiarity with the function of these commands will help you to understand subsequent topics in this chapter. Asterisks indicate commands that can be used *only* in CL programs.

System Function	Command	Command Function
Changing Program Control	CALL (Call)	Calls a program
	RETURN (Return)	Returns to the command following the command that caused a program to be run (called the program)
	TFRCTL (Transfer Control) *	Transfers control to a program
CL Program Limits	PGM (Program) *	Indicates the start of CL program source
	ENDPGM (End Program) *	Indicates the end of CL program source
CL Program Logic	IF (If) *	Processes commands based on the value of a logical expression
	ELSE (Else) *	Defines the action to be taken for the else (false) condition of an IF command
	DO (Do) *	Indicates the start of a Do group
	ENDDO (End Do) *	Indicates the end of a Do group
	GOTO (Go To) *	Branches to another command
CL Program Variables	CHGVAR (Change Variable) *	Changes the value of a CL program variable
	DCL (Declare) *	Declares a program variable in a program
Conversion	CHGVAR (Change Variable) *	Changes the value of a CL program variable
	CVTDAT (Convert Date) *	Changes the format of a date used in a program
Data Areas	CHGDTAARA (Change Data Area)	Changes a data area
	CRTDTAARA (Create Data Area)	Creates a data area
	DLTDTAARA (Delete Data Area)	Deletes a data area
	DSPDTAARA (Display Data Area)	Displays a data area
	RTVDTAARA (Retrieve Data Area)	Copies the content of a data area to a CL variable
Files	ENDRCV (End Receive) *	Cancels a request for input previously issued by a RCVF, SNDF, or SNDRCVF command to a display file
	DCLF (Declare File) *	Declares a display or database file to a program
	RCVF (Receive File) *	Reads a record from a display or database file to a program
	RTVMBRD (Retrieve Member Description) *	Retrieves a description of a specific member of a database file
	SNDF (Send File) *	Writes a record to a display file

System Function	Command	Command Function
	SNDRCVF (Send/Receive File) *	Writes a record to a display file and reads that record after the user has replied
	WAIT (Wait) *	Waits for data to be received from an SNDF, RCVF, or SNDRCVF command issued to a display file
Messages	MONMSG (Monitor Message) *	Monitors for escape, status, and notify messages sent to a program's message queue
	RCVMSG (Receive Message) *	Copies a message from a message queue into CL variables in a CL program
	RMVMSG (Remove Message) *	Removes a specified message from a specified message queue
	RTVMSG (Retrieve Message) *	Copies a predefined message from a message file into CL program variables
	SNDPGMMSG (Send Program * Message)	Sends a program message to a message queue
	SNDRPY (Send Reply) *	Sends a reply message to the sender of an inquiry message
	SNDUSRMSG (Send User Message)	Sends an informational or inquiry message to a display station or system operator
Miscellaneous Commands	CHKOBJ (Check Object)	Checks for the existence of an object and, optionally, the necessary authority to use the object
	PRTCMDUSG (Print Command Usage)	Produces a cross-reference list for a specified group of commands used in a specified group of CL programs
	RTVCFGSRC (Retrieve Configuration Source)	Generates CL command source for creating existing configuration objects and places the source in a source file member
	RTVCFGSRC (Retrieve Configuration Status)	Gives applications the capability to retrieve configuration status from three configuration objects: line, controller, and device.
	RTVCLSRC (Retrieve CL Source)	Re-creates source from a CL program (object)
	RTVJOBA (Retrieve Job Attributes) *	Retrieves the value of one or more job attributes and places the values in a CL variable
	RTVSYSVAL (Retrieve System Value) *	Retrieves a system value and places it into a CL variable
	RTVUSRPRF (Retrieve User Profile) *	Retrieves user profile attributes and places them into CL variables
Program Commands	CRTCLPGM (Create CL Program)	Creates a CL program
	DLTPGM (Delete Program)	Deletes a program

Using CL Programs

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

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

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

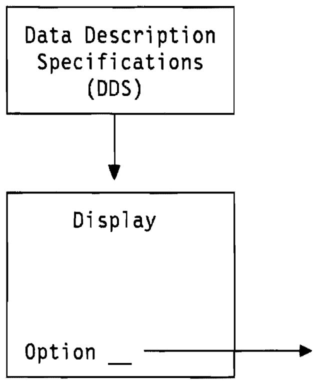
- Use a system value as a variable in a CL program.

```
                            PGM
                            DCL &TIME *CHAR 6
                            •
System Values               •
                            •
  QTIME  ─────────────────► RTVSYSVAL QTIME &TIME
  QDATE                     •
    .                       •
    .                       •
    .                       ENDPGM
```

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

```
                            PGM
                            DCL &USER *CHAR 10
Job Attributes              •
                            •
  Job Name                  •
  User Name ──────────────► RTVJOBA USER(&USER)
  Job Number                •
      .                     •
      .                     •
      .                     ENDPGM
```

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




```
PGM
DCL FILE(DISPLAY)
DCL &OPTION *CHAR 2
•
•
•
RCVF ...
IF (&OPTION *EQ 1) THEN(CALL PGMA)
•
•
•
ENDPGM
```

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

```
        PGM

        MONMSG MSGID(CPF0001) EXEC(GOTO ERROR)
        CALL PROGA
        CALL PROGB
        RETURN
ERROR:  SNDPGMMSG MSG('A CALL command failed') MSGTYPE(*COMP)
        ENDPGM
```

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

PROGA

```
PGM
DCL ...
.
.
.
CALL PROGB
.
.
.
ENDPGM
```

PROGB

```
PGM
DCL ...
.
.
.
CALL PROGC PARM(&FILEX)
.
.
.
RETURN
ENDPGM
```

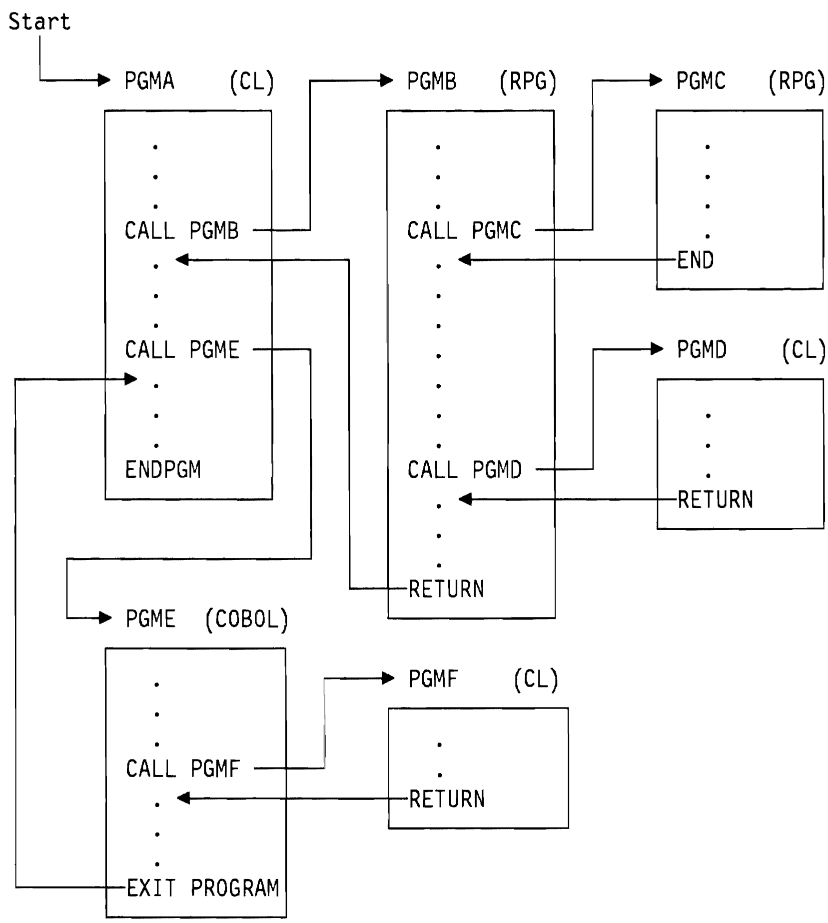
PROGC

```
PGM (&FILEX)
DCL ...
.
.
.
OVRDBF FILE(INPUT) TOFILE(&FILEX)
.
.
.
CALL PROGD
.
.
.
RETURN
ENDPGM
```

Using CL Programs to Control Processing

Used as a controlling program, a CL program can call programs written in other languages. The following illustration shows how control can be passed between a CL program and RPG/400* and COBOL programs in an application. To use the application, a work station user would request program A, which controls the entire application. The illustration shows:

- A CL program (PGMA) calling a COBOL program (PGME)
- A CL program (PGMA) calling an RPG/400 program (PGMB)
- An RPG/400 program (PGMB) calling another RPG/400 program (PGMC)
- An RPG/400 program (PGMB) calling a CL program (PGMD)
- A COBOL program (PGME) calling a CL program (PGMF)



Working with Variables

CL programs consist of CL commands, and the commands themselves consist of the command statement, parameters, and parameter values. Syntax rules for writing commands are explained in the *CL Reference*.

Parameter values may be expressed as variables, constants, or expressions. A *variable* is a named changeable value that can be accessed or changed by referring to its name. Variables can be used as substitutes for most parameter values on CL commands. When a CL program 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 programs.

Program variables are not stored in libraries; they are not objects; and their values are destroyed when the program that contains them is no longer called. Variable names in CL programs must begin with an & (ampersand) followed by no more than 10 characters. The first character following the & must be alphabetic and the remaining characters alphanumeric, for example, &FILE or &DSPFL1. The use of variables as values gives CL programming a special flexibility, because this allows high-level manipulation of objects whose content may change by specific applications. You might, for instance, write a CL program to direct the processing of other programs or the operation of several work stations without specifying which programs or work stations are to be controlled. These would be identified as variables in the CL program. The value of the variables can be defined (specified) when the CL program is run.

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

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

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

- Declare file. If your CL program 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 program compilation, the DCLF command implicitly declares CL variables for the fields and indicators defined in the file.

  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 (except the PGM command), but they can be intermixed in any order.

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

- Pass information between programs and jobs. See Chapter 3, "Controlling Flow and Communicating between Programs" on page 3-1.
- Pass information between programs and device displays. See "Working with Multiple Device Display Files" on page 5-14.
- Conditionally process commands. See "Controlling Processing within a CL Program" on page 2-22.
- Create objects. A variable can be used in place of an object name or library name, or both. The following example shows the Create Physical File (CRTPF) command used with a specified library in the first line, and with a variable replacing the library name in the second line:

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

Variables cannot be used to change a command name or keyword. Command parameters, however, can be changed during the processing of a CL program through the use of the prompting function. See "Allowing User Changes to CL Commands at Run Time" on page 6-12 for more information.

It is also possible to assemble the keywords and parameters for a command and process it using the QCMDEXC function. See "Using the QCMDEXC Program" on page 6-1 for more information about QCMDEXC.

Declaring a Variable

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

```
                                    ┌*CHAR┐
DCL     VAR(variable-name)  TYPE    {*DEC  }LEN(length)
        VALUE(initial-value)        └*LGL ┘
```

P7730437-0

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 9999 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'.
- If you do not specify an initial value, the following is assumed:
  - 0 for decimal variables
  - Blanks for character variables
  - '0' for logical 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 3000.
- Parameters must be declared as variables.

Using Variables to Specify a List or Qualified Name

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

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

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

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

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

Incorrect:

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

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

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

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

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

Incorrect:

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

Here again the value is viewed by the system as a single character string, not as two objects (a library and an object). If a qualified name must be handled as a single variable with a character string value, you can use the built-in function %SUBSTRING and the *TCAT concatenation function to assign object and library names to separate variables. See "Using the %SUBSTRING Built-In Function" on page 2-34 and Chapter 9 for examples using the %SUBSTRING function.

Lowercase Characters in Variables

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

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

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

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

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

Variables Replacing Reserved or Numeric Parameter Values

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

Each parameter on a command can accept only certain types of values. The parameter may allow an integer, a character string, a reserved value, a variable of a specified type, or some mixture of these, as values. Some types of values are required for parameters. If the parameter allows numeric values (if the value is defined in the command as *INT2, *INT4, 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 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 desired for a specified output queue. The variable &NRESP is a character variable manipulating numeric and predefined values (note the use of apostrophes). The JOBSEP parameter on the CHGOUTQ command will recognize these values as if they had been entered as numeric or predefined values. The DDS for the display file used in this program should use the VALUES keyword to restrict the user responses to IGNR, NONE, NORM, or SAME.

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

More information about types of values allowed by command parameters can be found in Chapter 9 and in the *CL Reference*.

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

Changing the Value of a Variable

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

- To a constant:

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

  or

  ```
  CHGVAR   &INVCMPLT  0
  ```

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

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

  or

  ```
  CHGVAR   &A  &B
  ```

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

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

  or

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

  The value of &A is increased by 1.
- To the value produced by the built-in function %SST (see "Using the %SUBSTRING Built-In Function" on page 2-34 for more information):

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

  &A is set to the first five characters of the value of the variable &B.
- To the value produced by the built-in function %SWITCH (see "Using the %SWITCH Built-In Function" on page 2-37 for more information):

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

  &A is set to 1 if job switches 1 and 8 are 0 and job switches 4, 5 and 6 are 1; otherwise, &A is set to 0.

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)
```

For a logical variable, the value to which the variable is to be changed must be a logical value. For character and decimal variables, either character or decimal values are accepted.

When specifying a decimal value for a character variable, remember the following:

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

For example, &A is a character variable to be changed to the value of the decimal variable &B. The length of &A is 6. The length and decimal positions of &B are 5 and 2, respectively. The current value of &B is 123. The resulting value of &A is 123.00.

When specifying a character value for a decimal 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 characters to the right of the decimal point than can be contained in the decimal variable, the characters are truncated. However, if the excess characters are to the left of the decimal point, they are not truncated and an error occurs.

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

Trailing Blanks on Command Parameters

Some command parameters are defined with the parameter VARY(*YES). (See the PARM, ELEM, and QUAL command definition statements for more information.) This parameter causes the length of the value passed to be the number of characters between the apostrophes. When a CL variable is used to specify the value for a parameter defined in this way, the system removes trailing blanks before determining the length of the variable to be passed to the command processor program. If the trailing blanks are present and are significant for the parameter, you must take special actions to ensure that the length passed includes them. Most command parameters are defined and used in ways which do cause this condition to occur. An example of a parameter defined where this condition is likely to occur is the key value element of the POSITION parameter on the OVRDBF command.

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

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

```
             PGM       PARM(&KEYVAL &LEN)
/*   PROGRAM TO SHOW HOW TO SPECIFY A KEY VALUE WITH TRAILING        */
/*      BLANKS AS PART OF THE POSITION PARAMETER ON THE OVRDBF       */
/*      COMMAND IN A CL PROGRAM.                                     */
/*   THE KEY VALUE ELEMENT OF THE POSITION PARAMETER OF THE OVRDBF   */
/*      COMMAND IS DEFINED USING THE VARY(*YES) PARAMETER.           */
/*      THE DESCRIPTION OF THIS PARAMETER ON THE ELEM COMMAND        */
/*      DEFINITION STATEMENT SPECIFIES THAT IF A PARAMETER           */
/*      DEFINED IN THIS WAY IS SPECIFIED AS A CL VARIABLE THE        */
/*      LENGTH IS PASSED AS THE VARIABLE WITH TRAILING BLANKS        */
/*      REMOVED. A CALL TO QCMDEXC USING APOSTROPHES TO DELIMIT      */
/*      THE LENGTH OF THE KEY VALUE CAN BE USED TO CIRCUMVENT        */
/*      THIS ACTION.                                                 */
/*    PARAMETERS--                                                   */
           DCL       VAR(&KEYVAL) TYPE(*CHAR) LEN(32)  /*  THE VALUE +
                       OF THE REQUESTED KEY. NOTE IT IS DEFINED AS +
                       32 CHAR.  */
           DCL       VAR(&LEN) TYPE(*DEC) LEN(15 5)  /*  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 OVERIDE 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
```

Writing Comments in CL Programs

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

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

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

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

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

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

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

```
        PGM /* ORD040C Order dept general menu */
        DCLF FILE(ORD040CD)
START:  SNDRCVF RCDFMT(MENU)
        IF (&RESP=1) THEN(CALL CUS210)
        /*Customer inquiry */
        ELSE +
              IF (&RESP=2) THEN(CALL ITM210)
              /**Item inquiry */
              ELSE +
                    IF (&RESP=3) THEN(CALL CUS210)
                     /* Customer name search */
                    ELSE +
                          IF (&RESP=4) THEN(CALL ORD215)
                          /** Orders by cust */
                          ELSE +
                                IF (&RESP=5) THEN(CALL ORD220)
                                /* Existing order */
                                ELSE +
                                      IF (&RESP=6) THEN(CALL ORD410C)
                                      /** Order entry */
                                      ELSE +
                                            IF (&RESP=7) THEN(RETURN)
        GOTO START
        ENDPGM
```

Controlling Processing within a CL Program

Commands in a CL program 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 program. These commands can be conditional (IF) or unconditional (GOTO).

Unconditional branching means that you can instruct processing to branch to commands or sets of commands located anywhere in the program without regard to what conditions exist at the time the branch instruction is processed. You can do this with the GOTO command.

Conditional branching means that under certain specified conditions, processing may branch to sections or commands that are not consecutive within the program. The branching may be to any statement in the program. This is called conditional processing because the branching only occurs when the specified condition is true. Conditional processing is usually associated with the IF command. With the ELSE command, you can specify alternative processing if the condition is not true.

The DO command allows you to create groups of commands that are always processed together, as a group, under specified conditions.

Using the GOTO Command and Labels

The GOTO command processes an unconditional branch. With the GOTO command, processing is directed to another part (identified by a label) of the program 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. 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 to which processing is directed by the GOTO command. To use a GOTO command, the command you are branching to must have a label.

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

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

Using the IF Command

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

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

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

The logical expression on the COND parameter must describe a relationship between two or more operands; the expression is then evaluated as true or false. See “Using the *AND, *OR, and *NOT Operators” on page 2-31 for more detailed information on the construction of logical expressions.

If the condition described by the logical expression is evaluated as true, the program processes the CL command on the THEN parameter. This may be a single command, or a group of commands (see "Using the DO Command and DO Groups" on page 2-25). If the condition is not true, the program runs the next sequential command.

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

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

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

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

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

In this case, the program processes the first IF command and then branches 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 then processes the next command, ENDPGM, which signals the end of the program, and returns control to the calling program.

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 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 program runs the next statement. If &C is equal to &D, PGMA is called. When PGMA returns, the third IF statement is considered, and so on. Note the difference in logic and processing between these simple sequential IF statements and the use of IF with ELSE or the use of embedded IF commands described later in the chapter (see "Using the ELSE Command" on page 2-27 and "Using Embedded IF Commands" on page 2-29).

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

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

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

Using the DO Command and DO Groups

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

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

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

P7730439-0

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

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

```
IF (&A=&B) THEN DO
              CALL PGMA
              CHGVAR &A &B    Do Group
              SNDPGMMSG...
              ENDDO
CALL PGMB
CHGVAR &ACCTS &B
```

P7730440-0

Do groups can be nested within other Do groups, up to a maximum of 10 levels of nesting.

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

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

P7730441-0

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

Using the ELSE Command

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

The IF command can be used without the ELSE command:

```
IF (&A=&B) THEN(CALL PROGA)
CALL PROGB
```

In this case, PROGA is called only if &A=&B, but PROGB is always called.

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

```
IF (&A=&B) THEN(CALL PROGA)
ELSE CMD(CALL PROGB)
CHGVAR &C 8
```

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

The real usefulness of the ELSE command is best demonstrated when combined with Do groups. In the following example, the Do group may not be run, depending on the evaluation of the IF expression, but the remaining commands are always processed.

```
IF (&A-&B) THEN(DO)
                    .
                    .     Conditioned-Run Only if True
                    .
          ENDDO

CHGVAR &C 8
SAVOBJ...                 Unconditioned-Run Whether or Not
CALL PGM(PAYROLL)         Expression Is True
ENDPGM
```

RSLF157-0

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:

```
IF (&A-&B) THEN(DO)
                .
                .     Conditioned for True Only
                .
          ENDDO
ELSE DO
      .
      .     Conditioned for False Only
   ENDDO
CHGVAR &C 8
SAVOBJ...
CALL PGM(PAYROLL)     Unconditioned
```

P7730443-1

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 program 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 program processes the RCLRSC command:

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

Using Embedded IF Commands

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

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

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

Up to 10 levels of such embedding are permitted in CL programming.

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

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

The preceding IF series is handled as one embedded command. Whenever any one of the IF conditions fails, processing branches to the remainder of the code (CHGVAR and subsequent commands). If the purpose of this code is to accumulate a series of conditions, all of which must be true for the Do group to process, it could be more easily coded using *AND with several expressions in one command. See "Using the *AND, *OR, and *NOT Operators" on page 2-31.

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 CALL PGMA
      ELSE CALL PGMB
CHGVAR &A 8
CALL PGM(DAILY)
ENDPGM
```

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

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

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

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

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

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

*AND and *OR are the reserved values for logical operators used to specify the relationship between operands in a logical expression. The ampersand symbol (&) can replace the reserved value *AND, and the vertical bar (|) can replace *OR. The reserved values must be preceded and followed by blanks. The operands in a logical expression consist of relational expressions or logical variables or constants separated by logical operators. The *AND operator indicates that both operands (on either side of the operator) have to be true to produce a true result. The *OR operator indicates that one or the other of its operands must be true to produce a true result.

Operators, other than logical 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 three kinds of operators other than logical operators:

- arithmetic (+, -, *, /)
- character (*CAT, ||, *BCAT, |>, *TCAT, |<)
- relational (*EQ, =, *GT, >, *LT, <, *GE, >=, *LE, <=, *NE, ¬=, *NG, ¬>, *NL, ¬<)

Information about these operators is found in the *CL Reference*.

The following are examples of logical expressions:

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

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

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

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

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

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

The operands in this relational expression could also be constants.

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

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

The series of dependent IF commands cited as an example in "Using Embedded IF Commands" on page 2-29 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)
CALL PGM(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 ORed 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.

For more information about logical and relational expressions, see the *CL Reference*.

Using the %SUBSTRING Built-In Function

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

The format of the substring built-in function is:

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

or

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

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

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

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

- ```
  DCL &NAME *CHAR VALUE(INVOICE)
  DCL &ERRCODE *DEC (2 0)
  IF (%SUBSTRING(&NAME 1 2) *EQ 'IN') +
  THEN(CALL INV210 &NAME)
  ELSE CHGVAR &ERRCODE 99
  ```

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

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

- Position and length can also be variables:

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

  This example changes the value of &X beginning at position &Y for the length &Z to 123.

- ```
  CHGVAR %SUBSTRING(&A 2 3) '123'
  ```

  If &A is ABCDEFG before this CHGVAR command is run, &A is A123EFG after the command runs.
  ```

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

  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.

  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.

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

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

- ```
  IF (((%SST*LDA 1 3) *CAT 'XYZ') *EQ &A) THEN(CALL PGMA)
  ```

  If the concatenation of positions 1 through 3 of the local data area plus the constant 'XYZ' is equal to variable &A, then PGMA 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 will be true and PGMA will be called.

- ```
        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 5) GOTO LOOP
        ENDDO
  ```

This program 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.

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

```
         PGM (&INPUT &REMAINDER)   /*  SEARCH  */
         DCL &INPUT *CHAR LEN(50)
         DCL &REMAINDER *CHAR LEN(50)
         DCL &X *DEC LEN(2 0) VALUE(03)
         DCL &L *DEC LEN(2 0)   /*  REMAINING LENGTH  */
SCAN:    IF (%SUBSTRING(&INPUT &X 1) *EQ '.') THEN(DO)
             CHGVAR VAR(&L) VALUE(50-&X)
             CHGVAR VAR(&X) VALUE(&X+1)
             CHGVAR VAR(&REMAINDER) VALUE(%SUBSTRING(&INPUT &X &L))
             RETURN
             ENDDO
         IF (&X *EQ 49) THEN(RETURN)
         CHGVAR &X (&X+1)
         GOTO SCAN
         ENDPGM
```

The program 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 program 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 program advances &X to position 4 and repeats the process.

Using the %SWITCH Built-In Function

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

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

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

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

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

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

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

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

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

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

%SWITCH with the IF Command

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

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

If job switches 1, 3, 4, 5, and 8 contain 0, 1, 1, 1, and 0, respectively, the result is true and the program 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 programs.

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

PGM  /* CONTROL */
IF (%SWITCH(11XXXXXX)) CALL PGMA
IF (%SWITCH(10XXXXXX)) CALL PGMB
IF (%SWITCH(01XXXXXX)) CALL PGMC
IF (%SWITCH(00XXXXXX)) CALL PGMD
ENDPGM

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

%SWITCH with the CHGVAR Command

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

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

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

Using the Monitor Message (MONMSG) Command

Escape messages are sent to CL programs by the commands in the CL programs and by the programs they call. These escape messages are sent to tell the programs that errors were detected and requested functions were not performed. CL programs 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 tries to move a data area that has been deleted, an object-not-found escape message is sent to the program by the Move Object (MOVOBJ) command.

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

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

Each message sent for a specific error has a unique identifier. You can enter as many as 50 message identifiers on the MSGID parameter. (See the *CL Reference* 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, DO, or a GOTO), which directs the program to perform error recovery.

In the following example, the MONMSG command follows a CHGVAR command and, therefore, is only monitoring for messages sent by the CHGVAR command:

```
CHGVAR VAR(&A) VALUE(&A/&B)
MONMSG MSGID(MCH1211) EXEC(CHGVAR VAR(&A) VALUE(1))
```

The escape message MCH1211 is sent to the program's message queue when a division by 0 is attempted. Because MSGID(MCH1211) is specified, the MONMSG command is monitoring for this condition; when it receives the message, the second CHGVAR command is run.

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

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

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

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

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

If the error occurs 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 PGMA would be called.

```
IF(&A/&B *EQ 5) THEN(DLTF ABC)
ELSE CALL PGMA
```

If you code the MONMSG command at the beginning of your CL program, the messages you specify are monitored throughout the program, regardless of which command produces these messages.

You can specify the same message identifier on a program-level or a command-level MONMSG command. The command-level MONMSG commands take precedence over the program-level MONMSG commands. In the following example, if message CPF0001 is received on CMDB, CMDC is run. If message CPF001 is received on any other command in the program, the program branches to EXIT2. If message CPF1999 is received on any command, including CMDB, the program 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, you must decide which ones you want to monitor for and handle. Most of these messages are sent to a program only if there is an error in the program. Others are sent because of conditions outside the program. Generally, a CL program should monitor for those messages that pertain to its basic function and that it can handle appropriately. For all other messages, OS/400 assumes an error has occurred and takes appropriate default action.

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

Values That Can Be Used as Variables

Retrieving System Values

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

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

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

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

The type of the variable must match the type of the system value. For character and logical system values, the length of the CL variable must equal the length of the value. For decimal values, the length of the variable must be greater than or equal to the length of the system value. System value attributes are defined in the *Work Management Guide*.

RTVSYSVAL Example

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

```
      PGM
      DCL  VAR(&PWRDNTME)  TYPE(*CHAR)  LEN(6)  VALUE('162500')
      DCL  VAR(&TIME)  TYPE(*CHAR)  LEN(6)
LOOP: RTVSYSVAL  SYSVAL(QTIME)  RTNVAR(&TIME)
      IF  (&TIME *GT &PWRDNTME)  THEN(SNDPGMMSG ('Powering down +
               in 5 minutes.  Please sign off.'))
      ELSE     DO
               DLYJOB DLY(300)   /*  WAIT 5 MINUTES  */
               GOTO LOOP
               ENDDO
      ENDPGM
```

See the *Work Management Guide* for a list of system values and how you can change and display them.

Converting the Format of a Date

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

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

The format for the CVTDAT command is:

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

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

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

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

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

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

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

Remember the following when converting dates:

- The length of the value in the DATE parameter and the length of the variable on the TOVAR parameter must be compatible with the date format. The length of the variable on the TOVAR parameter must be at least:
  - For non-Julian dates, 6 characters when no separators are used and 8 characters when separators are used.
  - For Julian dates, 5 characters when no separators are used and 6 characters when separators are used.

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

Retrieving Configuration Source

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

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

For more information about the RTVCFGSRC command, see the *CL Reference*.

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 to check the status of a configuration description.

For more information about the RTVCFGSRC command, see the *CL Reference*.

Retrieving Network Attributes

Using the Retrieve Network Attributes (RTVNETA) command, you can retrieve the network attributes of the system into a CL program. These attributes can be changed using the Change Network Attributes (CHGNETA) command and displayed using the Display Network Attributes (DSPNETA) command. See the *Work Management Guide* for more information about network attributes.

RTVNETA Example

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

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

Retrieving Job Attributes

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

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

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

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

The variable &CLKNAM, in which the user name is to be passed, is first declared using a DCL command. The RTVJOBA command follows the declare commands. When the program ORD410S2 is called, two variables, &NXTPGM and &CLKNAM, are passed to it. &NXTPGM is passed as blanks but could be changed by ORD410S2.

RTVJOBA Example

In the following CL program, a Retrieve Job Attributes (RTVJOBA) command retrieves the name of the message queue to which the job's completion message is sent, and uses that message queue to communicate with the user who submitted the job.

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

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

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

Return Code Summary

The return code (RTNCDE) parameter on the RTVJOBA command is a 5-digit decimal value with no decimal positions that indicates the status of called programs.

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

- RPG/400 programs

  The return codes sent by the RPG/400 compiler are:

  0 When the program is created
  2 When the program is not created

  The return codes sent by running RPG/400 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/400 return codes are tested only after a CALL:

  – 0 or 1 indicate no error
  – 3 gives an RPG/400 status code of 231
  – Any other value gives an RPG/400 status code 202 (call ended in error)

  The return code cannot be tested directly by the user in the RPG/400 program.

- 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. A return code value of 2 sends message CBE9001 and runs a Reclaim Resources (RCLRSC) command with the *ABNORMAL option.

- BASIC programs

  Compiled or interpreted BASIC programs can set the return code to any value between -32768 and 32767 by coding an expression on the END or STOP statements. If no expression is coded, the return codes are set to:

  0 For normal completion
  1 For programs ending because of an error

  BASIC return code values can be retrieved using the CODE intrinsic function.

- PL/I programs

  The return codes sent by PL/I programs are:

  0 For normal completion, set at the beginning of the run unit
  2 When set by a STOP statement or a call to PLIDUMP with the S option
  3 When an ERROR condition is raised
  4 When PL/I detects an error that did not allow the ERROR condition to be raised

  If any other value is found, it is set up by the PLIRETC built-in function or by a called procedure. PLIRETC passes a return code from the PL/I program to the program that called it, changing the current return code value.

PL/I programs can retrieve the return code using the PLIRETV built-in function.

- Pascal programs

  The Pascal compiler sets the following return codes:

  0 For successful compilation
  2 When the program object is not produced

  Pascal does not explicitly set the return code at run time. The user can use the ONERROR exception handling routine to monitor for particular exceptions, then set the return code using the RETCODE procedure.

  Pascal return codes can be retrieved using the RETVALUE parameter of the Pascal SYSTEM procedure. The return code is set to 0 before the SYSTEM call and will contain the updated return code when returned.

- CL programs

  The current value of the return code for compiled CL programs can be retrieved using the RTNCDE parameter on the RTVJOBA command.

Retrieving Object Descriptions

You can use the Retrieve Object Description (RTVOBJD) command to return the descriptions of a specific object to a CL program. Variables are used to return the descriptions. You can use these descriptions to help you detect unused objects. For more information about retrieving object descriptions, see "Retrieving Object Descriptions" on page 4-23.

You can use the Retrieve Object Description (QUSROBJD) application programming interface (API) to return the description of a specific object to a program. A variable is used to return the descriptions. For more information, see the *System Programmer's Interface Reference*.

Retrieving User Profile Attributes

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

You can also monitor for escape messages after running the RTVUSRPRF command. See the *CL Reference* for a list of the messages that could be sent.

RTVUSRPRF Example

In the following CL program, a RTVUSRPRF command retrieves the name of the user who called the program 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 program:

- &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. If no message queue is specified in the user profile, &USRMSGQ contains *NONE.
- &USRMSGQLIB contains the name of the library containing the message queue associated with the user profile. If no message queue is specified in the user profile, &USRMSGQLIB is blank.

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. See the *Database Guide* for information on database files and members.

RTVMBRD Example

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

```
DCL  &LIB      TYPE(*CHAR) LEN(10)
DCL  &MBR      TYPE(*CHAR) LEN(10)
DCL  &SYS      TYPE(*CHAR) LEN(4)
DCL  &MTYPE    TYPE(*CHAR) LEN(5)
DCL  &CRTDATE  TYPE(*CHAR) LEN(13)
DCL  &CHGDATE  TYPE(*CHAR) LEN(13)
DCL  &TEXT     TYPE(*CHAR) LEN(50)
DCL  &NBRRCD   TYPE(*DEC)  LEN(10 0)
DCL  &SIZE     TYPE(*DEC)  LEN(10 0)
DCL  &USEDATE  TYPE(*CHAR) LEN(13)
DCL  &USECNT   TYPE(*DEC)  LEN(5 0)
DCL  &RESET    TYPE(*CHAR) LEN(13)
.
.
.
RTVMBRD     FILE(*CURLIB/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 program:

- The current library name (MYLIB) is placed into the CL variable name &LIB.
- The system that MYFILE was found on is placed into the CL variable name &SYS. (*LCL means the file was found on the local system, and *RMT means the file was found on a remote system.)
- The member name (BMEMBER), since BMEMBER is the member immediately after AMEMBER in a name ordered member list (*NEXT), is placed into the CL variable named &MBR.
- The file attribute of MYFILE is placed into the CL variable named &MTYPE. (*DATA means the member is a data member, and *SRC means the file is a source member.)
- The creation date of BMEMBER is placed into the CL variable called &CRTDATE.

- The text used to describe BMEMBER is placed into the CL variable called &TEXT.
- The current number of records in BMEMBER is placed into the CL variable called &NBRRCD.
- The size of BMEMBER's data space (in bytes) is placed into the CL variable called &SIZE.
- The date that BMEMBER was last used is placed into the CL variable called &USEDATE.
- The number of days that BMEMBER has been used is placed into the CL variable called &USECNT. The start date of this count is the value placed into the CL variable called &RESET.

Working with CL Programs

A CL source program must be created (compiled) before it can be run. To create a CL program, you use the Create Control Language Program (CRTCLPGM) command. The following CRTCLPGM command creates the program ORD040C and places it in library DSTPRODLB:

```
CRTCLPGM     PGM(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 list is created.

On the CRTCLPGM command, you can specify list options and whether the program should operate under the program owner's user profile.

A CL program can run using either the owner's user profile or the user's user profile. See *Security Concepts and Planning* for more information.

Logging CL Program Commands

You can specify that most CL commands run in a CL program be written (logged) to the job log by specifying one of the following values on the LOG parameter on the CRTCLPGM command when the 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 CHGJOB command. Therefore, if you create the 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 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 CRTCLPGM command, you must recompile the program to change them.

When you specify logging, you should use the Remove Message (RMVMSG) command with care in order not to remove any logged commands from the job log. If you specify CLEAR(*ALL) on the RMVMSG command, any commands logged prior to running the RMVMSG command do not appear in the job log.

This affects only the CL program containing the RMVMSG command and does not affect any logged commands for the preceding or following recursion levels.

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

CHGVAR	DO	GOTO
DCL	ELSE	IF
DCLF	ENDDO	MONMSG
PGM	ENDPGM	

If the logging option is on, logging messages are sent to the CL program's message queue. If the CL program has been called 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 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). Arguments on the Transfer Control (TFRCTL) command are not printed.

Logging of commands in CL programs affects performance.

Listing Commands Used in CL Programs

You can list which CL programs use specified commands using the PRTCMDUSG command. This provides a cross-reference tool for application analysis.

Using the list of commands used, you can easily determine which programs need to be recompiled when changes are made to the command definitions of the specified commands. You may want to see what is specified for a typical parameter for all uses of a certain command, or you may want to see if you are monitoring for a specific error condition when using a particular command (for example, the CHKJOB command).

The PRTCMDUSG command allows you to specify as many as 50 command names and search all programs or all programs associated with a generic name by the following:

- A specified library for which you are authorized
- All libraries in the user part of the library list for the job
- All user libraries for which you are authorized

CL Program Compiler Lists

When you create a CL program, you can create various types of lists using the OPTION and GENOPT parameters on the CRTCLPGM command.

The OPTION parameter values and their meanings are:

- *SOURCE, *SRC, *NOSOURCE, or *NOSRC

  Whether a list of the source input is to be produced (*SOURCE or *SRC is the default).

- *GEN or *NOGEN

  Whether a program is to be created (*GEN is the default).

- *XREF or *NOXREF

  Whether a list of cross-references to variables and data references in the source input is to be produced (*XREF is the default). This option can only be specified if *SOURCE or *SRC is specified.

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

```
         1                                                                                                    2                    3
5728SS1 R03 M00 861114                  Control Language &CONT.
                 MYLIB/DUMPERR         06/15/88 13:15:57         Page    1
Program . . . . . . . . . . . . . . . . . . . :   DUMPERR
  Library . . . . . . . . . . . . . . . . . . :     MYLIB
Source file . . . . . . . . . . . . . . . . . :   MYFILE
  Library . . . . . . . . . . . . . . . . . . :     QCLSRC
Source member name  . . . . . . . . . . . . . :   DUMPERR            06/15/88 13:15:37   4
Source printing options . . . . . . . . . . . :   *SOURCE   *XREF  *GEN  *NOSECLVL
Program generation options  . . . . . . . . . :   *NOLIST   *NOXREF  *NOPATCH
User profile  . . . . . . . . . . . . . . . . :   *USER
Program logging . . . . . . . . . . . . . . . :   *JOB
Allow RTVCLSRC command  . . . . . . . . . . . :   *YES
Authority . . . . . . . . . . . . . . . . . . :   *CHANGE
Text  . . . . . . . . . . . . . . . . . . . . :   Test program
Compiler  . . . . . . . . . . . . . . . . . . :   IBM AS/400 Control Language Compiler  5
   6                                           Control Language Source
SEQNBR  *...+... 1 ...+... 2 ...+... 3 ...+... 4 ...+...  5 ...+... 6 ...+... 7 ...+... 8 ...+... 9 ...+.  DATE  8
   100- PGM                                                                                                06/15/88
   200- DCL &ABC *CHAR 10 VALUE('THIS')                                                                    06/15/88
   300- DCL &XYZ *CHAR 10 VALUE('THAT')     7                                                              06/15/88
   400- DCL &MNO *CHAR 10 VALUE('OTHER')                                                                   06/15/88
   500- CRTLIB LB(LARRY)                                                                                   06/15/88
* CPD0043 30 Keyword LB not valid for this command   9
   600- DLTLIB LIB(MOE                                                                                     06/15/88
* CPD0013 30 Unbalanced parenthesis found
   700- MONMSG CPF0000 EXEC(GOTO ERR)                                                                      06/15/88
   800- ERROR:                                                                                             06/15/88
   900- CHGVAR &ABC 'ONE'                                                                                  06/15/88
  1000- CHGVAR &XYZ 'TWO'                                                                                  06/15/88
  1100- CHGVAR &MNO 'THREE'                                                                                06/15/88
  1200- DMPCLPGM                                                                                           06/15/88
  1300- ENDPGM                                                                                             11/01/85
                                        * * * * *   E N D   O F   S O U R C E   * * * * *
5728SS1 R08 M00 861114                  Control Language                 MYLIB/DUMPERR        06/15/88 13:15:57        Page    2
                                         Cross Reference
Declared Variables
Name            Defined      Type            Length      References
&ABC               200       *CHAR             10          900
&MNO               400       *CHAR             10         1100
&XYZ               300       *CHAR             10         1000   10
Defined Labels
Label           Defined      References    11
ERR             ******         700
* CPD0715 30 Label 'ERR      ' referenced in command but label does not exist
ERROR              800
                               * * * * *   E N D   O F   C R O S S   R E F E R E N C E   * * * * *
5728SS1 R08 M00 861114                  Control  Language                MYLIB/DUMPERR        06/15/88 13:15:57        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
Program DUMPERR not created in library MYLIB. Maximum error severity 30.   13
                               * * * * *   E N D   O F   M E S S A G E   S U M M A R Y   * * * * *
                                   * * * * *   E N D   O F   C O M P I L A T I O N   * * * * *
```

Title:

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

2 The date and time of the compiler run.

3 The page number in the list.

Prolog:

4 The parameter values specified (or defaults if not specified) on the CRTCLPGM 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.

See the *Database Guide* for information about how sequence numbers are assigned.

7 The source statements.

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

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

For more information about compilation errors, see “Errors Encountered during Compilation” on page 2-53.

Cross-References:

10 The symbolic variable table is a cross-reference list 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 list 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 list because no general error messages were issued for the sample program. If there were general error messages for this program, 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.

The CL compiler produces, from the CL commands in a CL program, an intermediate system code needed to cause the requested function to be performed when the program runs. The program resolution monitor (PRM) creates the program from the intermediate system code.

If you specify *GEN on the OPTION parameter (a program is created), you can also specify that the IRP be listed for problem analysis purposes. The GENOPT parameter values and their meanings are:

- *LIST or *NOLIST

  Whether a list of the intermediate representation of a program (IRP) and the machine instructions is produced (*NOLIST is the default).

- *XREF or *NOXREF

  Whether a list of cross-references to data such as variables, constants, and labels in the IRP is to be produced (*NOXREF is the default).

- *PATCH or *NOPATCH

  Whether a patch area is to be reserved in the program. The size of the patch area is determined by the system and is based on the size of the program (*NOPATCH is the default).

Errors Encountered during Compilation

In the compiler list of a program, an error condition that relates directly to a specific command is listed after that command. See "CL Program Compiler Lists" on page 2-50 for an example of these inline messages. Messages that do not relate to a specific command but are more general in nature are listed in a messages section of the list, not inline with source statements.

The types of errors that are detected at compile time include syntax errors, references to variables and labels not defined, and missing statements. The following types of errors stop the program 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 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 program again.

Obtaining a Program Dump

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

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

- Run the Dump CL Program (DMPCLPGM) command. This command can only be used in a CL program and does not end the CL program.
- Enter D in response to inquiry message CPA0701. This message is sent whenever an unmonitored escape message is received by a CL program. If the program is running in an interactive job, the message is sent to the job's external message queue. If the program is running as a batch job, the message is sent to the system operator message queue, QSYSOPR.
- Specify INQMSGRPG(*SYSRPYL) for the job. See the *Work Management Guide* for a description of this job attribute. The IBM-supplied system reply list specifies a reply of D for message CPA0701, which will cause a dump to be printed when the inquiry message is sent.
- Change the default reply for message CPA0701 from C (cancel program) to D (dump program). A program dump is then printed whenever a function check occurs in a CL program. To change the default, enter the following command:

  ```
  CHGMSGD MSGID(CPA0701) MSGF(QCPFMSG) DFT(D)
  ```

  Note: The CHGMSGD command must be entered by the security officer or another user with update authority to message file QCPFMSG.

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                                                                                                        2
5728SS1 R08 M00 861114                                     CL Program Dump                                12/18/87 13:13:13          Page    1

Job name . . . . . . . . :   E_SMITH 3     User name . . . . . . . . :   SMITH 3     Job number . . . . . . . :    006757 3
Program name . . . . . . :   DUMP 4        Library . . . . . . . . . :   SMITH 4     Statement . . . . . . . . :    1200   5

                                                           Messages

              Message 6                                                                          From                    To
Time          ID             Sev      Type        Message Text                            Pgm          Inst        Pgm       Inst

103203        CPC2102        00       COMP        Library LARRY created.                  QLICRLIB     0000        DUMP      0000
103203        CPF2110        40       ESC         Library MOE not found.                  QLICRLIB     0151        DUMP      0011

                                                           Variables 7

Variable           Type              Length     Value                       Value in Hexadecimal
                                                *...+....1....+....2....+   * . . . + . . . . 1 . . . . + . . . . 2 . . . . +

&ABC               *CHAR                10      'ONE       '                D6D5C540404040404040
&MNO               *CHAR                10      'THREE     '                E3C8D9C5C54040404040
&XYZ               *CHAR                10      'TWO       '                E3E6D640404040404040

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

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

2 The date and time the dump was printed.

3 The fully qualified name of the job in which the program 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 program's 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 program, 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 program is used as a command processing program for a command that has a parameter with either TYPE(*NULL) or PASSVAL(*NULL) specified, or if RTNVAL(*YES) was specified for the parameter and a return variable is not coded on the command.

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

Displaying Program Attributes

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

- Name of the user profile that owns the program
- Program attribute, which indicates the compiler used to create the program.
- Program creation information:

  - Creation date and time
  - Name of source file and library and member
  - Date and time of last change to source file
  - Observable information associated with the program
  - User profile under which the program runs
  - Handling of adopted authority (if any)
  - Command logging option (for CL programs only)
  - Handling of saved and retrievable source (for CL programs only)
  - Handling of decimal data that is not valid
  - Text
- Program statistics
  - Number of parameters
  - Size of program
  - Size of associated space
  - Size of automatic and static storage
  - Number of machine interface instructions
  - Number of ODT entries (32,767 maximum)
  - State of the program
  - Compiler licensed program ID and release number
- Program performance information:
  - Optimization
  - Paging pool
  - PASA storage information
  - Amount of paging for program.

For more information on this command, see the *CL Reference*.

You can use the Change Program (CHGPGM) command to change some of the attributes.

Retrieving CL Program Source

If you need to recompile a CL program whose source has been deleted, you can re-create the CL source statements using the Retrieve CL Source (RTVCLSRC) command. In this command, you specify the name of the CL program, the name of the database source file to which the CL source statements are written and, optionally, the name of the source file member.

You can only retrieve CL program source for programs created with ALWRTVSRC(*YES) on the CRTCLPGM command. When the source statements are retrieved, a prologue, which includes the following information, is created:

- Owner name
- Last modification date of original source
- Date and time of source reconstruction
- Licensed program level at time of original compilation.

This information may be useful if a new version of the program needs to be created due to changes in the command definition or licensed program, or if the original source is lost. The new source may be used with the Create CL Program (CRTCLPGM) command to create a new version of the program.

Any comments that were present in the original source are not reproduced. The sequence number and source update field also cannot be reproduced; sequence numbers start at 100 and are incremented by 100, and the source update field is set to 0. (However, note that the date of last change to the original source is available in the new prologue.) If the source member already exists, the RTVCLSRC command clears the member before entering the new source.

Compiling for a Previous Release

The OS/400 Create CL Program (CRTCLPGM) command allows you to compile CL programs for use on the previous release. The CRTCLPGM command contains the Target Release (TGTRLS) parameter. TGTRLS allows the values of previous (*PRV) and current (*CURRENT), with *CURRENT being the default.

A CL program compiled with TGTRLS(*CURRENT) runs only on the current release or subsequent releases of the system. A CL program compiled with TGTRLS(*PRV) can also run on a previous release.

A CL program in which new release CL features exist cannot be compiled for a previous release. Likewise, a CL program containing new-release system commands cannot be compiled for a previous release. A CL program compiled with TGTRLS(*PRV) causes the compilation to occur as if it were compiling on a system running a previous release (N-1). After the program compiles successfully, it can be restored to a system with the previous release or subsequent releases and run correctly. To restore a program on such a system, you must first save the program specifying TGTRLS(*PRV). (CL programs compiled in the S/38 Environment cannot be saved for a previous release.)

You can install library QSYSPRV through the Licensed Program Menu (GO LICPGM), by selecting 5728SS1 OS/400* Compiler Support. The QSYSPRV library contains all system commands from the previous release. If a command specified in the CL program source exists in library QSYSPRV, the command object in library QSYSPRV is used for syntax checking the command when the CL program is compiled with TGTRLS(*PRV) specified. If you are not using this capability of CRTCLPGM, you can delete this library in order to save disk space. To delete the QSYSPRV library, use the following command:

```
DLTLICPGM LICPGM(5728SS1) OPTION(9)
```

You can also put your own commands from a previous release in library QSYSPRV. The system commands in library QSYSPRV should not be run because unexpected results may occur. The commands are used only for syntax checking by the CL compiler. If you change system commands, they are removed from QSYSPRV when a new level of the library is installed. You should also put library QSYSPRV into the system library list before library QSYS. To do this, use the Change System Library (CHGSYSLIBL) command when editing CL source (using SEU) for a previous release.

Chapter 3. Controlling Flow and Communicating between Programs

You can use the CALL, Transfer Control (TFRCTL), and RETURN commands to pass control back and forth between programs. Each command has slightly different characteristics. Information may be passed to called programs as parameters when control is passed.

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

CALL Command

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

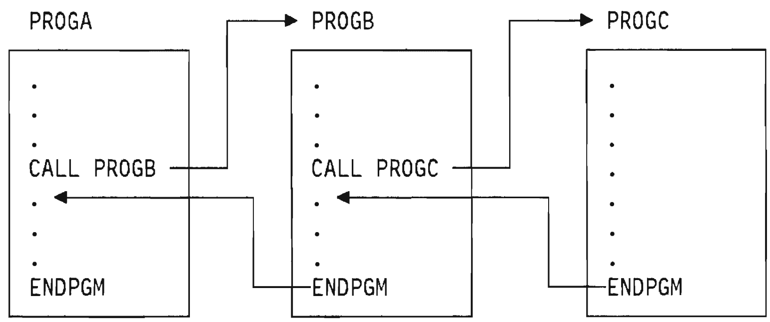
```
CALL PGM(library-name/program-name) PARM(parameter-values)
```

The program name and/or library name may be a variable. If the called program is in a library that is not on the library list, you must specify the qualified name of the program on the PGM parameter. The PARM parameter is discussed under "Passing Parameters between Programs" on page 3-4. When the called program finishes running, control returns to the next command in the calling program.

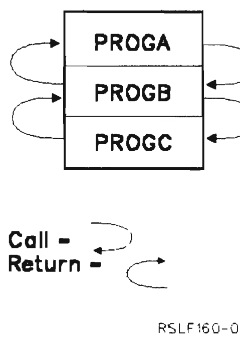
```
PROGA                       PROGB

PGM                         PGM
DCL ...                     DCL ...
.                           .
.                           .
.                           .
CALL PROGB                  CHGVAR ...
SNDPGMMSG ...               .
.                           .
.                           .
.                           .
ENDPGM                      ENDPGM
```

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



the program stack is:



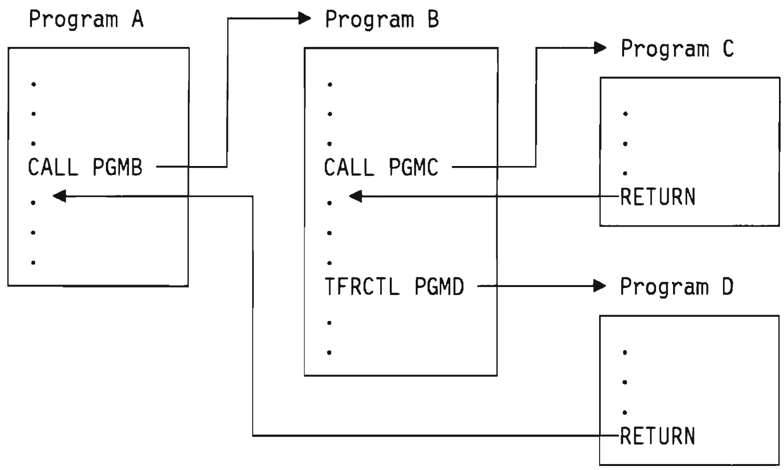
When PROGC finishes processing, control returns to PROGB at the command after the call to PROGC. Control is thus returned up the program stack. This occurs whether or not PROGC ends with a RETURN or an ENDPGM command.

A CL program can call itself.

TFRCTL Command

The Transfer Control (TFRCTL) command calls the program specified on the command, passes control to it, and removes the transferring program from the program stack. Reducing the number of programs on the program stack can have a performance benefit. When a CALL command is used, the program called returns control to the program containing the CALL command. When a TFRCTL command is used, control returns not to the program containing the TFRCTL command but to the statement after the most recently run CALL command in the program that called the transferring program.

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 the programs complete processing, the owner's authorities are no longer in effect. Any overrides issued by Program B remain in effect while Program D is running and are lost when Program D does a return.



The TFRCTL command has the following format:

```
TFRCTL PGM(library-name/program-name) PARM(CL-variable)
```

The program (and library qualifier) may be a variable.

It is important to note that only variables may be used as parameter arguments on this command, and that those variables must have been received as a parameter in the argument list from the program that called the transferring program. That is, the TFRCTL command cannot pass a variable that was not passed to the program running the TFRCTL command.

In the following example, the first TRFCTL is valid. The second TFRCTL command is not valid because &B was not passed to this program. The third TFRCTL command is not valid because a constant cannot be specified as an argument.

```
PGM PARM(&A)
DCL &A *DEC 3
DCL &B *CHAR 10
IF (&A *GT 100) THEN (TFRCTL PGM(PGMA) PARM(&A)) /*   valid   */
IF (&A *GT 50) THEN (TFRCTL PGM(PGMB) PARM(&B))  /* not valid */
ELSE (TFRCTL PGM(PGMC) PARM('1'))                /* not valid */
ENDPGM
```

The PARM parameters are discussed under "Passing Parameters between Programs" on page 3-4.

RETURN Command

The RETURN command in a CL program removes that program from the program stack.

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

If the program containing the RETURN command was called by a TFRCTL command, control is returned to the next sequential statement after the command that called the transferring program. If this command is also a TFRCTL command, control is returned up the program stack until a CALL command is found.

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

The RETURN command has no parameters.

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

Passing Parameters between Programs

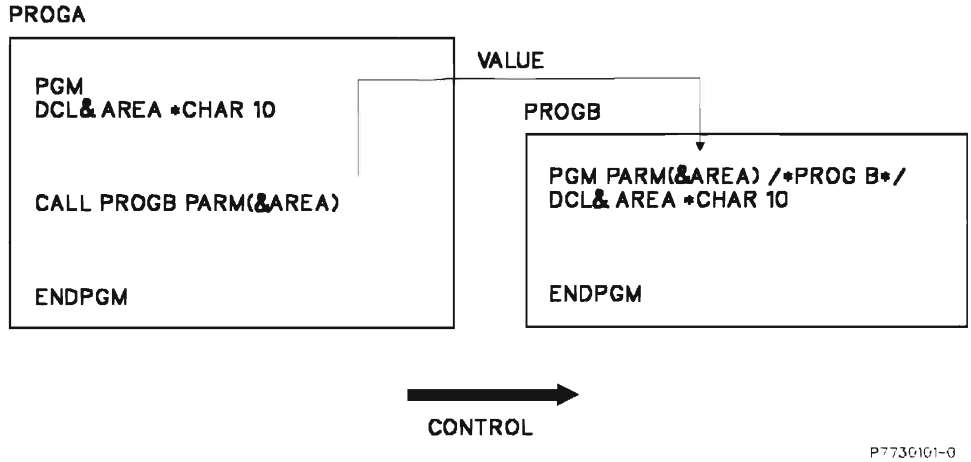
When you pass control to another program, you can also pass information to it for modification or use within the receiving program. See the discussion of this under "Using the CALL Command" on page 3-6. You can specify the information to be passed on the PARM parameter on the CALL or TFRCTL command. 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 either the CALL or TFRCTL 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. Because parameters are passed by position, not name, the position of the value passed in the CALL or TFRCTL 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 program must be declared as the same length and type as they are in the passing program. Decimal constants are always passed with a length of (15 5). 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.

For instance, if you wrote a program receiving a value &VAR1, like this:

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

The CALL command must specify 36 characters:

```
CALL PGMA (ABCDEFCHIJKLMNOPQRSTUVWXYZABCDEFGHIJ)
```

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.

Using the CALL Command

When the CALL command is issued by a CL program, each parameter value passed to the called program can be a character string constant, a numeric constant, a logical constant, or a CL program variable. A maximum of 40 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 storage in the called program is associated with the variables it receives. Instead, when a variable is passed, the storage for the variable is in the program in which it was originally declared. When a constant is passed, a copy of the constant is made in the calling program and that copy is passed 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.

If a CL program might be called using a CALL command that has not been compiled (an interactive CALL), 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 run in batch (except via the SBMJOB command in a CL program) or interactive mode cannot pass variables as arguments.

If the character parameter defined in the program is 32 characters or more, the parameter passed on the CALL command must be the same length. If the character parameter is less than 32 characters, the parameter value is padded on the right with blanks to the required length.

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 (in this case, no message is sent). 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` is accepted and used by the program. Quoted character strings can also be passed. Numeric values passed as characters must be enclosed in apostrophes.

- Decimal constants are passed in packed form and with a length of `LEN(15 5)`, where the value is 15 digits long, of which 5 digits are decimal positions. Thus, if a parameter of `12345` is passed, the receiving program must declare the decimal field with a length of `LEN(15 5)`; the parameter is received as `12345.00000`.

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

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

- Logical constants are passed with a length of 32 bytes. The logical value 0 or 1 is in the first byte and the remaining bytes are blank or ignored by the receiving program. If a value other than 0 or 1 is passed to a program which expects a logical value, the results may be unexpected.

- A floating point literal or floating point special value (*NAN, *INF, or *NEGINF) is passed as a double precision value, which occupies 8 bytes. Although a CL program cannot process floating point numbers, it can receive a floating point value into a character variable and pass that variable to an HLL program that can process floating point values.

- A program variable can be passed if the call is made from a CL program, in which case the receiving program must declare the field to match the variable defined in the calling CL program. For example, if a CL program defines a decimal variable named &CHKNUM as `LEN(5 0)`, the receiving program must declare the field as packed with 5 digits total, with no decimal positions. When a CALL command is run in batch mode, via the SBMJOB command in a CL program, variables passed as arguments are treated as 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.

- 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.

The logical constant '1' does not have to be declared in the passing 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 (passed &D in program A) is received as a character because it is declared as a character. Apostrophes are not necessary to indicate a string if TYPE is *CHAR. In program A, the length defaults to the value's length of 7 (the decimal point is considered a position in a character string). Program B expects a length of 32. The first 7 characters are passed, but the contents past the position 7 cannot be predicted.

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

```
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 (XYZ) was 456 and was intended as alphanumeric data, the value would have been specified as '456' in the parameter.

For more information on the default lengths in the DCL commands, see the DCL command in the *CL Reference*.

Using the TFRCTL Command

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 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.

PROGA

```
PGM
DCL &A
DCL &B
.
.
.
CALL PROGB (&A &B)
.
.
.
ENDPGM
```

PROGB

```
PGM (&A &B)
DCL &A *DEC
DCL &B *DEC
DCL &C *DEC
CHGVAR &C (&A + &B)
CHGVAR &A (&A - 2)
IF (&A = 3) GOTO LABEL
TFRCTL PROGC (&A &B)
LABEL: RETURN
ENDPGM
```

PROGC

```
PGM (&A &B)
DCL &A *DEC
DCL &B *DEC
CHGVAR &B (&B - &A)
RETURN
ENDPGM
```

Common Errors in Calling Programs

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

Data Type Errors

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

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

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

In the following example, a decimal value is passed to a program defining a character variable. Generally, this error does not cause run-time failures, though calculations made on the value may produce erroneous results:

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

PGM PARM(&A)              /* PGMB */
DCL &A *CHAR 8
.
.
.
ENDPGM
```

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

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

Decimal Length and Precision Errors

If a decimal value is passed with an incorrect decimal length and precision errors in calling program length, 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 receiving 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 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 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')
```

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

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

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

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

```
PGM          /* PGMA */
DCL &A *DEC (7 2)
CALL PGMB PARM(&A)  /* (7 2) PASSED TO PGMB */
IF (&A *NE 0)  THEN(...)  /* *MCH1202 OCCURS HERE */
.
.
.
ENDPGM

PGM PARM(&A) /* PGMB */
DCL &A *DEC (5 2)        /* WRONG LENGTH */
.
.
.
CHGVAR &A (&B-&C)      /* VALUE PLACED in &A */
RETURN
```

When control returns to program PGMA and &A is referred to, the error occurs.

Character Length Errors

If you pass a character value longer than the declared character length errors in calling program length of the receiving variable, the receiving 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, program variables handled this way may function differently than expected.

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

If the receiving 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 program fails. Changes to the adopted storage take effect immediately.

In the following example, two 3-character constants are passed to the receiving program. Character constants are passed with a minimum of 32 characters; 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 in variables behave in exactly the same way.

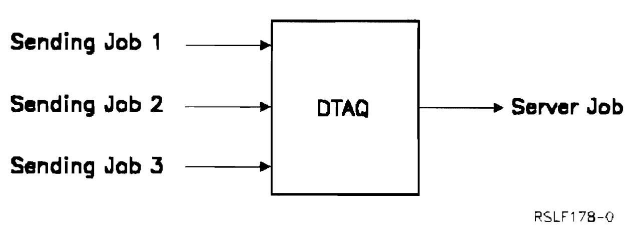
Using Data Queues to Communicate between Programs

Data queues are a type of system object that you can create, to which one HLL program can send data, and from which another HLL 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:

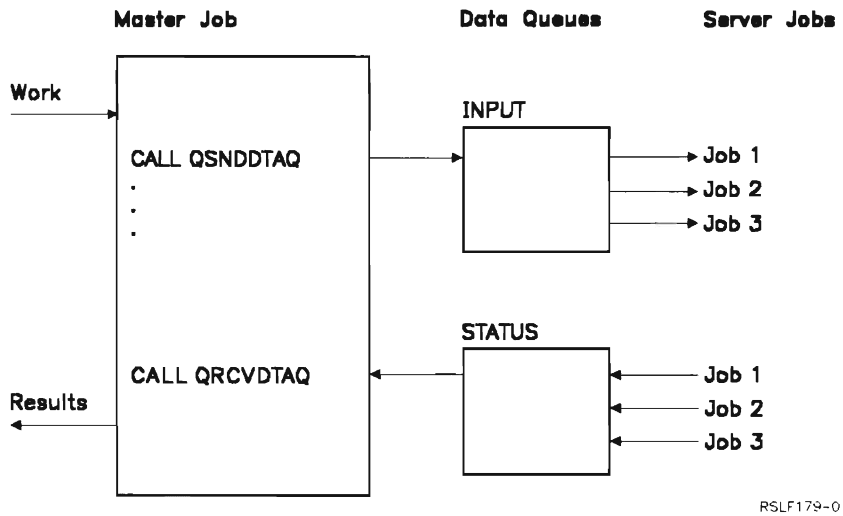
- Using data queues frees a job from performing some work. If the job is an interactive job, this can provide better response time and decrease the size of the interactive program and its process access group (PAG). This, in turn, can help overall system performance. For example, if several work station users enter a transaction that involves updating and adding to several files, the system can perform better if the interactive jobs submit the request for the transaction to a single batch processing job.
- Data queues are the fastest means of asynchronous communication between two jobs. Using a data queue to send and receive data requires less overhead than using database files, message queues, or data areas to send and receive data.
- You can send to, receive from, and clear a data queue in any HLL program by calling the QSNDDTAQ, QRCVDTAQ, and QCRLDTAQ programs without exiting the HLL program or calling a CL program to send, receive, or clear.
- When receiving data from a data queue, you can set a time out such that the job waits until a data entry is shown on the data queue. This differs from using the EOFDLY parameter on the OVRDBF command, which causes the job to be activated whenever the delay time ends.
- More than one job can receive data from the same data queue. This has an advantage in certain applications where the number of entries to be processed is greater than one job can handle within the desired performance restraints. For example, if several printers are available to print orders, several interactive jobs could send requests to a single data queue, and a separate job for each printer could receive from the data queue, either in first-in-first-out (FIFO) or last-in-first-out (LIFO) order.

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



Another example using data queues follows. A master 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 master program using another data queue.

Data queues allow the master job to route the work to the server jobs. This frees the master 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 programs have the following options:

- Wait until an entry is placed on the queue
- Wait for a specific period of time and then continue processing
- Return immediately.

Data queues can also be used when a program needs to wait on 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

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 for an invited ICF file
- A user-defined entry is made to a data queue by one or more jobs running on the system

When an application program calls QRCVDTAQ, the data is processed according to whether it is a display file, ICF file, or user-defined entry. For more information, see "Example 2: Waiting for Input from a Display File and an ICF File" on page 3-19 and "Example 3: Waiting for input from a Display File and a Data Queue" on page 3-22.

Comparisons with Using Database Files as Queues

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

- Data queues have been improved to communicate between active programs, not to store large volumes of data or large numbers of entries. For these purposes, use database files as queues.
- Data queues should not be used for long-term storage of data. For this purpose, you should use database files.
- When using data queues, you should include abnormal end procedures in your programs to recover any entries not yet completely processed before the system is ended.

Similarities to Message Queues

Data queues are similar to message queues, in that programs can send data to the queue that is received later by another 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 or last-in-first out 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 MAXLEN parameter is required and specifies the maximum length of the entries that can be sent to the data queue.

Managing the Storage Used by a Data Queue

When an entry is received from a data queue, the entry is removed from the data queue but the auxiliary storage is not freed. The same auxiliary storage is used again when a new entry is sent to the data queue. The queue grows larger as entries are sent to the queue and not received. Performance is better if the size of the queue is kept to less than 100 entries. If a data queue has grown too large, delete the data queue using the Delete Data Queue (DLTDTAQ) command, then re-create the queue using the Create Data Queue (CRTDTAQ) command.

Allocating Data Queues

If your application requires that a data queue is not accessed by more than one job at a time, it should be coded to include an Allocate Object (ALCOBJ) command before using a data queue. The data queue should then be deallocated using the Deallocate Object (DLCOBJ) command when the application is finished using it.

The ALCOBJ command does *not*, by itself, restrict another job from sending or receiving data from a data queue. However, if all applications are coded to include the ALCOBJ command before any use of a data queue, the allocation of a data queue already allocated to another job will fail, preventing the data queue from use by more than one job at a time.

When an allocation fails because the data queue is already allocated to another job, the system issues an error message, CPF1002. The Monitor Message (MONMSG) command can be used in the application program to monitor for this message and respond to the error message. Possible responses include sending a message to the user and attempting to allocate the data queue again. See "Monitoring for Messages in a CL Program" on page 8-16 for more information.

Sending Data with Data Queues

To send data to a data queue, call the QSNDDTAQ program from your HLL program. For example, in a CL program, you could specify the following:

```
CALL  PGM(QSNDDTAQ)  PARM(&QNAME  &LIB  &FLDLEN  &FIELD)
```

You must pass four parameters, as follows:

- &QNAME is a 10-byte character field that names the data queue.
- &QLIB is a 10-byte character field that names the library containing the data queue. *LIBL can be used.

  Note: To improve data queue performance, the system uses the name and library of the last data queue used. This has two implications:

  - If *LIBL is specified, the library list for the job that runs the program is used to find the appropriate data queue. If the library list is changed after a call to QSNDDTAQ, but the &LIB parameter value is unchanged, the same queue is used again.
  - A data queue can be renamed (RNMOBJ command), deleted (DLTDTAQ command), or moved (MOVOBJ command). A queue with the same name and library as the data queue that was renamed, deleted, or moved can be created. If this is done and data is sent to the same named data queue, the data is sent to the old queue. The data is not sent to the newly created queue.

    To solve these two problems, a Reclaim Resources (RCLRSC) command can be issued after the data queue has been moved, deleted, or renamed, or after the library list has been changed.

- &FLDLEN is a 5-digit packed decimal variable with no decimal positions. &FLDLEN specifies the number of characters sent to the data queue.

- &FIELD is a character field of length &FLDLEN. &FIELD is the field that contains the data to be sent to the data queue.

  Note:

  - If the length of this field is larger than &FLDLEN, only the number of characters (beginning from the left) as defined by &FLDLEN are sent to the data queue. If the length of this field is smaller than &FLDLEN, unexpected results can occur.
  - An error occurs if the &FLDLEN value is greater than the MAXREC length specified on the Create Data Queue (CRTDTAQ) command.

Receiving Data with Data Queues

To receive data from a data queue, call the QRCVDTAQ program from your HLL program. For example, in a CL program, you could specify the following:

```
CALL  PGM(QRCVDTAQ)  PARM(&QNAME  &LIB  &FLDLEN  &FIELD  &WAIT)
```

You must pass five parameters, as follows:

- &QNAME is a 10-byte character field that names the data queue.
- &QLIB is a 10-byte character field that names the library containing the data queue. *LIBL can be used.

  Note: To improve data queue performance, the system remembers the name and library of the last data queue used. This has two implications:

  - If *LIBL is specified, the library list for the job that runs the program is used to find the appropriate data queue. If the library list is changed after a call to QRCVDTAQ, but the &LIB parameter value is unchanged, the same queue is used again.
  - A data queue can be renamed (RNMOBJ command), deleted (DLTDTAQ command), or moved (MOVOBJ command). A queue with the same name and library as the data queue that was renamed, deleted, or moved can be created. If this is done and data is sent to the same named data queue, the data is sent to the old. The data is not sent to the newly created queue.

    To solve these two problems, a Reclaim Resources (RCLRSC) command can be issued after the data queue has been moved, deleted, or renamed, or after the library list has been changed.

- &FLDLEN is a 5-digit packed decimal variable with no decimal positions. &FLDLEN specifies the number of characters received from the data queue. If a time out occurs, this field is zero.
- &FIELD is a character field of length &FLDLEN. &FIELD is the field that receives the data coming from the data queue.

  Note: If the length of this field is larger than that specified by &FLDLEN, only the number of characters (beginning from the left) as defined by &FLDLEN are changed. If the length of this field is smaller than the value specified for the MAXLEN parameter on the Create Data Queue (CRTDTAQ) command, unexpected results can occur.

- &WAIT is a 5-digit packed decimal field with no decimal positions. When no entries are on the data queue, the WAIT parameter specifies the following:
  - A negative value indicates an unlimited wait request.
  - Zero indicates to continue processing immediately (no waiting).
  - A positive value specifies the number of seconds to wait.

Note: The maximum value, 99999, allows a wait time of 27.77 hours.

When more than one program has a receive pending on a data queue at one time, a data entry sent to the data queue is received by only one of the programs. The program with the highest priority receives the entry. The next entry sent to the queue is given to the job with the next highest priority.

Entries on a data queue are received in either first-in-first-out or last-in-first-out order. When an entry is received, it is removed from the data queue.

Clearing Data from Data Queues

To clear data from a data queue, call the QCLRDTAQ program from your HLL program. For example, in a CL program, specify the following:

```
CALL PGM(QCLRDTAQ)  PARM(&QNAME &LIB)
```

You must pass the two parameters as follows:

- &QNAME is a 10-byte character field that names the data queue.
- &LIB is a 10-byte character field that names the library containing the data queue.

Note: If *LIBL is specified, the library list for the job that compiles the program is used to find the appropriate data queue. If the library list is changed after the call to QCLRDTAQ but the &LIB parameter is unchanged, the same data queue is used again.

Examples Using a Data Queue

Example 1: Sending Entries and Waiting Two Hours after Receiving: In the following example, program A sends entries to data queue DTAQ1 in library QGPL. Program B receives the entries from this data queue and waits for 2 hours (7200 seconds) before continuing with the command that occurs after the call to QRCVDTAQ.

The data queue would be created with the following command:

```
CRTDTAQ QGPL/DTAQ1 MAXLEN(80)
```

In this example, all data queue entries are 80 bytes long.

In program A, the following statements relate to the data queue:

```
PGM
DCL  &FLDLEN  *DEC  LEN(5 0)  VALUE(80)
DCL  &FIELD  *CHAR  LEN(80)
.
.(determine data to be sent to the queue)
.
CALL  QSNDDTAQ  PARM(DTAQ1 QGPL &FLDLEN &FIELD)
.
.
.
```

In program B, the following statements relate to the data queue:

```
        PGM
        DCL  &FLDLEN  *DEC  LEN(5 0)  VALUE(80)
        DCL  &FIELD  *CHAR  LEN(80)
        DCL  &WAIT  *DEC  LEN(5 0)  VALUE(7200)   /* 2 hours */
        .
        .
        .
LOOP:   CALL QRCVDTAQ  PARM(DTAQ1 QGPL &FLDLEN &FIELD &WAIT)
        IF    (&FLDLEN *NE 0) DO   /* Entry received */
               .
               . (process data from data queue)
               .
               GOTO LOOP   /* Get next entry from data queue */
        ENDDO
        .
        . (no entries received for 2 hours; process time-out condition)
        .
```

Example 2: Waiting for Input from a Display File and an ICF File

```
                          Job A

 ----------        ---------------       -----------
| ICF      |<----->| Application |      | Display   |
| file     |       |   program   |      |   file    |
 ----------        -------.-------       -----------
      |                  /|\                  |
      |                   |                   |
      |              ----------               |
       ------------->| Data    |<-------------
                     | queue   |
                      ----------
```

This example is different from the usual use of data queues because there is only one job. The data queue serves as a communication object within the job rather than between two jobs.

In this example, a program is waiting for input from a display file and an ICF file. Instead of alternately waiting for one and then the other, a data queue is used to allow the program to wait on one object (the data queue). The program calls QRCVDTAQ and waits for an entry to be placed on the data queue that was specified on the display file and the ICF file. Both files specify the same data queue. Two types of entries are put on the queue by display data management and ICF data management support when the data is available from either file. ICF file entries start with *ICFF and display file entries start with *DSPF.

The display file or ICF file entry that is put on the data queue is 80 characters in length and contains the field attributes described in the following list. Therefore, the data queue that is specified using the CRTDSPF, CHGDSPF, OVRDSPF, CRTICFF, CHGICFF, and OVRICFF commands must have a length of at least 80 characters.

Position (and Data Type)	Description
1 through 10 (character)	The type of file that placed the entry on the data queue. This field will have one of two values: *ICFF for ICF file *DSPF for display file If the job receiving the data from the data queue has only one display file or one ICF file open, then this is the only field needed to determine what type of entry has been received from the data queue.
11 through 12 (binary)	The unique identifier for the file. The value of the identifier is the same as the value in the open feedback area for the file. This field should be used by the program receiving the entry from the data queue only if there is more than one file with the same name placing entries on the data queue.
13 through 22 (character)	The name of the display file or ICF file. This is the name of the file actually opened, after all overrides have been processed, and is the same as the file name found in the open feedback area for the file. This field should be used by the program receiving the entry from the data queue only if there is more than one display file or ICF file that is placing entries on the data queue.
23 through 32 (character)	The library where the file is located. This is the name of the library, after all overrides have been processed, and is the same as the library name found in the open feedback area for the file. This field should be used by the program receiving the entry from the data queue only if there is more than one display file or ICF file that is placing entries on the data queue.

33 through 42 (character) The program device name, after all overrides have been processed. This name is the same as that found in the program device definition list of the open feedback area. For file type *DSPF, this is the name of the display device where the command or Enter key was pressed. For file type *ICFF, this is the name of the program device where data is available. This field should be used by the program receiving the entry from the data queue only if the file that placed the entry on the data queue has more than one device or session invited prior to receiving the data queue entry.

43 through 80 (character) Reserved.

The following example shows coding logic that the application 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        */
```

```
                                               /*  file. Since this entry*/
                                               /* does not contain the   */
                                               /* data received, the data*/
                                               /* must be read from the  */
                                               /* file before it can be  */
                                               /* processed.             */
      READ DATA FROM ICF  FILE
      PROCESS INPUT DATA FROM ICF FILE
      WRITE TO ICF FILE                         /* Write with Invite     */
    LOOP BACK TO RECEIVE ENTRY FROM DATA QUEUE
 .
 .
 .
 END
```

Example 3: Waiting for input from a Display File and a Data Queue

```
     Job A                                    Job B

-----------------      -----------      --------------------
| Batch job      |     |         |      | Interactive job   |
| placing user   |     |  Data   |      | using display     |
| entries on the |---->|  queue  |----->|   file and        |
| data queue     |     |         |      | receiving entries |
|                |     -----.-----      | from the data     |
-----------------          /|\          |    queue          |
                            |           ---------.----------
                            |                   /|\
                            |                    |
                            |                   \|/
                            |               -----------
                            |               | Display |
                            ----------------|  file   |
                                            -----------
```

In this example, the program in Job B is waiting for input from a display file that it is using and for input to arrive on the data queue from Job A. Instead of alternately waiting for the display file and then the data queue, the program waits for one object, the data queue.

The program calls QRCVDTAQ and waits for an entry to be placed 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 put on this queue, the display file entry and the user-defined entry. The display file entry is placed on the data queue by display data management when data is available from the display file. The user-defined entry is placing on the data queue by Job A.

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
```

Using Data Areas to Communicate between Programs

A data area is an object used to hold data for access by any job running on the system. A data area can be used whenever you need to store information of limited size, independent of the existence of programs 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 program, use a Retrieve Data Area (RTVDTAARA) command 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 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.

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.

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 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 program.
- Store information without the overhead of creating and deleting a data area yourself.

Most high-level languages can also use the local data area. The SBMxxxJOB and STRxxxRDR commands cause jobs to start with a local data area initialized to blanks. Only the SBMJOB command allows the contents of the submitting job's local data area to be passed to the new job

Group Data Area

The system creates a group data area when an interactive job becomes a group job (using the Change Group Attributes [CHGGRPA] command). Only one group data area can exist for a group. The group data area is deleted when the last job in the group is ended (with the ENDJOB, SIGNOFF, or ENDGRPJOB command, or with an abnormal end), or when the job is no longer part of the group job (using the CHGGRPA command with GRPJOB(*NONE) specified).

A group data area, which is initially filled with blanks, has a length of 512 and type *CHAR. You can use a group data area from within a group job by specifying *GDA for the DTAARA parameter on the CHGDTAARA, RTVDTAARA, and DSPDTAARA commands. A group data area is accessible to all of the jobs in the group.

The following are true for a group data area:

- You cannot use the group data area as a substitute for a character variable on the substring built-in function (%SUBSTRING or %SST). (You can, however, move a 512-byte character variable used by the substring function into or out of the group data area.)
- A group data area cannot be referred to by jobs outside the group.
- You cannot create, delete, or allocate a group data area.
- No library is associated with a group data area.

The contents of a group data area are unchanged by the Transfer to Group Job (TFRGRPJOB) command.

In addition to using the group data area as you use other data areas, you can use the group data area to communicate information between group jobs in the same group. For example, after issuing the Change Group Job Attributes (CHGGRPA) command, the following command can be used to set the value of the group data area:

```
CHGDTAARA  DTAARA(*GDA)  VALUE('January1988')
```

This command can be run from a CL program or can be issued by the work station user.

Any other CL program in the group can retrieve the value of the group data area with the following CL command:

```
RTVDTAARA  DTAARA(*GDA)  RTNVAR(&GRPARA)
```

This command places the value of the group data area (January1988) into CL variable &GRPARA.

Program Initialization Parameter (PIP) Data Area

A PIP data area (PDA) is created for each prestart job when the job is started. The object sub-type of the PDA is different than a regular data area. The PDA can only be referred to by the special value name *PDA. The size of the PDA is 2000 bytes but the number of parameters contained in it is not restricted.

The RTVDTAARA, CHGDTAARA, and DSPDTAARA CL commands and the RTVDTAARA and CHGDTAARA macro instructions support the special value *PDA for the data area name parameter.

Creating a Data Area

Unlike program variables, data areas are objects and must be created before they can be used in a program or job. 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 will be used only in a CL program or also with other high-level language programs. For CL programs, the data area can have as many as 15 digits to the left of the decimal point and as many as 9 digits to the right, but only 15 digits total. For other languages, the data area can have as many as 15 digits to the left of the decimal point and as many as 9 to the right, for a total of up to 24 digits.
- A logical value '0' or '1', where '0' can mean off, false, or no; and '1' can mean on, true, or yes.

When you create a data area, you can also specify an initial value for the data area. If you do not specify one, the following is assumed:

- 0 for decimal
- Blanks for character
- '0' for logical.

To create a data area, use the Create Data Area (CRTDTAARA) command. In the following example, a data area is created to pass a customer number from one program to another:

```
CRTDTAARA  DTAARA(CUST) TYPE(*DEC) +
           LEN(5 0) TEXT('Next customer number')
```

Data Area Locking and Allocation

The CHGDTAARA command uses a *SHRUPD (shared for update) lock on the data area during command processing. The RTVDTAARA and DSPDTAARA commands use a *SHRRD (shared for read) lock on the data area during command processing. If you are performing more than one operation on a data area, you may want to use the Allocate Object (ALCOBJ) command to prevent other users from accessing the data area until your operations are completed.

For example, if the data area contains a value that is read and incremented by jobs running at the same time, the ALCOBJ command can be used to protect the value in both the read and update operations. See Chapter 4 for how to allocate objects.

For information on handling data areas in other (non-CL) languages, refer to the appropriate HLL reference manual.

Displaying a Data Area

You can display the attributes (name, library, type, length, and text description) and the value of a data area. See the *CL Reference* for a detailed description of the Display Data Area (DSPDTAARA) command.

The display uses the 24-digit format with leading zeros suppressed.

Changing a Data Area

The Change Data Area (CHGDTAARA) command changes all or part of the value of a specified data area. It does not change any other attributes of the data area. The new value can be a constant or a CL variable. If the command is in a CL program, the data area does not need to exist when the program is created.

Retrieving a Data Area

The Retrieve Data Area (RTVDTAARA) command retrieves all or part of a specified data area and copies it into a CL variable in a CL program. The data area does not need to exist when the CL program is created, and the CL variable need not have the same name as the data area. Note that this command retrieves, but does not alter, the contents of the specified data area.

Retrieve Data Area Examples

Example 1: Assume that you are using a data area named ORDINFO to track the status of an order file. This data area is designed so that:

- Position 1 contains an O (open), a P (processing), or a C (complete).
- Position 2 contains an I (in-stock) or an O (out-of-stock).
- Positions 3 through 5 contain the initials of the order clerk.

You would declare these fields in your program as follows:

```
DCL VAR(&ORDSTAT) TYPE(*CHAR) LEN(1)
DCL VAR(&STOCKC) TYPE(*CHAR) LEN(1)
DCL VAR(&CLERK) TYPE(*CHAR) LEN(3)
```

To retrieve the order status into &ORDSTAT, you would enter the following:

```
RTVDTAARA DTAARA(ORDINFO (1  1))  RTNVAR(&ORDSTAT)
```

To retrieve the stock condition into &STOCK, you would enter the following:

```
RTVDTAARA DTAARA(ORDINFO (2  1)) RTNVAR(&STOCKC)
```

To retrieve the clerk's initials into &CLERK, you would enter the following:

```
RTVDTAARA DTAARA(ORDINFO (3  3)) RTNVAR(&CLERK)
```

Each use of the RTVDTAARA command requires the data area to be accessed. If you are retrieving many subfields, it is more efficient to retrieve the entire data area into a variable, then use the substring built-in function to extract the subfields.

Example 2: The following example of the RTVDTAARA command places the specified contents of a 5-character data area into a 3-character variable. This example:

- Creates a 5-character data area named DA1 (in library MYLIB) with the initial value of 'ABCDE'
- Declares a 3-character variable named &CLVAR1
- Copies the contents of the last three positions of DA1 into &CLVAR1

To do this, the following commands would be entered:

```
CRTDTAARA DTAARA(MYLIB/DA1) TYPE(*CHAR) LEN(5) VALUE(ABCDE)
.
.
.
DCL VAR(&CLVAR1) TYPE(*CHAR) LEN(3)
RTVDTAARA DTAARA(MYLIB/DA1 (3 3)) RTNVAR(&CLVAR1)
```

&CLVAR1 now contains 'CDE'.

Example 3: The following example of the RTVDTAARA command places the contents of a 5-digit decimal data area into a 5-digit decimal digit variable. This example:

- Creates a 5-digit data area named DA2 (in library MYLIB) with two decimal positions and the initial value of 12.39
- Declares a 5-digit variable named &CLVAR2 with one decimal position
- Copies the contents of DA2 into &CLVAR2

To do this, the following commands would be entered:

```
CRTDTAARA DTAARA(MYLIB/DA2) TYPE(*DEC) LEN(5 2) VALUE(12.39)
.
.
.
DCL  VAR(&CLVAR2) TYPE(*DEC) LEN(5 1)
RTVDTAARA DTAARA(MYLIB/DA2) RTNVAR(&CLVAR2)
```

&CLVAR2 now contains 0012.3 (fractional truncation occurred).

Changing and Retrieving a Data Area Example

The following is an example of using the CHGDTAARA and RTVDTAARA commands for character substring operations.

This example:

- Creates a 10-character data area named DA1 (in library MYLIB) with initial value ABCD5678IJ
- Declares a 5-character variable named &CLVAR1
- Changes the contents of data area DA1 (starting at position 5 for length 4) to the value EFG padding after the G with 1 blank)
- Retrieves the contents of data area DA1 (starting at position 5 for length 5) into the CL variable &CLVAR1

To do this, the following commands would be entered:

```
CRTDTAARA DTAARA(MYLIB/DA1) TYPE(*CHAR) LEN(10) +
     VALUE('ABCD5678IJ')
.
.
.
DCL VAR(&CLVAR1) TYPE(*CHAR) LEN(5)
CHGDTAARA DTAARA(MYLIB/DA1) (5 4)) VALUE('EFG')
RTVDTAARA DTAARA(MYLIB/DA1) (5 5)) RTNVAR(&CLVAR1)
```

The variable &CLVAR1 now contains 'EFG I'.

Chapter 4. Objects and Libraries

Objects are the basic units on which commands perform operations. For example, programs and files are objects. Through objects you can find, maintain, and process your data on the AS/400 system. You need only know what object and what function (command) you want to use; you do not need to know the storage address of your data to use it.

Object Types and Common Attributes

Each type of object on the AS/400 system has a unique purpose within the system and has an associated set of commands which process that type of object. The *CL Reference* contains a complete list of the types of objects, the abbreviations used as parameter values for object type parameters, and the definition of the object belonging to that type.

Each object type has a set of common attributes that describes the object. These common attributes are listed in Table 4-3 on page 4-21. The online help information for the Display Object Descriptions (DSPOBJD) display describes these attributes.

Functions Performed on Objects

Many functions can be performed on objects. Some functions the system performs automatically and others you request through commands.

Functions the System Performs Automatically

The functions performed automatically ensure that objects are processed in a consistent, secure, and proper 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. For example, create, change, and display are specific functions. The specific functions are described in other sections of this manual that describe the object type.
- Some common functions that apply to objects in general are explained in this guide:

Table 4-1. Common Functions for Objects

Function	Page
Searching for multiple objects or a single object in a library	4-12
Specifying authority for objects in a library	4-13
Placing objects in libraries	4-15
Describing objects	4-20
Displaying object descriptions	4-20
Retrieving object descriptions	4-23
Detecting unused objects on the system	4-25
Moving objects between libraries	4-29
Creating duplicate objects	4-31
Renaming objects	4-32
Deleting objects	4-34
Allocating and deallocating objects	4-34
Displaying the lock states on objects	4-38
Checking for object existence	5-3

Libraries

On the AS/400 system, objects are grouped in special objects called *libraries*. Objects are found using libraries. Usually to access an object, you specify its name and type, and the library in which it exists. Then, the system searches the specified library for the specified object name and type. (To access an object in a library, you must be authorized to the library and to the object. See "Security Considerations" on page 4-12 and "Specifying Authority for Libraries" on page 4-13 for more information.)

The object type and library do not always have to be specified. Sometimes they are implied in the command. For example, the Display Class (DSPCLS) command implies that the object type is a class. Sometimes the command does not imply the object type, and the object type can be specified in the OBJTYPE parameter. For example, the Display Object Description (DSPOBJD) command applies to any type of object.

If you specify a library name in the same parameter as the object name, the object name is called a *qualified name*. If you are entering a command in which you must specify a qualified name, for example, the object name could be:

```
DISTLIB/ORD040C
```

The order entry program ORD040C is in the library DISTLIB.

If you are using prompting during command entry and you are prompted for a qualified name, you receive prompts for both the object name and the library name. On most commands, you can specify a particular library name, specify *CURLIB (the current library for the job), or use a library list. Library lists are described in the following section.

Library Lists

For commands in which a qualified name can be specified, you can omit specifying the library name. If you do so, either of the following happens:

- For a create command, the object is created and placed in the user's current library, *CURLIB, or in a system library, depending on the object type. For example, programs are created and placed in *CURLIB; authorization lists are created and placed in the system library, QSYS.
- For commands other than a create command, the system normally uses a library list to find the object.

Library lists used by the AS/400 system consist of the following four parts.

System part
The system part of the library list (QSYSLIBL) contains objects needed by the system.

Product libraries
Two product libraries may be included in the library list. The product libraries are used to support languages and utilities that are dependent on libraries other than QSYS to process their commands.

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.

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, 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.

For a list of the libraries shipped with the system or optionally installable on the system, see the *CL Reference*.

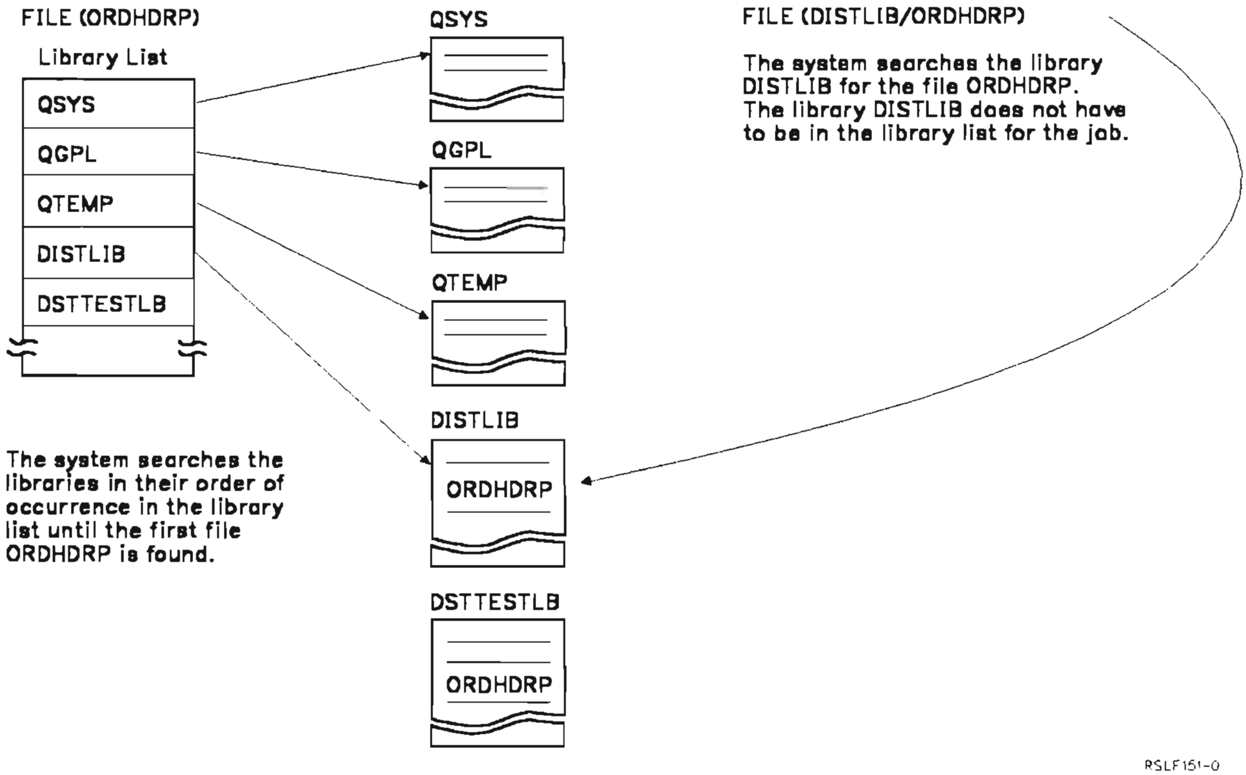
The following diagram shows an example of the structure of the library list:

Search order

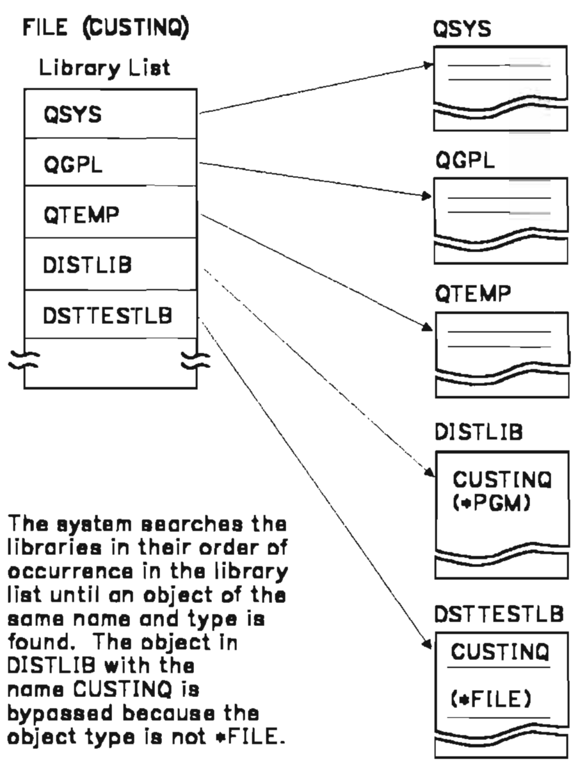
QSYS QSYS2 QUSRSYS QHLPSYS	System Part
QPDA	Product Library 1
QRPG	Product Library 2
OELIB	Current Library
OELIB QGPL QTEMP	User Part

Note: The system places library QPDA in product library 1 when the source entry utility (SEU) is used. When SEU is being used to syntax check source code, a second product library can be added to product library 2. For example, if you are syntax checking RPG source, then QPDA is product library 1 and QRPG is product library 2. In most other system functions, product library 2 is not used.

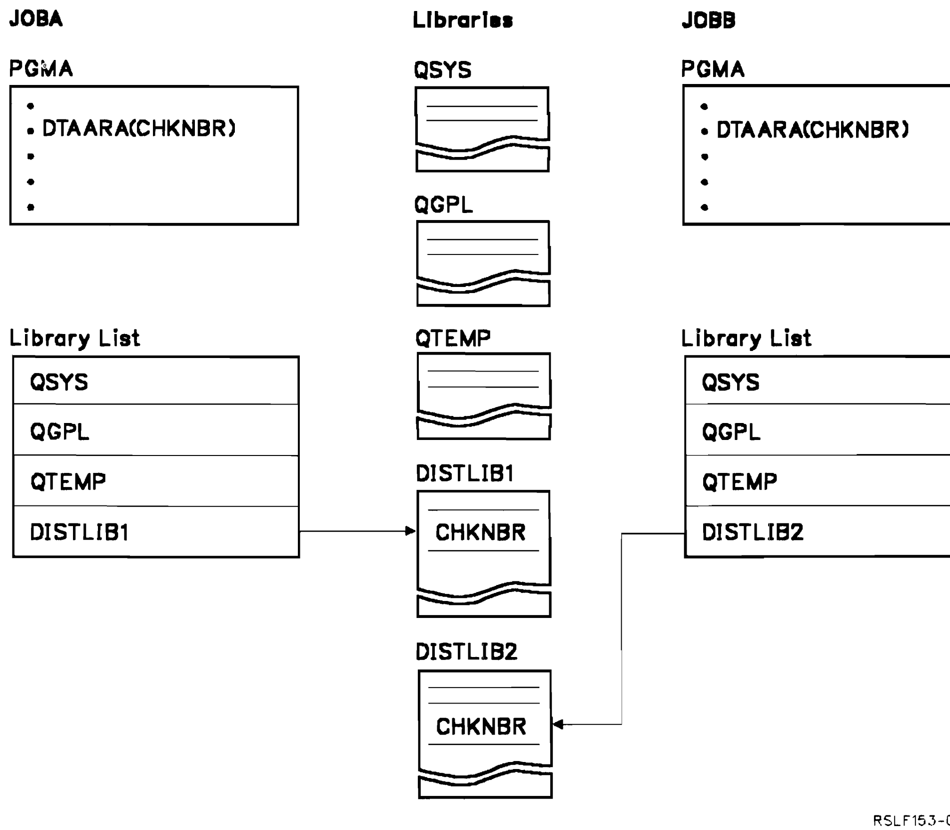
Using a library list simplifies finding objects on the system. Each job has a library list associated with it. When a library list is used to find an object, each library in the list is searched in the order of its occurrence in the list until an object of the specified name and type is found. If two or more objects of the same type and name exist in the list, you get the object from the library that appears first in the library list. The following diagram shows the searches made for an object both when the library list (*LIBL) is used and when a library name is specified:



The following diagram shows what happens when two objects of the same name but different types are in the library list:



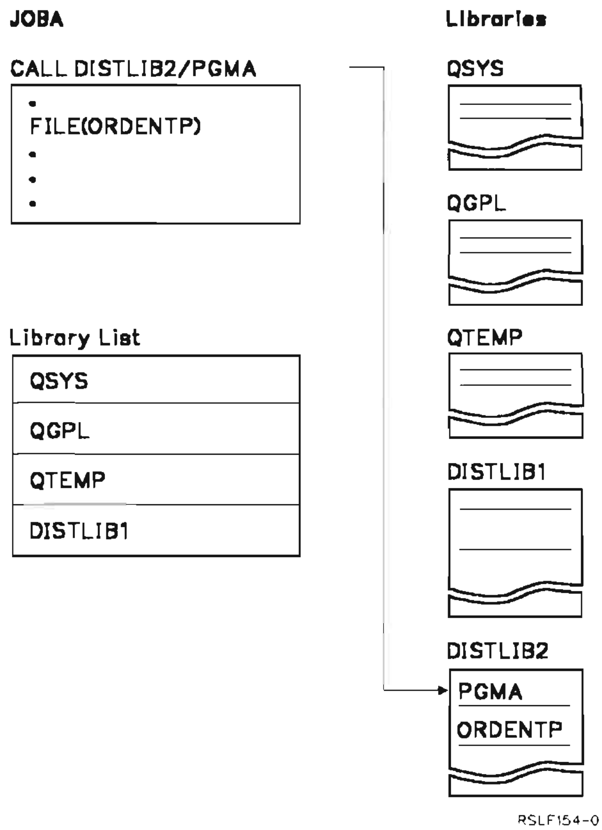
Generally, a library list is more flexible and easier to use than qualified names. More important than the advantage of not entering the library name, is the advantage of performing functions in an application on different data simply by using a different library list without having to change the application. For example, a CL program PGMA updates a data area CHKNBR. If the library name is not specified, the program can update the data area named CHKNBR in different libraries depending on the use of the library list. For example, assume that JOBA and JOBB both call PGMA as shown in the following illustration:



However, the use of a qualified name is advantageous in either 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

If, however, you call a program using a qualified name and the program attempts to open files whose names are not qualified, the files are not opened if they are not in the library list, as shown in the following example:



The call to PGMA is successful because the program name is qualified on the CALL command. However, when the program attempts to open file ORDENTP, the open fails because the file is not in one of the libraries in the library list, and its name is not qualified. If the library DISTLIB2 were added to the library list, or if a qualified file name had been used, the program could open the file.

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 *must* be included in the library list.

When the system is shipped, the system value QSYSLIBL contains the names of the libraries in the system part of the library list and the library names QSYS, QSYS2, QHLPSYS, and QUSRSYS. The system value QUSRLIBL contains the names of the libraries to become the user part of the library list.

QSYSLIBL can contain 15 library names, and QUSRLIBL can contain 25 library names. To change the system portion of a job's library list, use the Change System Library List (CHGSYSLIBL) command. To change the value of either QSYSLIBL or QUSRLIBL, use the Change System Value (CHGSYSVAL) command. A change to these system values takes effect on new jobs that are started after the system values are changed.

Changing the Library List

For a running job, you can add entries to or remove entries from the library list by using the Add Library List Entry (ADDLIBLE) command or the Remove Library List Entry (RMVLIBLE) command, or you can change the libraries in the library list by using the CHGLIBL command or the EDTLIBL command. The Add Library List Entry (ADDLIBLE) command allows you to add a library to the user portion of the library list. The Remove Library List Entry (RMVLIBLE) command allows you to remove a library from the user portion of the library list. 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.

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.

When a job is started, the user portion of the library list is determined by the values contained in the job description or by values specified on the SBMJOB command. A value of *SYSVAL can be specified, which causes the libraries specified by the system value QUSRLIBL to become the user portion of the library list. If you have specified library names in both the job description and the Batch Job (BCHJOB) or SBMJOB command, the library names specified in the BCHJOB or SBMJOB command override both the libraries specified in the job description and the system value QUSRLIBL.

The following shows the order in which the user part of the library list specified in QUSRLIBL is overridden by commands for individual jobs:

- A library list can be specified in the job description that, when the job is run, overrides the library list specified in QUSRLIBL. (See the *Work Management Guide* for information on job descriptions.)
- When a job is submitted either through a BCHJOB command or a SBMJOB command, a library list can be specified on the command. This list overrides the library list specified in the job description or in the system value QUSRLIBL.
- When a job is submitted using the SBMJOB command, *CURRENT (the default) can be specified for the library list. *CURRENT indicates that the library list of the job issuing the SBMJOB command is used.
- Within a job, an ADDLIBLE, RMVLIBLE, or CHGLIBL command can be used. These commands override any previous library list specifications.
- The current library for the job can be changed using the CHGCURLIB or CHGLIBL command.

Instead of entering the CHGLIBL command each time you want to change the library list, you can place the command in a CL program:

```
PGM  /* SETLIBL - Set library list */
CHGLIBL LIBL(APPDEVLIB QGPL QTEMP)
ENDPGM
```

If you normally work with this library list, you could set up an initial program to establish the library list instead of calling the program each time:

```
PGM  /* Initial program for QPGMR */
CHGLIBL LIBL(APPDEVLIB QGPL QTEMP)
TFRCTL PGM(QPGMMENU)
ENDPGM
```

This program must be created and the user profile to which it will apply changed to specify the new initial program. Control then transfers from this program to the QPGMMENU program, which displays the Programmer Menu.

If you occasionally need to add a library to the library list specified in your initial program, you can use the ADDLIBLE command to add that library to the library list. For example, the following command adds the library JONES to the end of the library list:

```
ADDLIBLE LIB(JONES) POSITION(*LAST)
```

If part of your job requires a different library list, you can write a CL program that saves the current library list and later restores it, such as the following program.

```
       PGM
       DCL &LIBL *CHAR 275
       DCL &CMD *CHAR 285
(1)    RTVJOBA USRLIBL(&LIBL)
(2)    CHGLIBL (QGPL QTEMP)
       .
       .
       .
(3)    CHGVAR &CMD ('CHGLIBL (' *CAT &LIBL *TCAT ')')
(4)    CALL QCMDEXC (&CMD 285)
       .
       .
       .
       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.

This program is an example of a CL program that saves the library list, changes it temporarily, and then restores the library list.

Considerations for Setting Up a Library List

You should consider the following when setting up a library list and using it:

- The libraries in a library list must exist on the system. The system values QSYSLIBL and QUSRLIBL are accessed when OS/400 is started. If a library in either of these values does not exist on the system, a message is sent to the system operator's message queue (QSYSOPR), the library is ignored, and OS/400 is started without the library. Once OS/400 is started, no libraries can be deleted from the library list of any active job. If any library in the library list specified in the job description or in a Job (JOB) or Submit Job (SBMJOB) command does not exist or is not available, the job is not started.
- The libraries in a library list must be authorized to all users who need to use them. To initialize a library list (for example, in a Submit Job [SBMJOB], Job [JOB], or Create Job Description [CRTJOBD] command), a user must have object operational authority for the libraries or the job is not started. A user must also have *USE authority to libraries added to the library list using the Add Library List Entry (ADDLIBLE) or Change Library List (CHGLIBL) command.
- When a program running under an adopted user profile adds a library to the library list that the current user is not authorized to, and does not remove the library from the library list before ending the program, the user retains access to certain object types in that library after the program exits. This only occurs when the *LIBL specification is used to access the objects.
- System performance is better when the library list is kept as short as possible.

Displaying a Library List

You can use the Display Library List (DSPLIBL) command to display the library list for a job currently running. The display contains a list of all the libraries in the library list in the order that they appear in the library list. If you are using batch processing, the library list is printed. If you are using interactive processing, the library list can be displayed or printed.

For a description of the library list display, see the *CL Reference*.

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. On the job library list display, the libraries are listed in the order they are searched, and both the system and user parts of the library list are displayed.

Typical Errors in Using Library Lists

The following are some typical errors that can occur when you use library lists:

- When using a qualified name to call a program: If the program tries to open a file using the library list and the file is not in a library in the library list, the file open fails.
- When submitting a create program command in a batch job: If the program uses an externally described file whose name is not qualified and the file is not in a library in the library list, the create program command fails.
- When creating a CL program: If the program uses a CHGLIBL command to access an object not in a library in the library list for the job, the program

creation fails. The CHGLIBL command is not processed by creating the program, therefore the referenced object will be unavailable, causing the creation to fail. If a program is dependent on an object that is not in the libraries in the library list, run the CHGLIBL command before attempting to create the program.

- When looking for the message queue for error messages: The message queue for error messages can be found using the system library list, not the library list for the current job.

Using Generic Object Names

Sometimes you may want to search for more than one object (even though only one might be found) when the object names start with the same characters. This type of search is called a *generic search* and can be used on several commands.

To use a generic search, specify a generic name in place of the object name on the command. A generic name consists of a set of characters common to all the object names that identifies a group of objects and ends with an * (asterisk). All objects whose names begin with the specified characters and to which you are authorized have the requested function performed on them. For example, if you entered the Display Object Description (DSPOBJD) command using the generic name ORD*, object descriptions for the objects beginning with ORD are shown.

A generic search can be limited by the following library qualifiers on the generic name (the library name parameter value is given in parentheses, if applicable):

- A specified library. The operation you requested is performed on the generically named objects in the specified library only.
- The library list for the job (*LIBL). The libraries are searched in the order they are listed in the library list. The operation you requested is performed on the generically named objects in the libraries specified in the library list for the job.
- The current library for the job (*CURLIB). The current library for the job is searched. If no current library exists, QGPL is used.
- All libraries in the user part of the library list for the job (*USRLIBL). The libraries are searched in the order they are listed in the library list, including the current library (*CURLIB). The operation you requested is performed on the generically named objects in the libraries specified in the user portion of the library list for the job.
- All user libraries for which you are authorized (*ALLUSR). The libraries are searched in alphanumeric order. QGPL is searched, but all other libraries that begin with the letter Q, such as QRPG and QPDA, are not searched. 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.

For information on operations using generic functions, see the *CL Reference.*

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).

If you specify an object name with *LIBL or *USRLIBL as the library name, you can specify only one object type.

Security Considerations

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.
- Public authority can be granted for an object. Public authority is authority granted to all users not given authority from another source. See *Security Concepts and Planning* for more information about public authority.

Security Concepts and Planning explains in detail the types of authority that can be granted for an object and what authority a user needs perform a function on that object. Authority that can be granted for libraries is discussed under "Specifying Authority for Libraries" on page 4-13.

Special considerations apply when writing a program that must be secure (for example, a program that adopts the security officer's user profile). See *Security Concepts and Planning* for information about writing these programs.

Using Libraries

A library is an object used to group related objects and to find objects by name. Thus, a library is a directory to a group of objects.

You can use libraries to:

- Group certain objects for individual users. This helps you manage the objects on your system. For example, you might place all the files that a user JOE can use in a library JOELIB.
- Group all objects used for an individual application. For example, you might place all your order entry files and programs into an order entry library DISTLIB. You need only add one library to the library list to ensure that all your order entry files and programs are in the list. This is advantageous if

you do not want to specify a library name every time you use an order entry file or program.

- Ensure security. For example, you can specify which users have authority to use the library and what they are allowed to do with the library.
- Simplify save/restore operations by grouping objects that are saved and restored at the same time into the same library. You can use a Save Library (SAVLIB) command instead of saving objects individually using the Save Object (SAVOBJ) command.
- Use multiple libraries for testing. Refer to Chapter 11 for more information.
- Use multiple production libraries. For example, you can use one production library for source files and for the creation of objects, one for the application programs and files, one for objects that are infrequently saved, and one for objects that are frequently saved.

Multiple libraries make it easier to use objects. For example, you can have two files with the same name but in different libraries so that one can be used for testing and the other for normal processing. As long as you do not specify the library name in your program, the file name in the program does not have to be changed for testing or normal processing. You control which library is used by using the library list. (Objects of the same type can have the same names only if they are in different libraries.)

Two types of libraries are: production and test. A production library is for normal processing. A test library is used in debug mode. In debug mode, you can protect database files in production libraries from being updated. (See Chapter 11 for more information on using test libraries.)

Creating a Library

To create a library, use the Create Library (CRTLIB) command. For example, the following CRTLIB command creates a library to be used to contain order entry files and programs. The library is named DISTLIB and is a production library. The default authority given to the public prevents the user from accessing the object.

```
CRTLIB   LIB(DISTLIB) TYPE(*PROD) +
         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 any library whose name begins with the letter Q (such as QRPG or QPDA) is a system library. Except for QGPL, system libraries are not searched during a generic search when *ALLUSR is specified as the library.

Specifying Authority for Libraries

The default public authority for a library is *EXCLUDE, which prevents the user from accessing the object.

The following describes each of the authorities that can be given to users for libraries. See *Security Concepts and Planning* for more information.

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 to other users. Only the object owner or a user with *ALLOBJ can grant object management authority for a library.
- Rename the library

Object existence authority for a library includes authority to:

- Delete the library
- Transfer ownership of the library

***USE authority** for a library (consisting of object operational authority and read authority) includes authority to:

- Use a library to find an object
- Display library contents
- Place a library in the library list
- Save a library

***CHANGE authority** provides operational authority to objects and all data authorities. The user can add, change, and delete entries in an object, or read the contents of an entry in the object.

***ALL authority** provides all object authorities and data authorities. The user can control the object's existence, specify the security for the object, change the object, and perform basic operations on the object, such as run a program or display the object's description and contents.

Add authority for a library allows a user to create a new object in the library or to move an object into the library.

Update and **read authority** for a library allows 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. The following display shows an example of authority that is in effect for library DISTLIB. See the description of the Display Object Authority (DSPOBJAUT) command in the *CL Reference* for more information

```
                        Display Object Authority

Object . . . . . . . :   DISTLIB         Object type . . . . :    *LIB
  Library  . . . . . :     QSYS          Owner . . . . . . . :    QSECOFR

Object secured by authorization list   . . . . . . . . . . . :    *NONE

               Object
User           Authority
QSECOFR        *ALL
*PUBLIC        *EXCLUDE

                                                                       Bottom
Press Enter to continue.
                                                     (C) COPYRIGHT IBM CORP.
F3=Exit   F11=Display detail   F12=Cancel   F17=Top   F18=Bottom
```

The following display is shown if the user presses F11 (Display detail) from the first Display Object Authority display:

```
                        Display Object Authority

Object . . . . . . . :   DISTLIB         Object type  . . . . :    *LIB
  Library  . . . . . :     QSYS          Owner  . . . . . . . :    QSECOFR

Object secured by authorization list  . . . . . . . . . . . . :    *NONE

             Object     -----Object------    ------------Data------------
User         Authority  Opr   Mgt   Exist    Read   Add   Update   Delete
QSECOFR      *ALL        X     X     X        X      X      X        X
*PUBLIC      *EXCLUDE

                                                                       Bottom
Press Enter to continue.

F3=Exit   F11=Non-display detail   F12=Cancel   F17=Top   F18=Bottom
```

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. 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: All objects except libraries, device descriptions, line descriptions, controller descriptions, mode descriptions, class-of-service descriptions, and user profiles are stored in libraries. These object types that are not stored in libraries are stored internally in the system. It is permissible to refer to these objects as being in QSYS even though they are not actually located in this library. Authorization lists are always in QSYS and are therefore not usually qualified by the library name.

Note: Use the QSYS library for system objects only. If a new release of OS/400 is installed, other licensed programs should not be restored to the QSYS library because any changes are lost.

Deleting and Clearing Libraries

When you delete a library with the Delete Library (DLTLIB) command, you delete the objects in the library as well as the library itself. When you clear a library with the Clear Library (CLRLIB) command, you delete objects in the library without deleting the library. To delete or clear a library, all you need to specify is the library name. For example:

```
DLTLIB LIB(DISTLIB)
```

or:

```
CLRLIB LIB(DISTLIB)
```

To delete a library, you must have object existence authority for both the library and the objects within the library, and operational 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), and Remove Library List Entry (RMVLIBLE) commands can also be used to change the library list.

3. Use the DLTLIB command to delete the library and the objects in the library.

Note: You cannot delete the library QSYS and should not delete any objects in it. You may cause the system to end because the system needs objects that are in QSYS to operate properly. You should not delete the library QGPL because it also contains some objects that are necessary for the system to be able to perform effectively. You should not use the library QRECOVERY because it is intended for system use only. The library QRECOVERY contains objects that the system needs to operate properly.

For concerns about deleting objects other than libraries, see "Deleting Objects" on page 4-34.

To clear a library, you must have object existence authority for the objects within the library and read authority for the library. If you try to clear a library but do not have object existence authority for all the objects in the library, the objects you do not have authority for are not deleted from the library. If an object is allocated to someone else, it is not deleted.

Displaying Library Names and Contents

You can use the Display Library (DSPLIB) or Work with Libraries (WRKLIB) command to display or print all the libraries you have authority to and find basic information on each object within the libraries.

The object information includes:

- The name and type of the object
- The attributes of the object
- Whether storage for the object's data has been freed
- The size of the object
- The description entered for the object when it was created

On the DSPLIB command, you can also specify a specific library name or names, in which case you bypass the library selection display. In this list, the objects are grouped by library; within each library, they are grouped by object type; within each type, they are listed in alphanumeric order. The order of the libraries is one of the following:

- If libraries are specified on the DSPLIB command, the libraries are displayed in the order they are specified in the display command.
- If *LIBL or *USRLIBL is specified on the DSPLIB command, the order of the libraries matches the order of the libraries in the library list for the job.
- If *ALL or *ALLUSR is specified on the DSPLIB command, the order of the libraries is in alphanumeric order. Only libraries the user is authorized to are displayed.

For example, the following DSPLIB command displays a list of the objects contained in DISTLIB:

```
DSPLIB  LIB(DISTLIB)  OUTPUT(*)
```

The asterisk (*) for the OUTPUT parameter means that the libraries are to be shown at the display station if in interactive processing and printed if in batch processing. To print a list when in interactive processing, specify *PRINT instead of taking the default *.

See the *CL Reference* for more information and sample displays for the DSPLIB command.

OS/400 Multiple Language Support

OS/400 supports different national languages on the same system. This allows information in one national language to be presented to one user while information in a different national language is presented to another user.

The language used for user-readable information (displays, messages, printed output, and online help information) is controlled by the library list for the job. By adding a national language library to the system portion of the library list, different national language versions of information can be presented.

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 QRPG for RPG/400) are added to the library list by the system when they are needed.

National language versions other than the system primary language are installed in secondary national language libraries. Each secondary language library contains a single national language version of the displays, messages, commands prompts, and help for *all* IBM licensed programs. The name of a secondary language library is in the form `QSYSnnnn`, where `nnnn` is a language feature code. For example, the feature code for French is 2928, so the secondary national language library name for French is QSYS2928.

If a user wants information presented in the primary national language of the system, no special action is required. To present information in a national language different from the primary national language of the system, the user must change the library list so that the desired national language library is positioned before all other libraries in the library list that contains national language information.

For example, to present the French version of displays, messages, and so on, the user could enter the following command to place French information at the top of the library list:

```
CHGSYSLIBL  LIB(QSYS2928)
```

The library list for an interactive job can be set up by an initial program specified in the user profile so the user does not have to run the CHGSYSLIBL command at every sign-on. The initial program uses the Change System Library List (CHGSYSLIBL) command to add the desired national language library to the top of the library list.

Note: The authority shipped with the CHGSYSLIBL command does not allow all users to run the command.

To enable a user to run the CHGSYSLIBL command without granting the user rights to the command, you can write a CL program containing the CHGSYSLIBL command. The program is owned by the security officer, and adopts the security officer's authority when created. Any user with authority to run the program can use it to change the system part of the library list in the user's job. The following is an example of a program to set the library list for a French user.

```
PGM
  CHGSYSLIBL  LIB(QSYS2928)  /* Use French information */
ENDPGM
```

Table 4-2. Feature Codes for National Languages

National Language	Feature Code
Belgian Dutch	2963
Belgian French	2966
Canadian French	2981
Traditional Chinese Double-Byte Character Set (DBCS)	2987
Danish	2926
Dutch Netherlands	2923
English uppercase	2950
English upper/lower case	2924
English upper/lower case Double-byte Character Set (DBCS)	2984
English Uppercase Double-byte Character Set (DBCS)	2938
Finnish	2925
French	2928
French Multinational Character Set (MNCS)	2940
German	2929
German Multinational Character Set (MNCS)	2939
Greek	2957
Icelandic	2958
Italian	2932
Italian Multinational Character Set (MNCS)	2942
Japanese Double-byte Character Set (DBCS)	2962
Korean Double-byte Character Set (DBCS)	2986
Norwegian	2933
Portuguese	2922
Portuguese Multinational Character Set (MNCS)	2991
Simplified Chinese Double-byte Character Set (DBCS)	2989
Spanish	2931
Swedish	2937
Turkish	2956

Describing Objects

Whenever you use a create command to create an object, you can describe the object in a 50-character field on the TEXT parameter of the create command. Some commands allow a default of *SRCMBRTXT which indicates the text for the object being created is to be taken from the text of the source member from which the object is being created. This is valid only for objects created from source in database source files.

If the source input for the create command is a device or inline file, or if source is not used, the default value is blank. This text becomes part of the object description and can be displayed using the Display Object Description (DSPOBJD) or Display Library (DSPLIB) command. The text can be changed using the Change Object Description (CHGOBJD) command.

For more information on these commands, see the *Database Guide* and the command descriptions in the *CL Reference*.

Displaying Object Descriptions

You can use the Display Object Description (DSPOBJD) or Work with Objects (WRKOBJ) command to display descriptions of objects. These descriptions are helpful for determining if objects exist on the system but are not being used. If you are using batch processing, the descriptions can be printed or written to a database file. If you are using interactive processing, the descriptions can be displayed, printed, or written to a database file.

You can display basic, full, or service attributes for object descriptions. These object descriptions are found in the following table:

Table 4-3. Attributes displayed for object descriptions

Basic Attributes	**Full Attributes**	**Service Attributes (see Notes)**
• Object name • Library name • Object type • Extended attributes • Freed status • Object size • Text description (partial)	• Object name • Library name • Object type • Extended attributes • Freed status • Object size • Text description • Creation date and time • User who created object • System object created on • Object domain • Change date • Whether or not usage data collected • Date last used • Days used count • Date used count reset • Storage size • Offline size • Freed status • Auxiliary storage pool • Save date • Restore date • Save command • File label ID	• Object name • Library name • Object type • Source file and library • Member name • Extended attributes • Freed status • Object size • Creation data and time • Date and time member in source file was last updated • System level • Compiler • Object control level • Whether or not changed by user • Licensed program • PTF number • APAR ID • Text description of object or object status conditions

Notes:

1. The service information is used by programming support personnel to determine the level of the system on which an object was created and whether or not the object has been changed since it was shipped. Some of this information may be helpful to you because it indicates the source member used to create an object and the last date of change to that source from which the object was created.
2. Library objects contain only the *names* of the objects included in the library. If DSPOBJD for object type *LIB is used, the object size information refers to the size of the library object only, not the total size of the objects included in the library.

Using the DSPOBJD command, you can list the objects in a library for which you are authorized by:

- Name
- Generic name
- Type
- Name or generic name within object type

The objects are listed by library; within a library, they are listed by type. Within object type, the objects are listed in alphanumeric order.

You may want to use the DSPOBJD command in a batch job if you want to display many objects with the *FULL or *SERVICE option. The output can go to a spooled printer file and be printed instead of being shown at the display station, or the output can go to a database file. If you direct the output to a database file, and if you specify the *SERVICE option, 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:     DISTLIB

Type options, press Enter.
  5=Display full attributes   8=Display service attributes

Opt  Object      Type      Attribute   Freed         Size   Text
 _   ORDDTLP     *FILE        PF        NO           1024   Order detail
 _   ORDHDRP     *FILE        PF        NO           1024   Order header











                                                                     Bottom
F3=Exit   F12=Cancel
```

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
Type . . . . . . . . :    *FILE

Text . . . . . . . . . . . . . . . . :

Creation information:
  Creation date  . . . . . . . . . . :  06/08/90  10:17:03
  Created by user  . . . . . . . . . :
  System created on  . . . . . . . . :
  Object domain  . . . . . . . . . . :  *SYSTEM
Object usage information:
  Date last changed  . . . . . . . . :  05/11/90  10:03:02
  Usage data collected . . . . . . . :  Y
  Date last used . . . . . . . . . . :  05/11/90  10:03:02
  Days used count  . . . . . . . . . :  20
  Date use count reset . . . . . . . :  03/10/90  11:09:21

                                                                  More...
Press Enter to continue.

F3=Exit   F12=Cancel
                                        (C) COPYRIGHT IBM CORP. 1980, 1990.
```

```
                     Display Object Description - Full
                                                               Library 1 of 1
Object . . . . . . . :    ORDDTLP       Attribute  . . . . . :
  Library  . . . . . :      DISTLIB     Owner  . . . . . . . :
Type . . . . . . . . :    *FILE

Storage information:
  Size . . . . . . . . . . . . . . . :  2560
  Offline size . . . . . . . . . . . :  0
  Freed  . . . . . . . . . . . . . . :  NO
  Auxiliary storage pool . . . . . . :  1
Save/Restore information:
  Save date  . . . . . . . . . . . . :
  Restore date . . . . . . . . . . . :
  Save command . . . . . . . . . . . :
  Sequence number  . . . . . . . . . :
  Save file  . . . . . . . . . . . . :
  Diskette volume  . . . . . . . . . :
  File label ID  . . . . . . . . . . :
                                                                  Bottom
Press Enter to continue.

F3=Exit   F12=Cancel
                                        (C) COPYRIGHT IBM CORP. 1980, 1990.
```

Retrieving Object Descriptions

You can use the Retrieve Object Description (RTVOBJD) command to return the descriptions of a specific object to a CL program. Variables are used to return the descriptions. You can use these descriptions to help you detect unused objects on the system

The descriptions that can be returned as variables for an object are:

- Name of the library that contains the object
- Any extended attribute of an object (such as program or file type)
- Text description of the object

- Name of the object owner's user profile
- Auxiliary storage pool ID
- 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 restored
- Name of the object creator's user profile
- System the object was created on
- Object domain
- Whether or not usage data was collected
- Date and time 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
- Size of the object in bytes
- Size of the object in bytes of storage at the time of the last save
- Command used to save the object
- Tape sequence number generated when the object was saved on tape
- Tape or diskette volumes used for saving the object
- Type of the device the object was last saved to
- Name of the save file if the object was saved to a save file
- Name of the library that contains the save file if the object was saved to a save file
- File label used when the object was saved
- Name of the source file that was used to create the object
- Name of the library that contains the source file that was used to create the object
- Name of the member in the source file
- Date and time the member in the source file was last updated
- Level of the operating system when the object was created
- Licensed program identifier, release level, and modification level of the compiler
- Object control level for the created object
- Information about whether or not the 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.

RTVOBJD Example

In the following CL program, a RTVOBJD command retrieves the description of a specific object. Assume an object called MOBJ exists in the current library (MYLIB).

```
DCL  &LIB       TYPE(*CHAR) LEN(10)
DCL  &CRTDATE   TYPE(*CHAR) LEN(13)
DCL  &USEDATE   TYPE(*CHAR) LEN(13)
DCL  &USECNT    TYPE(*DEC)  LEN(5 0)
DCL  &RESET     TYPE(*CHAR) LEN(13)
.
.
.
RTVOBJD     OBJ(MYLIB/MOBJ) OBJTYPE(*FILE) RTNLIB(&LIB)
            CRTDATE(&CRTDATE) USEDATE(&USEDATE)
            USECOUNT(&USECNT) RESETDATE(&RESET)
```

The following information is returned to the program:

- The current library name (MYLIB) is placed into the CL variable name &LIB.
- The creation date of MOBJ is placed into the CL variable called &CRTDATE.
- The date that MOBJ was last used is placed into the CL variable called &USEDATE.
- The number of days that MOBJ has been used is placed into the CL variable called &USECNT. The start date of this count is the value placed into the CL variable called &RESET.

Detecting Unused Objects on the System

Using the information provided for the following object descriptions, you can detect objects on the system that are no longer being used:

- Creator of the object
  - The creator of the object is the user profile that is doing the create operation, 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 is changed.
  - When an object is restored, the creator is the creator of the object on the media.
  - When an object is duplicated using CRTDUPOBJ, the creator of the object is the user running the command.
  - For IBM-supplied objects, the creator is `*IBM`.
  - For user objects that already exist on the system before Release 3.0 is installed, the creator of the object is blank.
- System created on (name of the system on which the object was created)
  - When an object is restored, the system created on is the system the object on the media was created on.
  - For IBM-supplied objects, the system created on is `00000000`.
  - For objects that already exist on the system before Release 3.0 is installed, the system created on is blank.
- 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 date last used. For example, if a user tries to change an object for which the user is not authorized, the date of last use does not change.
  - The date of last use is blanks 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 recreated during the restore operation lose the date of last use.

- 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.

- 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 recreated during the restore operation lose the days used count.
  - The days used count is zero for new objects.
  - 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 size is shown.
- Date days used count was reset
  - When the days used count is reset using the Change Object Description (CHGOBJD) command, the date is recorded so the user 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.

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: An application programming interface (API) exists that provides the same information as the Retrieve Object Description command. For more information, see the *System Programmer's Interface Reference*.

The Retrieve Member Description (RTVMBRD) command and Display File Description (DSPFD) command provide similar information for members in a file.

The following table gives information about object types and the commands and operations that update their usage information.

Table 4-4 (Page 1 of 3). 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)
Chart format	Display Chart (DSPCHT) command

Type of object	Commands and operations
C local description	Retrieve C Local Description Source (RTVCLDSRC) command or when referred to in a C program
Class	When used to start a job
System-defined command	When run When compiled in a CL program When prompted during entry of SEU source When calling the command analyzer in check mode **Note:** Prompting from the command line and then pressing F3 is not counted as a use of a system-defined command.
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	When data is sent to or received from a queue for the first time and when a different queue than the one used on the previous send or receive operation is specified in the job
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 queue	When an entry is placed on or removed from the queue
Line description	When the line goes beyond status of vary on pending
Menu	When a menu is displayed using the GO command **Note:** Does not apply to system menus

Table 4-4 (Page 2 of 3). Updating usage information

Table 4-4 (Page 3 of 3). Updating usage information

Type of object	Commands and operations
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
Message queue	When a message is sent to, received from, or removed from a user message queue other than QSYSOPR and QHST
Output queue	When an entry is placed on or removed from the queue
Overlay	When referred to during a print operation
Page segment	When referred to during a print operation
Print descriptor group	When referred to during a print operation
Program	Retrieve CL Source (RTVCLSRC) command When run and not a system program When bound with another program to create a program object
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
Subsystem description	When subsystem is started
Spelling aid dictionary	When used to create another dictionary When retrieved When a word is found in the dictionary during a spell check and the dictionary is not an IBM-supplied spelling aid dictionary
Table	When used by a program for translation
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)

Object usage information is not updated for the following object types:

- Alert table
- Authorization list
- Configuration list
- Class-of-service description
- Document
- Interactive data definition
- Edit description

- Forms control table
- Folder
- Double-byte character set dictionary
- Double-byte character set sort
- Double-byte character set table
- Journal
- Journal receiver
- Library
- Mode description
- Panel group
- Product definition
- Reference code translate table
- Index search
- Session description
- S/36 machine description.

Moving Objects from One Library to Another

You can use the Move Object (MOVOBJ) command to move objects between libraries. Moving objects from one library to another is useful in that you make an object temporarily unavailable and it lets you replace an out-of-date version of an object with a new version. For example, a new master file can be created to be temporarily placed in a library other than the one containing the old master file. Because the data in the old master file is normally copied to the new master file, the old master file cannot be deleted until the new master file has been created. Then, the old master file can be deleted and the new master file can be moved to the library that contained the old master file.

You can only move an object if you have object management authority for the object, delete and read authority for the library the object is being moved from, or add 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.

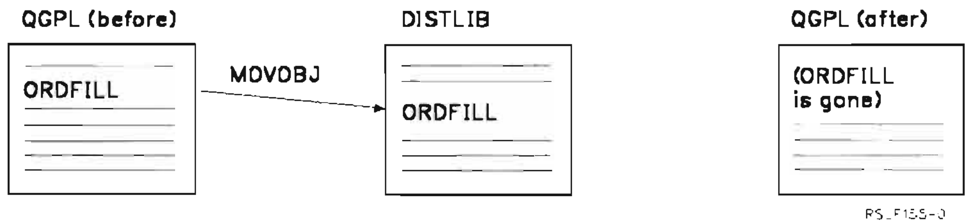
Moving journals and journal receivers is limited to moving these object types back into the library in which they were originally created. If the journal objects have been placed into QRCL by a Reclaim Storage (RCLSTG) command, they must be moved back into their original library to be made operational.

The following is a list of objects that *cannot* be moved:

- Authorization lists
- Class-of-service descriptions
- Configuration lists
- Controller descriptions
- Data dictionaries
- Device descriptions
- Display station message queues
- Edit descriptions
- DBCS font tables
- Libraries
- Line descriptions
- Mode descriptions

- System/36 machine descriptions
- The system history log (QHST)
- The system operator message queue (QSYSOPR)
- User profiles
- Documents
- Document lists
- Folders.

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 programs may depend on the command definitions of the commands used in the program to be in the same library at program run time as they were at program creation time. At program creation time and at program 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 program run time as they were at program creation time. If *LIBL is specified, the command definitions can be moved between program creation 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 and message files.
- 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 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, particularly for job and output queues.

Creating Duplicate Objects

You can use the Create Duplicate Object (CRTDUPOBJ) command to create a copy of an existing object. The duplicate object has the same object type and authorization as the original object and is created into the same auxiliary storage pool (ASP) as the original object. The user who issues the command owns the duplicate object.

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.

You can duplicate an object if you have read, object operational, and object management authority for the object, operational and add authority for the library in which the duplicate object is to be placed, object operational 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 save files are duplicated. Job queues and output queues cannot be duplicated into the temporary library (QTEMP). For a physical 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
- Configuration lists
- Controller descriptions
- Data dictionaries
- Device descriptions
- Data queues
- Documents
- Document lists
- Edit descriptions
- Folders
- DBCS font tables
- Journals
- Journal receivers
- Libraries

- Line descriptions
- Mode descriptions
- Reference code translation tables
- System/36 machine descriptions
- User profiles
- User queues.

In some cases, you may want to duplicate only some of the data in a file by following the CRTDUPOBJ command with a CPYF command that specifies the selection values.

The following command creates a duplicate copy of the order header physical file, and duplicates the data in the file:

```
CRTDUPOBJ OBJ(ORDHDRP) FROMLIB(DSTPRODLIB) OBJTYPE(*FILE) +
          TOLIB(DISTLIB2) NEWOBJ(*SAME) DATA(*YES)
```

When you create a duplicate object, you should consider the consequences of creating a duplicate of an object that refers to another object. Many objects refer to other objects by name, and many of these references are qualified by a specific library name. Therefore, the duplicate object could contain a reference to an object that exists in a library different from the one in which the duplicate object resides. For all object types other than files, references to other objects are duplicated in the duplicate object. For files, the duplicate objects share the formats of the original file.

Any physical files which exist in the from-library, and on which a logical file is based, must also exist in the to-library. The record format name and record level ID of the physical files in the to- and from-libraries are compared; if the physical files do not match, the logical file is not duplicated.

If a logical file uses a format selection that exists in the from-library, it is assumed that the format selection also exists in the to-library.

Renaming Objects

You can use the Rename Object (RNMOBJ) command to rename objects. However, you can rename an object only if you have object management authority for the object and update and read 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
- Configuration lists
- Controller descriptions
- Data dictionaries
- Device descriptions
- Display station message queues
- Journals
- Journal receivers
- The system library, QSYS, and the temporary library, QTEMP

- Line descriptions
- Mode descriptions
- System/36 machine descriptions
- The system history log (QHST)
- The system operator message queue (QSYSOPR)
- User profiles
- Documents
- Document lists
- Folders
- DBCS font tables.

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 being renamed is in use when you issue the RNMOBJ command, the command is not run and 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 program run time as they were at program creation 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, or data dictionary.

An object referring to another object may be dependent on the object and library names (even though *LIBL can be specified for the library name). Therefore, if you rename an object, you should change any references to it in other objects. See "Moving Objects from One Library to Another" on page 4-29 for a list of objects that refer to other objects.

If you rename a physical or logical file, the members in the file are not renamed. However, you can use the Rename Member (RNMM) command to rename a physical or logical file member.

Note: You should be careful when renaming objects in the system library QSYS. These objects are necessary for the system to perform effectively and the system must be able to find the objects. This is also true for some of the objects in the general-purpose library QGPL.

Deleting Objects

To delete an object, you can use a delete (DLTxxx) command for that type of object or you can use the delete option on the Work with Objects display (shown from the Work with Libraries (WRKLIB) display). To delete an object, you must have object existence authority to the object and read 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 in a command, 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.

The create program commands have a REPLACE parameter to allow users to continue using the old version of a program that has been replaced. The old versions of these re-created programs are stored in library QRPLOBJ.

You should be careful of deleting objects that exist in the system library QSYS. These objects are necessary for the system to perform properly.

On most delete commands, you can specify a generic name in place of an object name. Before using a generic delete, you may want to use a generic DSPOBJD command to verify that the generic delete will only delete the objects you want to delete. See "Using Generic Object Names" on page 4-11 for more information on specifying objects generically.

For information about deleting libraries, see "Deleting and Clearing Libraries" on page 4-16.

Allocating Resources

Objects are allocated on the system to guarantee integrity and to promote the highest possible degree of concurrency. An object is protected even though several operations may be performed on it at the same time. For example, an object is allocated so that two users can read the object at the same time or one user can only read the object while another can read and update the same object.

OS/400 allocates objects by the function being performed on the object. For example:

- If a user is displaying or dumping an object, another user can read the object.
- If a user is changing, deleting, renaming, or moving an object, no one else can use the object.
- If a user is saving an object, someone else can read the object, but not update or delete it; if a user is restoring the object, no one else can read or update the object.
- If a user is opening a database file for input, another user can read the file. If a user is opening a database file for output, another user can update the file.
- If a user is opening a device file, another user can only read the file.

Generally, objects are allocated on demand; that is, when a job step needs an object, it allocates the object, uses the object, and deallocates the object so another job can use it. The first job that requests the object is allocated the object. In your program, you can handle the exceptions that occur if an object cannot be allocated by your request. (See Chapter 7 and Chapter 8 for more information on monitoring for messages or your high-level language reference manual for information on handling exceptions.)

Sometimes you want to allocate an object for a job before the job needs the object, to ensure its availability so a function that has only partially completed would not have to wait for an object. This is called preallocating an object. Using the Allocate Object (ALCOBJ) command, you can preallocate the following types of objects. (Object types that are not included in this list cannot be allocated.)

- Authorization lists
- C locale description
- Cross-system product map
- Cross-system product table
- Data areas
- Data dictionaries
- Data queues
- Device descriptions, which can be allocated only if the device is powered on and varied on (except for APPC devices, which cannot be allocated by the ALCOBJ command)
- Database files (each member in a database file must be allocated with a separate ALCOBJ command)
- Font resource
- Form definition
- Forms control tables
- Libraries
- Menus
- Message queues
- Overlay
- Page segments
- Panel groups
- Print descriptor groups
- Programs
- Query manager forms
- Query manager queries
- Search indexes
- Session descriptions
- Subsystem descriptions
- System/36 machine descriptions
- User indexes
- User queues
- User spaces.

Objects are allocated on the basis of their intended use (read or update) and whether they can be shared (used by more than one job). The file and member are always allocated *SHRRD and the file data is allocated with the level of lock specified with the lock state. A lock state identifies the use of the object and whether it is shared. The five lock states are (parameter values given in parentheses):

- Exclusive (*EXCL). The object is reserved for the exclusive use of the requesting job; no other jobs can use the object. However, if the object is already allocated to another job, your job cannot get exclusive use of the object. This lock state is appropriate when a user does not want any other user to have access to the object until the function being performed is complete.
- Exclusive allow read (*EXCLRD). The object is allocated to the job that requested it, but other jobs can read the object. This lock is appropriate when a user wants to prevent other users from performing any operation other than a read.
- Shared for update (*SHRUPD). The object can be shared either for update or read with another job. That is, another user can request either a shared-for-read lock state or a shared-for-update lock state for the same object. This lock state is appropriate when a user intends to change an object but wants to allow other users to read or change the same object.
- Shared no update (*SHRNUP). The object can be shared with another job if the job requests either a shared-no-update lock state, or a shared-for-read lock state. This lock state is appropriate when a user does not intend to change an object but wants to ensure that no other user changes the object.
- Shared for read (*SHRRD). The object can be shared with another job if the user does not request exclusive use of the object. That is, another user can request an exclusive-allow-read, shared-for-update, shared-for-read, or shared-no-update lock state.

Note: The allocation of a library does not restrict the operations that can be performed on the objects within the library. That is, if one job places an exclusive-allow-read or shared-for-update lock state on a library, other jobs can no longer place objects in or remove objects from the library; however, the other jobs can still update objects within the library.

The following table shows the valid lock state combinations for an object:

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

The following table shows the lock states you can specify by object type:

Object Type	*EXCL	*EXCLRD	*SHRUPD	*SHRNUP	*SHRRD
Authorization list	x	x	x	x	x
C locale description	x	x	x	x	x
Cross-system product map	x	x	x	x	x
Cross-system product table	x	x	x	x	x
Data area	x	x	x	x	x
Data dictionary	x	x	x	x	x
Data queue	x	x	x	x	x
Device description		x			
File	x	x	x	x	x
Font resource	x	x	x	x	x
Form definition	x	x	x	x	x
Forms control table	x	x	x	x	x
Library		x	x	x	x
Menu	x	x	x	x	x
Message queue	x				x
Overlay	x	x	x	x	x
Panel group	x	x			
Program	x	x			x
Search index	x	x	x	x	x
Session description	x	x	x	x	x
Subsystem description	x				
System/36 machine description	x	x	x	x	x

To allocate an object, you must have object existence authority, object management authority, or operational authority for the object. Allocated objects are automatically deallocated at the end of a routing step. To deallocate an object at any other time, use the Deallocate Object (DLCOBJ) command.

You can allocate a program before it is called to protect it from being deleted. To prevent a program from running in different jobs at the same time, an exclusive lock must be placed on the program in each job before the program is called in any job.

You cannot use the ALCOBJ or DLCOBJ commands to allocate an APPC device description.

The following example is a batch job that needs two files members for updating. Members from either file can be read by another program while being updated, but no other programs can update these members while this job is running. The first member of each file is preallocated with an exclusive-allow-read lock state.

```
//JOB  JOBD(ORDER)
  ALCOBJ  OBJ((FILEA *FILE *EXCLRD) (FILEB *FILE *EXCLRD))
  CALL  PROGX
//ENDJOB
```

Objects that are allocated to you should be deallocated as soon as you are finished using them because other users may need those objects. However, allocated objects are automatically deallocated at the end of the routing step.

If the first members of FILEA and FILEB had not been preallocated, the exclusive-allow-read restriction would not have been in effect. When you are using files, you may want to preallocate them so that you are assured they are not changing while you are using them.

Note: If a single object has been allocated more than once (by more than one allocate command), a single DLCOBJ command will not completely deallocate that object. One deallocate command is required for each allocate command.

It is not an error if the DLCOBJ command is issued against an object where you do not have a lock or do not have the specific lock state requested to be allocated.

You can change the lock state of an object, as the following example shows:

```
PGM
ALCOBJ  OBJ((FILEX *FILE *EXCL)) WAIT(0)
CALL PGMA
ALCOBJ  OBJ((FILEX *FILE *EXCLRD))
DLCOBJ  OBJ((FILEX *FILE *EXCL))
CALL PGMB
DLCOBJ  OBJ((FILEX *FILE *EXCLRD))
ENDPGM
```

File FILEX is allocated exclusively for PGMA, but FILEX is allocated as exclusive-allow-read for PGMB.

You can use record locks to allocate data records within a file. See the *Database Guide* for more information on record locks. You can also use the WAITFILE parameter on a Create File command to specify how long your program is to wait for that file before a time-out occurs.

The WAITRCD parameter on a Create File command specifies how long to wait for a record lock. The DFTWAIT parameter on the Create Class (CRTCLS) command specifies how long to wait for other objects. For a discussion of the WAITRCD parameter, see the *Backup and Recovery Guide*.

For a description of the record locking functions provided by commitment control, see the *Database Guide*.

Displaying the Lock States for Objects

You can use the Work with Object Locks (WRKOBJLCK) command or the Work with Job (WRKJOB) command to display the lock states for objects.

The WRKOBJLCK command displays all the lock state requests in the system for a specified object. It displays both the held locks and the locks being waited for. For a database file, the WRKOBJLCK command displays the locks at the file level (the object level) but not at the record level. For example, if a database file is open for update, the lock on the file is displayed, but the lock on any records within the file is not.

If you use the WRKJOB command, you can select the locks option on the Display Job menu. This option displays all the lock state requests outstanding for the specified active job, the locks being held by the job, and the locks for which the job is waiting. However, if a job is waiting for a database record lock, this does not appear on the object locks display.

The following command displays all the lock state requests in the system for the logical file ORDFILL:

```
WRKOBJLCK  OBJ(QGPL/ORDFILL)  OBJTYPE(*FILE)
```

The resulting display is:

```
                              Work with Object Locks
                                                              System:   SN020024
Object:   ORDFILL        Library:   QGPL           Type:    *FILE-LGL

Type options, press Enter.
  1=Work with job   2=Work with job locks   4=End job

Opt     Job            User           Lock        Status
 _      WORKST04       QSECOFR        *SHRRD       HELD
                                      *SHRRD       HELD
                                      *SHRRD       HELD
                                      *SHRRD       HELD
                                      *SHRRD       HELD
                                      *SHRRD       HELD
                                      *SHRRD       HELD
                                      *SHRRD       HELD
                                      *SHRRD       HELD
                                      *SHRRD       HELD
                                      *SHRRD       HELD
                                      *SHRRD       HELD
                                      *SHRRD       HELD
                                                                         More...
F3=Exit   F5=Refresh   F9=Work with member locks   F12=Cancel
```

Chapter 5. Working with Objects in CL Programs

The rules for referring to objects in commands in CL programs are the same as for referring to objects in commands that are processed individually (not within a program). Object names can be either qualified or unqualified. Whenever an object is unqualified, it is found through a search of the library list.

Most objects referred to in CL programs are only accessed at command run time. If you qualify the name (*library/name*) of an object to a specific library in your CL source statement, that object must be in the specified library when the command referring to it is run, but the object does not have to be in that library when the program is created. This means that most objects can be qualified in CL source statements based only on their run-time location. Files declared on a Declare File (DCLF) command and command definitions are exceptions to this, because they are accessed at creation time too. Therefore, when either of these objects are qualified at creation time, they must be in the specified library at creation time and they are bound to that library at run time. These exceptions are discussed under "Accessing Objects in CL Programs."

You can avoid this run time consideration for all objects if you do not qualify object names on CL source statements, but refer to the library list (*LIBL/name) instead. If you refer to the library list at creation 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 program creation and command processing.

Accessing Objects in CL Programs

Most objects referred to in CL programs are not accessed until the command referring to them is run. This means that for most objects referred to in a program, the object does not have to exist when the program is compiled, but only when the individual command within the CL program (which refers to the object) is processed. Files and command definitions are exceptions to this rule, because they are also accessed at compile time; they must exist before a program referring to them is compiled. This is why their qualified names, if specified at compile time, must refer to their actual library location. This is not true for all objects, and it is not true for any objects if *LIBL is used.

Because most objects do not have to exist until the command referring to them is run, the following 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 the program TEST is called, but only when the CALL command is processed. Thus the called program can be created within the calling program immediately prior to the CALL command:

```
PGM   /* TEST */
DCL ...
.
.
.
MONMSG
.
.
.
CRTCLPGM  PGM(QGPL/PAYROLL)
CALL  PGM(QGPL/PAYROLL)
.
.
.
ENDPGM
```

Note that for create commands, such as CRTCLPGM or CRTDTAARA, the object accessed at creation time or run time is the create command definition, not the object being created. That is, if you are using a create command, the create command definition must be in the library where it was created if it was qualified at creation time, or it must be in a library on the library list if you used *LIBL. To avoid this type of complication, do not qualify IBM-supplied commands; use the library list instead.

Exceptions: Accessing Command Definitions and Files

When creating a CL program from source statements that refer to command definitions or files, the objects must exist at creation time, and when the command referring to them is processed. This means that when you use the Receive File (RCVF) command, you must create the file before creating a program referring to that file. After the program is created, you can delete the file, as long as it exists when the command referring to it is processed.

Accessing Command Definitions

Command definitions are accessed at creation time and at command run time. The command must exist when a program using it is created (to allow for statement syntax checking). If it is qualified at creation time, it must be in the referenced library during creation, and in the same library when it is processed. If it is not qualified, it must be in some library on the library list during creation and at run time.

If the command's definition might not be accessible through the library list when the program is running, or if there are multiple command definitions with the same name, the command name should be qualified in the program.

The name of the command must be the same when the program is processed as it was when the program was created. If the name of a command is changed after a program referring to that command is created, the program cannot find the command when it runs, and an error occurs. However, if a default is changed for a parameter on a command, the new default will be used when that command is processed. For more detail on attributes that can be changed on a command without re-creating the command, see "Effect of Changing the

Command Definition of a Command in a Program" on page 9-47 and the Change Command (CHGCMD) command description in the *CL Reference*.

Accessing Files

Files are also accessed when the command referring to them is compiled, and they can be handled in the same way as the data area in the example in the beginning of this section. That is, the file must exist before a CL program using it is created.

To create the file, DDS must be entered into a source file first. (DDS is not required, but may be used for a physical file referred to by a CL program.) The DDS describes the record formats and the fields within the records, and this information is compiled to create the file object using the Create Display File (CRTDSPF) command. The fields described in the DDS can be input or output fields (or both), and they are declared to the CL program as variables when the program (specifically, the DCLF command in the program) is compiled. It is through these variables that data from the display is manipulated by the program.

If DDS is not used to create a physical file, a CL variable is declared to the program 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 cannot manipulate data in any types of files other than display files and database files, except with specific CL commands.

After the file is created, you may delete the DDS, though it is not recommended. After the CL program referring to the file is compiled, you can delete the file too, so long as it exists when the command referring to it, such as a Declare File (DCLF), Send File (SNDF), or Receive File (RCVF), is processed in the program.

The rules on qualified names described here for data areas and command definitions also apply to files. For more details on files, see "Working with Files in CL Programs" on page 5-4.

Checking for the Existence of an Object

Before you try to use an object in a CL program, you can check to determine if the object exists and if you have the authority needed to use it. This is useful when more than one object is used by a function 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 program. The CHKOBJ command has the following format:

```
CHKOBJ  OBJ(library-name/object-name)  OBJTYPE(object-type)
```

Other optional parameters allow object authorization verification. If you are checking for authorization and intend to open a file, you should check for both operational and data authority.

When this command is processed, messages are sent to your program to report the result of the object check. You can monitor for these messages and handle them as you wish. For example:

```
           CHKOBJ  OELIB/PGMA  *PGM
           MONMSG  MSGID(CPF9801)  EXEC(GOTO NOTFOUND)
           CALL OELIB/PGMA
           .
           .
           .
NOTFOUND:  CALL FIX001  /* PGMA Not Found Routine */
           ENDPGM
```

In this example, the MONMSG command checks only for the object-not-found escape message. For a list of all the messages which may be sent by the CHKOBJ command, see the *CL Reference*. Additional information about monitoring for messages can be found under "Using the Monitor Message (MONMSG) Command" on page 2-39 and in Chapter 7 and Chapter 8.

The CHKOBJ command does not allocate an object. In many application uses, the check for existence is not an adequate function, and allocation should be performed. The Allocate Object (ALCOBJ) command provides both an existence check and allocation.

You can use the Check Tape (CHKTAP) or Check Diskette (CHKDKT) command in a CL program to ensure that a specific tape or diskette is placed on the drive and ready. These commands also provide an escape message that you can monitor for in your CL program.

Working with Files in CL Programs

Two types of files are supported in CL programs, display files and database files. You can send a display to a work station and receive input from the work station for use in the program, or you can read data from a database file for use in the program.

Note: Database files are made available for use within the CL program through the DCLF and RCVF commands. See the discussion on opening and closing database files in the *Database Guide*; it contains information about the OPNDBF and CLOF commands, which make database files available for later use in High-Level Language (HLL) programs.

To use a file in a CL 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 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 program is processed, but does not need to have a member when the file is referred to create the program.

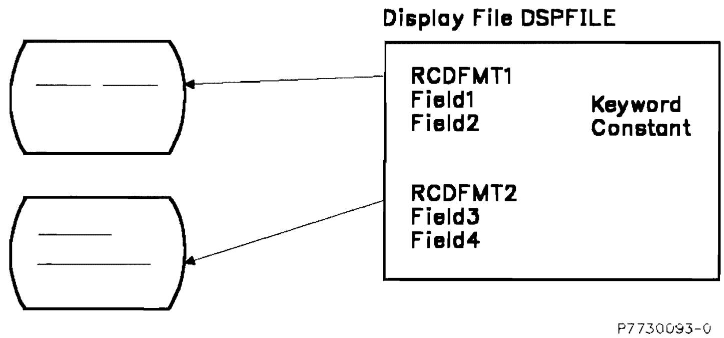
- Refer to the file in the CL program using the DCLF command, and refer to the record format on the appropriate data manipulation CL commands in your CL source.
- Create the CL program.

Only one display or database file can be referred to in a CL program. 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 programs:
  - Only database files with a single record format may be used by a CL program.
  - The file may be either a physical or logical file, and a logical file may be defined over multiple physical file members.
  - Only input operations, with the RCVF command, are allowed. The SNDF, SNDRCVF, ENDRCV, WAIT and DEV parameters on the RCVF command are not allowed for database files.
  - DDS is not required to create a physical file which is referred to in a CL 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 referred to create the program. It must have a member when the program is processed.
  - The file is opened for input only when the first RCVF command is processed.
  - The file remains open until the program returns or transfers control, or when the end of file is reached. When end of file is reached, message CPF0864 is sent to the CL program, and additional operations are not allowed for the file. The 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 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 program returns or transfers control.

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 program and an override can be performed before the first send or receive.

The format for the display is identified as a record format in DDS. Each record format may contain fields (input, output, and input/output), conditions/indicators, and constants. Several record formats can be entered in one display file. The display file name, record format name, and field names should be unique, because other HLLs may require it, even though CL programs do not.



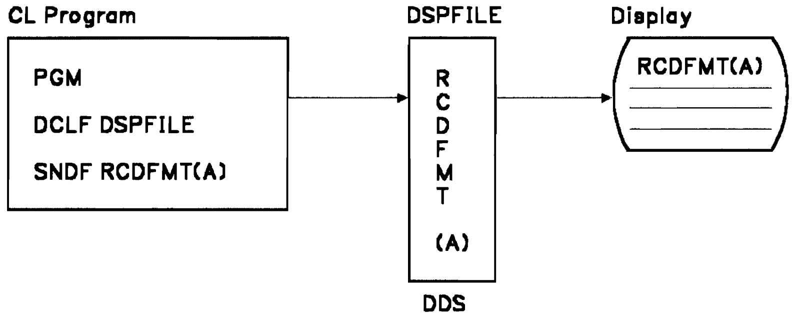
See the *DDS Reference* for more information about specifying DDS.

You can use the methods discussed in Chapter 10 or the Screen Design Aid (SDA) to enter DDS source for records and fields in the display file. See the *SDA User's Guide/Reference* for detailed information about interactive screen design.

A CL 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 let you refer to a display 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 program. The fields in the file are automatically declared as variables for use in the program.
- Send File (SNDF). Sends data to the display.
- Receive File (RCVF). Receives data from the display or database.
- Send/Receive File (SNDRCVF). Sends data to the display; then asks for input and, optionally, receives data from the display.
- Override with Display File (OVRDSPF). Allows a run-time override of a file used by a program with a display file.
- Override with Database File (OVRDBF). Allows a run-time override of a file used by a program with a database file.

These commands let a running CL 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 program to use the fields, the file must be referred to in the CL program by the DCLF command. This reference causes the fields and indicators in the file to be declared automatically in your program as variables. You can use these variables in any way in CL commands; however, their primary purpose is to send information to and receive information from a display. The DCLF command is not used at run time.

The format of the display and the options for the fields are specified in the device file and controlled through the use of indicators. Up to 99 indicator values can be used with DDS and CL support. Indicator variables are declared in your CL program in the form of logical variables with names &IN01 through &IN99 for each indicator that appears in the device file record formats referred to on the DCLF command. Indicators let you display fields and control data management display functions, and provide response information to your program from the device display. Indicators are not used with database files.

Referring to Files in a CL Program

Files are accessed during compiling of DCLF commands when CL programs are created so that variables can be declared for each field in the file.

If you have qualified the name of the file at compile time, the file must be in that library at run time. If you have used the library list at compile time, the file must be in a library on the library list at run time.

Opening and Closing Files in a CL Program

When you use CL support, you can refer to only one file in a program. 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 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 program in which it was opened returns or transfers control. Once a database file has been closed, it cannot be opened again during the same call of the program.

When a database file is opened, the first member in the file is opened, unless an OVRDBF command was previously used to specify a different member (MBR parameter). If a program ends because of an error, files are closed. Because a file remains open until the program in which that file was opened completes processing, you have an easy way to share open data paths between running programs. A file can be opened in one program and then its open data path can be shared with another 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 specifying SHARE(*YES) is in effect.

Files can be shared in this way between any two programs. For a detailed description of the function available when open data paths are shared, see the description of the SHARE parameter on the CRTDSPF, CRTPF, and CRTLF commands in the *CL Reference*. When a display file is opened in a CL program, it is always opened for both input and output. When a database file is opened in a CL program, it is opened for input only.

Do not specify LVL(*CALLER) on the Reclaim Resources (RCLRSC) command in CL programs using files. If you specified LVL(*CALLER), all files opened by the 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 program. The DCLF command cannot be used to declare files, such as tape, diskette, printer, and mixed files. Only one DCLF command is allowed in a CL program. The DCLF command has the following parameters:

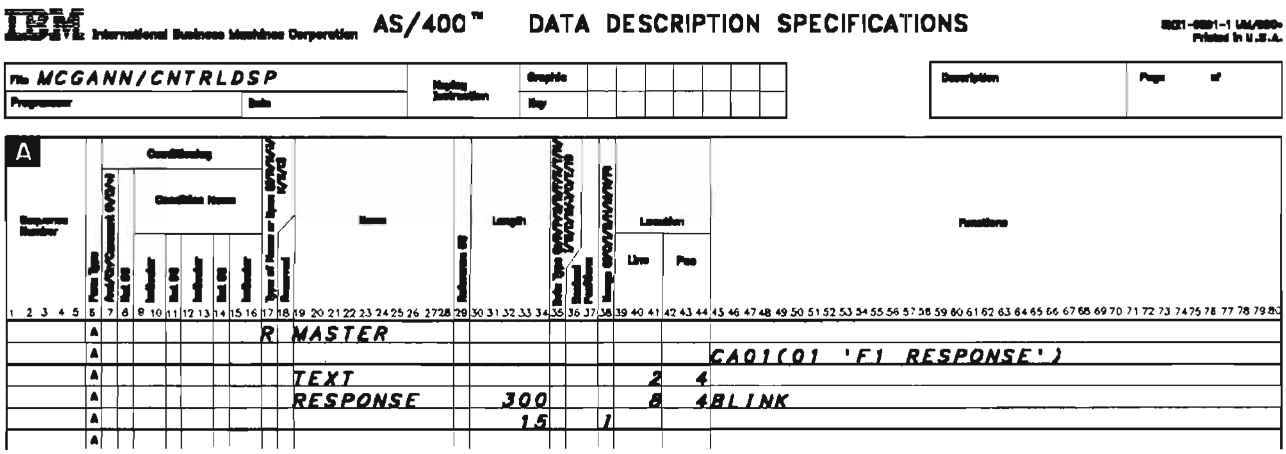
```
DCLF   FILE(library-name/file-name)
       RCDFMT(record-format-names)
```

Note that the file must exist before the program is compiled.

If you are using a file in your program, you may have to specify input/output fields in your DDS. These fields are handled as variables in the program. When a DCLF command is used, the CL compiler declares CL variables for each field and option indicators 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 preceded by &IN.

For example, if a field named INPUT and indicator 10 are defined in DDS, the DCLF command automatically declares them as &INPUT and &IN10. This declaration is performed when the CL 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:



RSLF162-3

Three variables, &IN01, &TEXT, and &RESPONSE, would be available from the display file. In a CL program 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:

```
 •
 •
 •
500-            DCLF      MCGANN/CNTRLDSP
     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
 •
 •
 •
```

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 programs are the SNDF, RCVF, and SNDRCVF commands.

When you run an SNDF command, the content of the variables associated with the output or output/input fields in the record format you specify is formatted by system and sent to the display device. Similarly, when you run an RCVF command, the values of the fields associated with input or output/input fields in the record format on the display are placed in the corresponding CL variables.

The SNDRCVF command sends the contents of the CL variables to the display and then performs the equivalent of an 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 defined as zoned decimal cause *DEC fields to be defined in the CL 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. Record formats that contain floating point data cannot be used in a CL program.

Note: If a SNDRCVF or RCVF command for a work station indicates WAIT(*NO), or if a SNDF command is issued using a record format containing the INVITE DDS keyword, then the WAIT command is used to receive data.

Except for message subfiles, any attempt to send or receive subfile records causes run-time errors. Most other functions specified for display files in DDS are available; some functions (such as using variable starting line numbers) are not. For more information on messages and subfiles in CL programs, see Chapter 8.

The following example shows the steps required to create a typical operator menu and to send and receive data using the SNDRCVF command. The menu looks like this:

```
Operator Menu

  1. Accounts Payable
  2. Accounts Receivable
  90. Signoff

Option:
```

First, enter the following DDS source. The record format is MENU, and OPTION is an input-capable field. The OPTION field uses DSPATR(MDT). This causes the system to check this field for valid values even if the operator does not enter anything.

```
     A          R MENU
     A                                      1  2'Operator Menu'
     A                                      3  4'1. Accounts Payable'
     A                                      4  4'2. Accounts Receivable'
     A                                      5  4'90. Signoff'
     A                                      7  2'Option:'
     A            OPTION         2Y 0I     +2VALUES(1 2 90) DSPATR(MDT)
```

PSLF172-1

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/400.

The display file could also be created using the Screen Design Aid (SDA) utility.

Next, enter the CL source to run the menu.

The CL source for this menu is:

```
         PGM /* OPERATOR MENU */
         DCLF INTMENU
BEGIN:   SNDRCVF RCDFMT(MENU)
         IF COND(&OPTION *EQ 1) THEN(CALL ACTSPAYMNU)
         IF COND(&OPTION *EQ 2) THEN(CALL ACTSRCVMNU)
         IF COND(&OPTION *EQ 90) THEN(SIGNOFF)
         GOTO BEGIN
         ENDPGM
```

When this source is compiled, the DCLF command automatically declares the input field OPTION in the program as a CL variable.

The SNDRCVF command defaults to WAIT(*YES); that is, the program waits until input is received by the program.

Writing a CL Program to Control a Menu

The following example shows how a CL program can be written to display and control a menu. See Chapter 10 for another method of creating and controlling menus.

This example shows a CL program, ORD040C, that controls the displaying of the order department general menu and determines which HLL program to call based on the option selected from the menu. The program 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:

IBM International Business Machines Corporation — AS/400™ DATA DESCRIPTION SPECIFICATIONS

A

```
     A* MENU ORD040CD 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 an existing order'
     A                                  8  3'6 Order entry'
     A                                  9  2'98 End of menu'
     A                                 11  2'Option:'
     A            RESP           2Y00I 11 10VALUES(1 2 3 4 5 6 98)
     A                                      DSPATR(MDT)
     A
     A
     A
     A
     A
     A
     A
     A
```

RSLF173-1

The source program for ORD040C looks like this:

```
       PGM /* ORD040C Order Dept General Menu */
       DCLF FILE(ORD040CD)
START: SNDRCVF RCDFMT(MENU)
       IF (&RESP=1) THEN(CALL CUS210)
       /* Customer inquiry */
       ELSE +
         IF (&RESP=2) THEN(CALL ITM210)
         /*Item inquiry*/
         ELSE +
           IF (&RESP=3) THEN(CALL CUS220)
           /* Cust name search */
           ELSE +
             IF (&RESP=4) THEN(CALL ORD215)
             /* Orders by cust */
             ELSE +
               IF (&RESP=5) THEN(CALL ORD220)
               /* Existing order */
               ELSE +
                 IF (&RESP=6) THEN(CALL ORD410C)
                 /* Order entry */
                 ELSE +
                   IF (&RESP=98) THEN(RETURN)
                   /* End of Menu */
       GOTO START
       ENDPGM
```

The DCLF command indicates which file contains the field attributes the system needs to format the order department general menu when the SNDRCVF command is processed. The system automatically declares a variable for each field in the record format in the specified file if that record format is used in an SNDF, RCVF, or SNDRCVF command. The variable name for each field automatically declared is an ampersand (&) followed by the field name. For example, the variable name of the response field RESP in ORD040C is &RESP.

Other notes on the operation of this menu:

- The SNDRCVF command is used to send the menu to the display and to receive the option selected from the display.
- If the option selected from the menu is 98, ORD040C returns to the program that called it.
- The ELSE statements are necessary to process the responses as mutually exclusive alternatives.

Note: This menu is run using the CALL command. See Chapter 10 for a discussion of those menus run using the GO command.

Overriding Display Files in a CL Program

You can use the Override with Display File (OVRDSPF) command to replace the display file named in a CL program or to change certain parameters of the existing display file. This may be especially useful for files that have been renamed or moved since the program was compiled.

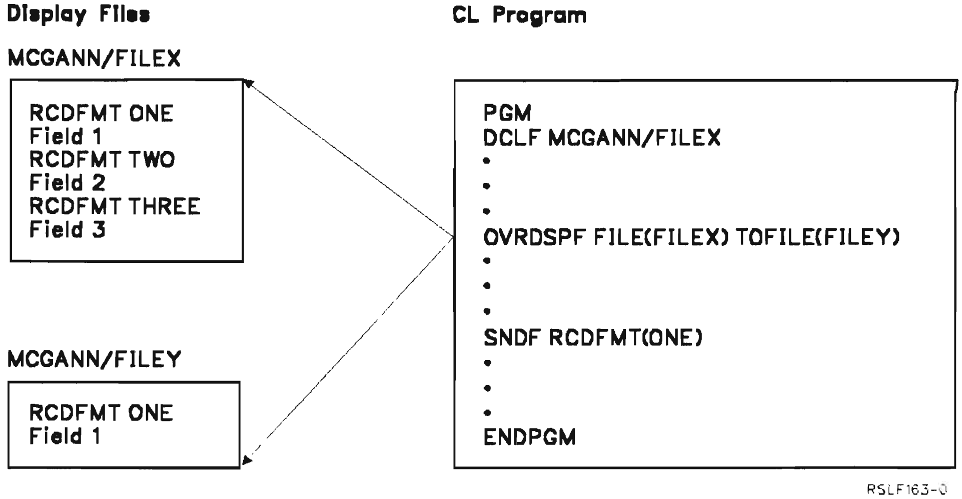
The initial parameters of the OVRDSPF command are:

```
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 program only if the file specified on the DCLF command was a display file when the program was created. The file used when the program is run must be of the same type as the file referred to when the program was created.

The OVRDSPF command must be run before the file to be overridden is opened for use. An open is caused by the first use of a send or receive command. The file is overridden if it is opened in the program containing the OVRDSPF command, or if it is opened in another program to which control is transferred by the TFRCTL or CALL command. See the *Data Management Guide* for more information about overriding files.

When you override to a different file, only those record format names referred to in the CL program on the SNDF, RCVF, or SNDRCVF command need to be in the overriding file. In the following illustration, display file FILEY does not need record format TWO or THREE.



You should make sure that the record format referred to names of the original file and the overriding files have the same field definitions and indicator names in the same order. You may get unexpected results if you specify LVLCHK(*NO).

Another consideration has to do with the DEV parameter on the SNDF, RCVF, and SNDRCVF commands when an OVRDSPF command is applied. If *FILE is specified on the DEV parameter of the RCVF, SNDF, or SNDRCVF command, the system automatically directs the operation to the correct device for the overridden file. If a specific device is specified on the DEV keyword of the RCVF, SNDF, or SNDRCVF command, one of the following may occur:

- If a single device display file is being used, an error will occur if the display file is overridden to a device other than the one specified on the RCVF, SNDF, or SNDRCVF command.
- If a multiple device display file is being used, an error will occur if the device specified on the RCVF, SNDF, or SNDRCVF command is not among those specified on the OVRDSPF command.

Working with Multiple Device Display Files

The normal mode of operation on a system is for the work station 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, including the display file in the program. Each requester calls a separate job in this kind of use. This is not considered to be multiple device display use.

A multiple device display configuration occurs when a single job called by one requester communicates with multiple display stations through one display file. While only one display file can be handled by a CL program, the display file, or different record formats within it, can be sent to several device displays. Commands used primarily with multiple device display files are:

- End Receive (ENDRCV). This command cancels requests for input that have not been satisfied.
- Wait (WAIT). Accepts input from any device display from which user data was requested by one or more previous RCVF or SNDRCVF commands when WAIT(*NO) was specified on the command, or by one or more previous SNDF commands to a record format containing the INVITE DDS keyword.

If you use a multiple device display file, the device names must be specified on the DEV parameter on the CRTDSPF command when the display file is created, on the CHGDSPF command when the display file is changed, or on an override command, and the number of devices must be less than or equal to the number specified on the MAXDEV parameter on the CRTDSPF command.

Multiple device display configurations affect the SNDRCVF and the RCVF commands and you may need to use the WAIT or ENDRCV commands. When an RCVF or SNDRCVF command is used with multiple display devices, the default value WAIT(*YES) prevents further processing until an input-capable field is returned to the program from the device named on the DEV parameter. Because the response may be delayed, it is sometimes useful to specify WAIT(*NO), thus letting your program continue running other commands before the receive operation is satisfied.

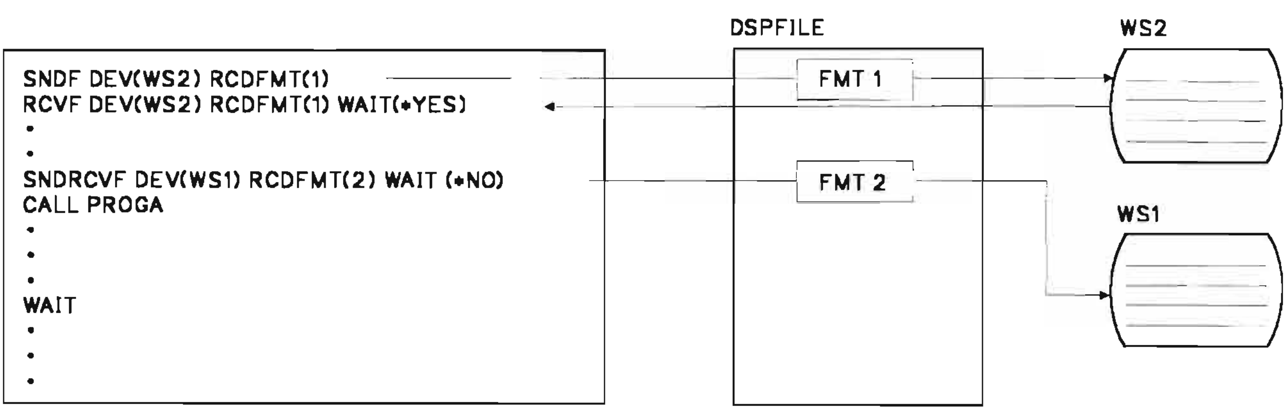
If you use an RCVF or SNDRCVF command and specify WAIT(*NO), the CL program continues running until a WAIT command is processed.

Using a SNDF command with a record format which has the DDS INVITE keyword is equivalent to using a SNDRCVF command with WAIT(*NO) specified. The DDS INVITE keyword is ignored for SNDRCVF and RCVF commands.

The WAIT command must be issued to access a data record. If no data is available, program processing is suspended until data is received from a device display or until the time limit specified in the WAITRCD parameter for the display file on the CRTDSPF, CHGDSPF, or OVRDSPF commands has passed. If the time limit passes, message CPF0889 is issued.

The WAIT will also be satisfied by the job being cancelled 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 above example, the first two commands show a typical sequence in which the default is taken; processing waits for the receive operation from WS2 to complete. Because WS2 is specified on the DEV parameter, the RCVF command does not run until WS2 responds, even if prior outstanding requests (not shown) from other stations are satisfied.

The SNDRCVF command, however, has WAIT(*NO) specified and so does not wait for a response from WS1. Instead, processing continues and PROGA is called. Processing then stops at the WAIT command until an outstanding request is satisfied by a work station, or until the function reaches time-out.

The WAIT command has the following format:

```
WAIT DEV(CL-variable-name)
```

If the DEV parameter is specified, the CL variable name is the name of the device that responded. (The default is *NONE.) If there are several receive requests (such as RCVF. . . WAIT(*NO)), this variable takes the name of the first device to respond after the WAIT command is encountered in the program, and processing continues. The data received is placed in the program 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 programs also support the ENDRCV command, which lets you cancel a request for input that has not yet been satisfied. A SNDF or SNDRCVF command will also cancel a request for input that has not yet been satisfied. However, if the data was available at the time the SNDF or SNDRCVF command was processed, message CPF0887 will be sent. In this case the data must be received with the WAIT command or RCVF command, or the request must be explicitly cancelled with a ENDRCV command before the SNDF or SNDRCVF command can be processed.

Receiving Data from a Database File

The only command you can use to receive data from a database file is the RCVF command.

When you run a RCVF command, the next record on the file's access path is read, and the values of the fields defined in the database record format are placed in the corresponding CL variables. Note that CL does not support zoned decimal or binary numbers. Consequently, fields in the database file defined as zoned decimal or binary cause *DEC fields to be defined in the CL 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 program.

When the end of file is reached, message CPF0864 is sent to the 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 will be sent again.

Overriding Database Files in a CL Program

You can use the Override with Database File (OVRDBF) command to replace the database file named in a CL program or to change certain parameters of the existing database file. This may be especially useful for files that have been renamed or moved since the program was created. It can also be used to access a file member other than the first member.

The initial parameters of the OVRDBF command are:

```
OVRDBF FILE(overridden-file-name) TOFILE(new-file-name)
       MBR(member-name)
```

The OVRDBF command is valid for a file referred to by a CL program only if the file specified in the DCLF command was a database file when the program was created. The file used when the program was processed must be of the same type as the file referred to when the program was created.

The OVRDBF command must be processed before the file to be overridden is opened for use (an open occurs by the first use of the RCVF command). The file is overridden if it is opened in the program containing the OVRDBF command, or if it is opened in another program to which control is transferred by the TFRCTL or CALL command. See the *Data Management Guide* for more information about the OVRDBF 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). See the *Database Guide* for more information about LVLCHK considerations.

Referring to Output Files from Display Commands

A number of the IBM display commands allow you to place the output from the command into a database file (OUTFILE parameter). The data in this file can be received directly into a CL program and processed. See the *Database Guide* for a discussion of some of these commands and the format descriptions.

The following CL program accepts two parameters, a user name and a library name. The program determines the names of all programs, files, and data areas in the library and grants normal authority to the specified users.

```
              PGM PARM(&USER &LIB)
              DCL &USER *CHAR 10
              DCL &LIB *CHAR 10
(1)   DCLF QSYS/QADSPOBJ
(2)   DSPOBJD OBJ(&LIB/*ALL) OBJTYPE(*FILE *PGM *DTAARA) +
      OUTPUT(*OUTFILE) OUTFILE(QTEMP/DSPOBJD)
(3)   OVRDBF QADSPOBJ TOFILE(QTEMP/DSPOBJD)
(4)   READ:  RCVF
(5)   MONMSG CPF0864 EXEC(RETURN)  /* EXIT WHEN END OF FILE REACHED  */
(6)   GRTOBJAUT OBJ(&ODLBNM/&ODOBNM) OBJTYPE(&ODOBTP) +
              USER(&USER) AUT(*CHANGE)
              GOTO READ        /*GO BACK FOR NEXT RECORD*/
              ENDPGM
```

(1) The declared file, QADSPOBJ in QSYS, is the IBM-supplied file that will be used by the DSPOBJD command. This file is the master file which is referenced by the command when creating the output file. It is referenced by the CL compiler to determine the format of the records and to declare variables for the fields in the record format.

(2) The DSPOBJD command creates a file named DSPOBJD in library QTEMP. This file has the same format as file QADSPOBJ.

(3) The OVRDBF command overrides the declare file (QADSPOBJ) to the file created by the DSPOBJD command.

(4) The RCVF command reads a record from the DSPOBJD file. The values of the fields in the record are copied into the corresponding CL variables, which were implicitly declared by the DCLF command. Because the OVRDBF command was used, the file QTEMP/DSPOBJD is read instead of QSYS/QADSPOBJ (the file QSYS/QADSPOBJ is not read).

(5) Message CPF0864 is monitored. This indicates that the end of file has been reached, so the program returns control to the calling program.

(6) The GRTOBJAUT command is processed, using the variables for object name, library name and object type, which were read by the RCVF command.

Chapter 6. Advanced Programming Topics

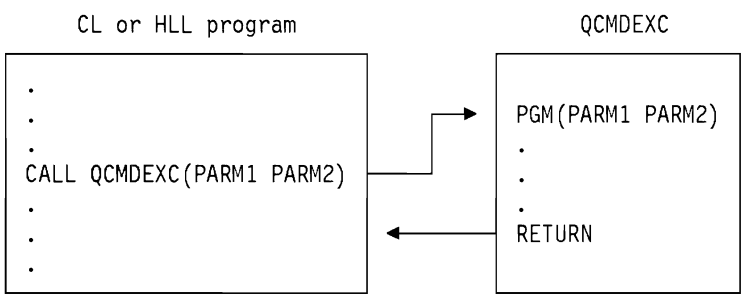
This chapter introduces more advanced programming topics, including:

- Special functions that can be called from high-level language programs (including CL programs)
- Using prompting and the Programmer Menu to enter program source

Several sample programs are included at the end of the chapter.

Using the QCMDEXC Program

QCMDEXC is an IBM-supplied program that runs a single command. It is used to run a command from within a high-level language (HLL) program or from within a CL 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 your HLL or CL program and the command it runs is passed to it as a parameter on the CALL command.



After the command runs, control returns to your HLL or CL program.

The command is run as if it were not in a program. Therefore, program variables cannot be used on the command; values cannot be returned by the command to CL variables; and commands that can only be used in CL 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)
```

The command you wish to run is entered as a character string on the first parameter. If the command contains blanks, it must be enclosed in apostrophes. The maximum length of the character string is 3000 characters; delimiters (the apostrophes enclosing the string) are never counted as part of the string. The length specified as the second value on the PARM parameter is the length of the character string passed as the command. Length must be a packed decimal value of length 15 with 5 decimal positions.

Thus, if you wish 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)
```

This statement could be coded into the HLL or CL program, and when the program runs, the library list would be replaced. Used this way, however, the QCMDEXC program does not provide run-time flexibility. That is accomplished by substituting variables for the constants in the parameter list, and specifying the values for the variables in the call to the HLL or CL program.

For instance:

```
CALL PGM(PGMA) PARM('CHGLIBL...' 31)
```

```
PGM PARM(&STRING &LENGTH) /* PGMA */
DCL &STRING *CHAR LEN(31)
DCL &LENGTH *DEC LEN(15 5)
.
.
.
CALL PGM(QCMDEXC) PARM(&STRING &LENGTH)
.
.
.
END
```

```
QCMDEXC
.
.
.
CHGLIBL...
.
.
.
```

The command length, which is passed to the QCMDEXC program on 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. 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.

The same command could be run using the following CALL command from the CL program because numeric literals are passed as packed decimal values of length 15 with 5 decimal positions:

```
CALL QCMDEXC (&STRING 31)
```

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

To determine whether a CL command can be passed on a call to the QCMDEXC program, look up the command in the *CL Reference*. To find the environment in which the command can be run, look at the syntax diagram for the command. The small box in the lower right corner of the diagram indicates the environment in which the command can be run. For example, `JOB: I` indicates that the command can be run interactively; `JOB: B` indicates that the command can be run in a batch job.

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 program using the Monitor Message (MONMSG) command. For more information about monitoring for messages, see Chapter 7 and Chapter 8.

If a syntax error is detected, message CPF0006 is sent. If an error is detected during the processing of a command, any escape message sent by the command is returned by the QCMDEXC program. You monitor for messages from commands run through the QCMDEXC program in the same way you monitor for messages from commands contained in CL programs.

See the appropriate HLL reference for information on how HLL programs handle errors on calls.

Using the QCMDEXC Program with DBCS Data

You can use QCMDEXC to request double-byte character set (DBCS) input data to be entered with a command. The command format used for QCMDEXC to prompt double-byte data is:

```
CALL QCMDEXC ('command' command-length IGC)
```

The third parameter of the QCMDEXC program, IGC, instructs the system to accept double-byte data. For example, the following CL program asks a user to provide double-byte text for a message. Then the system sends the following message:

```
PGM
   CALL QCMDEXC ('?SNDMSG' 7 IGC)
ENDPGM
```

An explanation of the system message follows:

- The `?` character instructs the system to present the command prompt for the Send Message (SNDMSG) command.
- The value `7` is the length of the SNDMSG command plus the question mark.
- The value `IGC` lets you 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   F11=KEYWORDS
 F12=PREVIOUS      F13=PROMPTER HELP
```

Using the QCMDCHK Program

QCMDCHK is an IBM-supplied program that performs syntax checking for a single command, and optionally prompts for the command. The command is not run. If prompting is requested, the command string is returned to the calling program with the updated values as entered through prompting. The QCMDCHK program can be called from a CL program or an HLL program.

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 work station user is placed in the variable.

The second parameter is the maximum length of the command string being passed. If the command string is passed as a quoted string, the command length is exactly the length of the quoted string. If the command string is passed in a variable, the command length is the length of the CL variable. The second parameter must be a packed decimal value of length 15 with 5 decimal positions.

The QCMDCHK program performs syntax checking on the command string which is passed to it. It verifies that all required parameters are coded, and that all parameters have allowable values. It does not check for the processing environment. That is, a command can be checked whether it is allowed in batch only, interactive only, or only in a batch or interactive CL program. QCMDCHK does not allow checking of command definition statements.

If a syntax error is detected on the command, message CPF0006 is sent. You can monitor for this message to determine if an error occurred on the command. Message CPF0006 is preceded by one or more diagnostic messages that identify the error. In the following example, control is passed to the label ERROR within the program, because the value 123 is not valid for the PGM parameter of the CRTCLPGM command.

```
CALL QCMDCHK ('CRTCLPGM PGM(QGPL/123)' 22)
MONMSG CPF0006 EXEC(GOTO ERROR)
```

You can request prompting for the command by either placing a question mark before the command name or by placing selective prompt characters before one or more keyword names in the command string.

If no errors are detected during checking and prompting for the command, the updated command string is placed in the variable specified for the first parameter. The prompt request characters are removed from the command string. This is shown in the following example:

```
DCL &CMD *CHAR 2000
  .
  .
CHGVAR &CMD '?CRTCLPGM'
CALL QCMDCHK (&CMD 2000)
```

After the call to the QCMDCHK program is run, variable &CMD contains the command string with all values entered through the prompter. This might be something like:

```
CRTCLPGM PGM(PGMA) SRCFILE(TESTLIB/SOURCE) USRPRF(*OWNER)
```

Note that the question mark preceding the command name is removed.

When prompting is requested through the QCMDCHK program, the command string should be passed in a CL variable. Otherwise, the updated command string is not returned to your 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 to exit from the prompter while running QCMDCHK, message CPF6801 is sent to the program that called QCMDCHK, and the variable containing the command string is not changed.

If PASSATR(*YES) is specified on the PARM, ELEM, or QUAL command definition statement, and the default value is changed using the CHGCMDDFT command, the default value will be highlighted as though this was a *user-specified value* and not a default value. If a default value of a changed PARM, ELEM, or QUAL command definition statement is changed back to its original default value, the default value will no longer be highlighted.

Using the QCLSCAN Program

You can use the QCLSCAN program to scan a string of characters to see if the string contains a pattern. This function is similar to the scan function supported within SEU and on the display presented by the DSPSPLF command. In addition, the QCLSCAN program also allows you to specify a 1-byte character in the pattern that matches with any character in the string to be searched, and a start position, which allows you to search the same string more than once.

A typical use of the QCLSCAN program is to allow the work station user to retrieve all records that contain a pattern that he specifies. For example, if the database has records with book titles, the work station user may want to retrieve all those books with the pattern CHICAGO in the title. The work station user enters CHICAGO on the device display. The application program reads the database, calling the QCLSCAN program at least once for each record to test for the pattern. The application program only processes the records that pass the test for the pattern CHICAGO.

Scanning a field can require many lines of code in a high level language such as RPG/400 and can cause a significant amount of overhead. Calling the QCLSCAN program and passing it a parameter list is simpler, and provides you with a fast-performing IBM-supplied program to do the scan. In a CL program, use the following CALL command:

```
CALL  PGM(QCLSCAN)    PARM(&STRING  &STRLEN  &STRPOS  &PATTERN  +
                      &PATLEN  &TRANSLATE  &TRIM  &WILD  &RESULT)
```

where:

- &STRING is a character field from 1 through 999 characters long. &STRING contains the string to be scanned for the pattern.
- &STRLEN is a 3-digit packed decimal variable with no decimal positions. &STRLEN contains the length of the string to be scanned. If this length is greater than the actual length of the string, unexpected results can occur.
- &STRPOS is a 3-digit packed decimal variable with no decimal positions. &STRPOS specifies the position in the string where the scan is to start. This must be greater than zero and not greater than the string length. Normally this value is 1. If the same string has multiple sets of patterns, this allows the string to remain the same while only the start position is varied to find the additional patterns.
- &PATTERN is a character variable from 1 through 999 bytes long. &PATTERN contains the pattern being scanned for.
- &PATLEN is a 3-digit packed decimal variable with no decimal positions. &PATLEN is the length of the pattern. If this length is greater than the actual length of the pattern, unexpected results can occur. See also the TRIM parameter.

- &TRANSLATE is a 1-byte character variable. If this field contains '1', the program translates lowercase characters of the string to uppercase before the scan. Only the lowercase EBCDIC characters 'a' through 'z' are translated. This does not change the user's data. Note that if '1' is specified and the pattern contains lowercase characters, a match will never occur. If '1' is specified and the data to be searched contains other than EBCDIC characters 'a' through 'z', unexpected results can occur.
- &TRIM is a 1-byte character variable. If this variable contains '1', trailing blanks are trimmed off the end of the pattern before the scan is started. This allows a fixed length pattern field to be filled (left-adjusted) by a variable number of characters. See Example 2 later in this section.
- &WILD is a 1-byte character variable. The value of this variable is a character that you can specify in the pattern in positions that should not be tested when scanning for a match. When this character appears in the pattern, any character in the data will be considered a match. A value of blank indicates that all characters of the pattern take part in the scan. If the wild character is the first character in the pattern, an error will occur. See Example 3 later in this section.
- &RESULT is a 3-digit packed decimal variable with no decimal positions. This parameter is returned to the user program when the call completes.

  If the value returned is positive, the result is the position of the first character of the pattern in the string.

  If the value returned is zero, the pattern was not found.

  If the value returned is negative, one of the following errors occurred:

  -1 The pattern is longer than the string.

  -2 The pattern length is less than 1.

  -3 The first character of the pattern is a wild character.

  -4 The pattern is blank and TRIM was requested.

  -5 The starting position within the string is not valid.

Example 1

Assume a 20-character database field containing only uppercase characters and the pattern 'ABC' will be scanned for. The user program calls the QCLSCAN program for each database record read. The parameters would be as follows:

```
&STRING     = The 20-byte field to be scanned
&STRLEN     = 20
&STRPOS     = 1
&PATTERN    = 'ABC'
&PATLEN     = 3
&TRANSLATE  = '0'
&TRIM       = '0'
&WILD       = ' '
&RESULT     = A value returned to your program
```

The following describes some fields and the results of the scan:

	String	Result	Comments
Scan 1:	ABCDEFGHIJKLMNOPQRST	001	
Scan 2:	XXXXABCXXXXXXXXXXXXX	005	
Scan 3:	abcXXXXXXXXXXXXXXXXX	000	Translation not requested
Scan 4:	XXXABCXXXXXABCXXXXXX	004	First occurrence found; see note
Scan 5:	ABABABABBCACCBACBABA	000	Not found
Scan 6:	ABABABCABCABCABCABCA	005	

Note: In scan 4, the string has two places where the pattern could have been found. Since the STRPOS value is 1, the first value (position 004) was found. If the value of STRPOS had been 4, the result would still have been 004. If the STRPOS value had been in a range of 5 through 12, the result would have been 012.

Example 2

Assume a 25-character database field containing only uppercase characters and a user program that will prompt for the pattern to be scanned, which will not exceed 10 characters. The work station user is allowed to enter 1 through 10 characters to search with and trailing blanks will be trimmed from the pattern. The program would call the QCLSCAN program for each database record read. The program parameters would be as follows:

&STRING	=	The 25-byte field to be scanned
&STRLEN	=	25
&STRPOS	=	1
&PATTERN	=	Varies
&PATLEN	=	10
&TRANSLATE	=	'0'
&TRIM	=	'1'
&WILD	=	' '
&RESULT	=	A value returned to your program

The following describes some fields and the results of the scan:

	String	Pattern	Result	Comments
Scan 1:	ABCDEFGHIJKLMNOPQRSTUVWXY	'CDE '	003	
Scan 2:	ABCDEFGHIJKLMNOPQRSTUVWXY	'CDEFGH '	003	
Scan 3:	ABCDEFGHIJKLMNOPQRSTUVWXY	'CDEFGHIJKL'	003	
Scan 4:	XXXXABCXXXXXXXXXXXXXXXXXX	'ABCD '	000	Not found
Scan 5:	abcXXXXXXXXXXXXXXXXXXXXXX	'ABC '	000	Not translated
Scan 6:	ABCXXXXXABC EXXXXXXXXXXXX	'ABC E '	009	
Scan 7:	XXXABCXXXXXABCXXXXXXXXXXX	'ABC '	004	See note

Note: In scan 7, the string has two places where the pattern could be found. Since the STRPOS value is 1, only the first value (position 004) is found. If the value of STRPOS were 4, the result would still be 004. If the STRPOS value were in the range of 5 through 12, the result would be 012.

Example 3

Assume a 25-character database field containing either uppercase or lowercase characters. The user program prompts for the pattern to be scanned, which does not exceed 5 characters. The work station user can enter 1 through 5 characters to be found. The system trims trailing blanks from the pattern. If the user enters an asterisk (*) in the pattern, the asterisk is handled as a wild character. The program calls the QCLSCAN program for each database record read. The parameters are as follows:

&STRING	= The 25-byte field to be scanned
&STRLEN	= 25
&STRPOS	= 1
&PATTERN	Varies
&PATLEN	= 5
&TRANSLATE	='1' (See note 1)
&TRIM	='1'
&WILD	= '*'
&RESULT	= A value returned to your program

The following describes some fields and the results of the scan:

	String	Pattern	Result	Comments
Scan 1:	ABCDEFGHIJKLMNOPQRSTUVWXY	'CDE '	003	
Scan 2:	ABCDEFGHIJKLMNOPQRSTUVWXY	'C*E '	003	
Scan 3:	abcdefghijklmnopqrstuvwxy	'C***G'	003	
Scan 4:	abcdefghijklmnopqrstuvwxy	'ABCD '	001	
Scan 5:	abcXXXXXXXXXXXXXXXXXXXXXX	'C*E '	000	Not found
Scan 6:	XXXAbcXXXXXabcXXXXXXXXXXX	'ABC '	004	See Note 2
Scan 7:	ABCDEFGHIJKLMNOPQRSTUVWXY	'*BC '	-003	See Note 3
Scan 8:	ABCDEFGHIJKLMNOPQRSTUVWXY	' '	-004	See Note 4

Notes:

1. When line translation is specified (the TRANSLATE parameter is specified as '1'), the string is translated to uppercase characters before scanning occurs; the data in the string is not changed.
2. In scan 6, the string has two places where the pattern could have been found. Since the STRPOS value is 1, the first value (position 004) was found.
3. In scan 7, the wild character (*) is the first character in the trimmed pattern. Wild characters cannot be the first character in a pattern.
4. In scan 8, the trimmed pattern is blank.

Using the QDCXLATE Program to Translate Fields

You can call the QDCXLATE program to translate individual fields. You can specify the translation table used by QDCXLATE to translate, for example, ASCII characters to EBCDIC.

IBM supplies four translation tables:

- QASCII and QEBCDIC for translation between ASCII and EBCDIC characters
- QSYSTRNTBL and QCASE256 for translation of lowercase to uppercase characters

You can also create your own translation tables.

QDCXLATE can distinguish double-byte from single-byte characters when converting from EBCDIC to ASCII and from ASCII to EBCDIC. QDCXLATE translates fields byte-for-byte and returns the translated fields to your program. For example, in a CL program, you could specify the following:

```
CALL PGM(QDCXLATE) PARM(&BUFLEN &BUFFER &SBCSTBLN &SBCSTBLL)
```

where:

- &BUFLEN is a 5-digit packed decimal field with no decimal positions. &BUFLEN contains the length of the field to be translated. The &FLDLEN variable cannot exceed 32767 and cannot be coded as a numeric constant in a CL program because a 15-digit packed decimal field will result.
- &BUFFER is a character field containing the data to be translated. This buffer also contains the output data after conversion when the program is called with only three or four parameters.
- &SBCSTBLN is a 10-character field containing the name of the SBCS translation table to be used. (The table name must be left-adjusted.)
- &SBCSTBLL is a 10-character field containing the name of the library that contains the SBCS translation table. (The library name must be left-adjusted.) If this field is not specified, the library list is used to find the translation table.

When the field is translated, the input (untranslated) data is replaced with the translated data.

When QDCXLATE is called with three or four parameters, it converts using single-byte characters. If more than four parameters are specified, then all 10 parameters must be specified and the conversion is done using the DBCS convert routines. All 10 parameters should only be specified when doing DBCS conversions.

The remaining parameters for QDCXLATE are:

- &OUTBUFF is a character field containing the output buffer. Because of the insertion of shift-in and shift-out character pairs, it is possible that the translated data will be longer than the source data so it is possible to do the translation in place, as is done on a 3- or 4-parameter call. The translated data is placed in the area pointed to by this parameter.
- &MAXOUTLN is a 5-digit packed decimal field containing the maximum length of converted output. The maximum length should match the actual size of OUTBUFF. If the translated output is longer than MAXOUTLN characters, an exception is signalled.
- &OUTLEN is a 5-digit packed decimal field containing the actual length of the translated output in OUTBUFF.
- &DBCSTBLN is a 10-character field containing the DBCS language which is being translated. It must be one of the following:

  ***JPN** Japanese

  ***KOR** Korean

  CHS Simplified Chinese

  CHT Traditional Chinese

The value of this parameter determines which module is called to do the DBCS translation.

- &DBCSSISO is a one-character field that indicates whether shift-out and shift-in characters should be inserted during the conversion. It must be one of the following:

  Y Insert shift-out and shift-in characters

  N Do not insert shift-out and shift-in characters

- &TRNTYPE is a 10-character field containing the type of conversion being done. It must be one of the following:

  ***AE** Convert ASCII to EBCDIC

  ***EA** Convert EBCDIC to ASCII

  A 10-character call can only be used to translate ASCII data to EBCDIC, or EBCDIC to ASCII. The &TRNTYPE parameter determines which translation is performed. This applies only to the DBCS conversion. The user is responsible to specify the correct SBCS table name for the type of conversion being done.

Character Equivalent Translation between ASCII and EBCDIC

When data is read from a tape or diskette file recorded in ASCII character code, the system converts each record to EBCDIC before returning it to the program using the data.

EBCDIC to ASCII Translation

IBM supplies a translation table, QASCII in library QSYS, with OS/400. If you need to determine the ASCII character equivalent of any fields recorded in EBCDIC, you should use the program QDCXLATE and specify QASCII as the value for the TBL parameter.

ASCII to EBCDIC Translation

IBM supplies a translation table, QEBCDIC in library QSYS, with OS/400. If you need to determine the EBCDIC character equivalent of any fields recorded in ASCII, you should use the program QDCXLATE and specify QEBCDIC as the value for the TBL parameter.

Usually, translation between ASCII and EBCDIC for tape and diskette is handled automatically by the system. The QASCII table can be used when a tape or diskette is written with ASCII labels but contains some packed decimal fields. These translation tables cannot be used for ICF files. Translation between ASCII and EBCDIC for ICF is handled by the system using the special tables required for ICF.

Lowercase to Uppercase Translation

IBM supplies two translation tables for lowercase to uppercase translation:

- QSYSTRNTBL in library QSYS translates only unaccented alphabetics (a through z) from lowercase to uppercase.
- QCASE256 in library QUSRSYS translates extended alphabetics (such as ä, ç, and ñ) as well as unaccented alphabetics (a through z) from lowercase to uppercase in code page 256.

Using one of these tables would allow your program to sort uppercase and lowercase letters together, such as a with A, b with B, and so forth.

Using Message Subfiles in a CL Program

In CL programs, message subfiles are the only type of subfiles supported. To use subfile message support, run a SNDF or SNDRCVF command using the subfile message control record. In the DDS, supply SFLPGMQ data and specify the SFLINZ keyword.

When you use message subfiles in CL programs, you must name a program. You cannot specify an * for the SFLPGMQ keyword in DDS. When you specify a program name, all messages sent to that program's message queue are taken from the program message queue and placed in the message subfile. All messages associated with the current request are taken from the program message queue and placed in the message subfile.

Message subfiles let a controlling program display one or more error messages.

For more information about message subfiles, see the *Data Management Guide* and the *DDS Reference*.

Allowing User Changes to CL Commands at Run Time

With most CL programs, the work station user provides input to the program by specifying parameter values passed to the program or by typing into input-capable fields on a display prompt.

You can also prompt the work station user for input to a CL program in the following ways:

- If you enter a ? before the CL command in the CL program source, the system displays a prompt for the CL command. Parameter values you have already specified in your program are filled in and cannot be changed by the work station user. See "Using the Prompter within a CL Program" on page 6-13 later in this section.
- If you call the QCMDEXC program and request selective prompting, the system displays a prompt for a CL command, but you need not specify in the CL program source which CL command is to be used at processing time. For more information on the QCMDEXC program, see "Using the QCMDEXC Program" on page 6-1.

Using the Prompter within a CL Program

You can request prompting within the interactive processing of a CL program. For example, the following program can be compiled and run:

```
PGM
.
.
.
?DSPLIB
.
.
.
ENDPGM
```

In this case, the prompter for the Display Library (DSPLIB) command appears on the display during processing of the program. Processing of the DSPLIB command waits until you have entered values for required parameters and pressed the Enter key.

Any values specified in the source program cannot be changed directly by the prompter. For example:

```
PGM
.
.
.
?SNDMSG TOMSGQ(WS01 WS02)
.
.
.
ENDPGM
```

When the program is called and the prompt for the Send Message (SNDMSG) command appears, you can enter values on the MSG, MSGTYPE, and RPYMSGQ parameters, but you cannot alter the values on the TOMSGQ parameter. You cannot add WS03 or delete WS02, for example. See "QCMDEXC with Prompting in CL Programs" on page 6-17 for an exception to this restriction. The following restrictions apply to the use of the prompter within a CL program at processing time:

- When the prompter is called from a CL 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)`
`?SNDMSG`
`ENDDO`
	`IF (&A=5) THEN(?SNDMSG)`

- Prompting *cannot* be used for the following commands:

```
CALL      ENDDO     PGM
CHGVAR    ENDPGM    RCVF
DCL       ENDRCV    RETURN
DCLF      IF        SNDF
DO        GOTO      SNDRCVF
ELSE      MONMSG    TFRCTL
                    WAIT
```

- Prompting cannot be used in batch jobs.

When you enter a prompting request (?) on a command in a CL source program, 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 prompter display when the 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 program. This allows you to prompt for any command while at a work station 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 program. You can monitor for this message using the MONMSG command in the CL program.

Selective Prompting for CL Commands

You can request to prompt for selected parameters within a command. This is especially helpful when you are using some of the longer commands and do not want to be prompted for certain parameters.

Selective prompting can be used during interactive prompting or entered as source (in SEU) for use within a CL program. You can enter the source for selective prompting with SEU but you cannot use selective prompting while entering commands in SEU. Refer to the *SEU User's Guide/Reference*.

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).
- Parameters can be specified by position but they cannot be preceded by selective prompt characters.
- A parameter must be in keyword form to be selectively prompted for.
- Blanks cannot be entered between the selective prompt characters and the keyword.
- Selective prompting is only applicable at a parameter level; that is, you cannot specify particular keyword values within a list of values.

- ?- is not allowed in prompt override programs.
- If a parameter is required, the ?? selective prompt must be used.

  You can tell that a parameter is required because the input slot is highlighted when the command is prompted.

User-specified values are marked with a special symbol (>) in front of the values in both selective and regular prompting. If a user-specified value on the parameter prompt is not preceded by this symbol, the command default is passed to the command processing program.

If PASSATR(*YES) is specified on the PARM, ELEM, or QUAL command definition statement, and the default value is changed using the CHGCMDDFT command, the default value is shown as a user-specified value (using the > symbol) and not a default value. If a default value of a changed PARM, ELEM, or QUAL command definition statement is changed back to its original default value, the > symbol is removed.

You can press F5 while you are using selective prompting to again display those values initially shown on the display.

If a CL variable is used to specify a value for a parameter which is to be displayed through selective prompting, you can change the value on the prompt, and the changed value is used when the command is run. The value of the variable in the program is not changed. If a CL program contains the following:

```
OVRDBF ?*FILE(FILEA) ??TOFILE(&FILENAME) ??MBR(MBR1)
```

the three parameters, FILE, TOFILE, and MBR will be 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 program. The following table lists 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

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	Reserved for IBM use.
?<KEYWORD(VALUE)	Value	Reserved for IBM use.
?/KEYWORD()	Default	Reserved for IBM use.
?/KEYWORD(VALUE)	Value	Reserved for IBM use.

Following is a brief description of the selective prompting characters:

Selective Prompting Character	Description
??	The parameter is displayed and input-capable.
?*	The parameter is displayed but is not input-capable. Any user-specified value will be passed to the command processing program.
?<	For IBM-supplied commands. 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.
?/	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.

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)
```

For a further discussion of QCMDEXC or QCMDCHK refer to "Using the QCMDEXC Program" on page 6-1 and "Using the QCMDCHK Program" on page 6-4.

QCMDEXC with Prompting in CL Programs

The QCMDEXC program may be used to call the prompter. This QCMDEXC with Prompting in CL 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 program (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 CL program using the QCMDEXC program, such as:

```
CALL QCMDEXC PARM('?OVRDBF FILE(FILEX)' 19)
```

you can specify a value for any parameter, including FILE. In this example, FILEX is the default.

Prompting with modifiable specified values may also be accomplished using selective prompting as described earlier in this chapter. However, each keyword which is desired must be explicitly selected. The prompter will be called directly with a command such as:

```
OVRDBF ??FILE(FILEX) ??TOFILE(*N) ??MBR(*N)
```

Using the Programmer Menu

The programmer menu can be called directly by calling the QPGMMENU program, or by using the Start Programmer Menu (STRPGMMNU) command. You can use the command to specify in advance the defaults that you will be using with the programmer menu. In addition, the STRPGMMNU command also supports other options that can be used to tailor the use of the programmer menu.

For a description of the STRPGMMNU command and its parameters, see the *CL Reference*.

Uses of the Start Programmer Menu (STRPGMMNU) Command

The Start Programmer Menu command can be used for the following:

- Performing the same function as a call to QPGMMENU
- Filling in the standard input fields

  Four of the command parameters allow you to fill in the standard input fields at the bottom of the menu. These parameters are the following:
  - Source file
  - Source library
  - Object library
  - Job description

The command may be used with one or more of the parameters that control the initial values of the menu. You could design this as part of an initial program for sign-on or for situations in which a user calls a specific user-written function. The following example shows such a program, with a separate function for each application area requiring different initial values.

```
     PGM
     CHGLIBL   LIBL(PGMR1 QGPL QTEMP)
LOOP:
     STRPGMMNU SRCLIB(PGMR1) OBJLIB(PGMR1) JOBD(PGMR1)
     MONMSG    MSGID(CPF2320) EXEC(GOTO END) /* F3 or F12 to leave menu */
     GOTO LOOP
END: ENDPGM
```

- Controlling programmer menu options

  The other parameters assist you in controlling the menu and its functions. For example, you can specify ALWUSRCHG(*NO) to prevent a user from changing the values that appear on the menu. This parameter should not be considered to be a security feature because a user who is using the menu can call the STRPGMMNU command and change the values in a separate call. (The user can also start functions by using F10 to call the command entry display.) If the menu is displayed by the STRPGMMNU command, you can prevent the user (by authorization) from calling the QPGMMENU program directly, but you cannot prevent the user from requesting another call of the STRPGMMNU command.

- Adapting the menu create option

  The EXITPGM and DLTOPT parameters allow you to provide your own support for the menu create option (option 3). A user program may be called when option 3 is requested. For a full discussion of the parameters and the parameter list passed to the user program, see the *CL Reference*. The following describes some typical uses of the EXITPGM parameter.

The EXITPGM Parameter

The EXITPGM parameter can be used for the following purposes:

- To change the defaults used on the create commands submitted by option 3.

  For example, if F4 (Prompt) is not used, the EXITPGM parameter could change one or more of the create commands to specify your own default requirements. This approach is used in the sample program "Basic Use of a STRPGMMNU Exit Program" on page 6-19. If F4 is used, the EXITPGM parameter could submit the command as entered by the programmer (with no parameters changed).

- To change parameters regardless of the programmer's use of F4.

  This requires scanning the value of the &RQSDTA512 parameter (which is passed to the exit program) to see if it had already been used and substituting the required value.

- To change other parameters on the SBMJOB command.

  For example, the USER parameter of the SBMJOB command could be changed to specify the value from the job description instead of the current user. It would also be possible to extract some of the current job attributes by using the RTVJOBA command and enter them as specific values (for example, INLLIBL or OUTQ).

- To enforce local programming conventions.

  For example, if you have a naming standard that requires all physical files to be named with 7 characters and end with a P, the exit program could reject any attempt to use the CRTPF command with a name that did not follow this standard.

- To assist in replacing existing objects.

  Replacing an object depends on certain considerations, such as:

  — Is anyone currently using the existing object?

  — Should the existing object be deleted if the create command fails?

  — Are there attributes of the existing object which should be included in the new object (for example, a file option of SHARE(*YES) or a program option of USRPRF(*OWNER))?

  — Should authorizations be copied from the existing object?

  — If a physical file is being replaced that contains data, should the existing data be copied after the new version is created? How should it be formatted?

  — If a physical file is being replaced, does it have logical files built over it? The logical files must be deleted before the physical file is deleted. Should the logical files be automatically rebuilt over the new physical file?

    While it is not possible to automate all functions that may be needed, it is possible to create a program that will perform many of the basic requirements. Because several of the functions required might be time-consuming, the main functions would normally be run in batch, freeing the programmer for interactive functions.

    The following commands support a REPLACE option that allows users to continue using program objects that have been replaced:

    - All create program commands (CRTxxxPGM)
    - Create Display File (CRTDSPF) command
    - Create ICF File (CRTICFF) command
    - Create Print File (CRTPRTF) command

  For an illustration of an exit program used to submit a CL program that replaces objects, see “Advanced Use of a STRPGMMNU Exit Program” on page 6-21.

Basic Use of a STRPGMMNU Exit Program

The following sample program uses the STRPGMMNU command so that the exit program SBMCRTC will submit a job that creates an object based on the parameters passed to the exit program.

The program shows how to add a parameter to the CRTDSPF command if F4 (Prompt) is not used. The program checks for the use of a CRTDSPF command and F4. If F4 is not used, the RSTDSP(*YES) parameter is added to the command; this effectively changes the default of the CRTDSPF command.

If the command exceeds 256 bytes, it is rejected since the SBMJOB command allows a maximum of 256 bytes of request data (see a later example for how larger commands could be handled). The command is then submitted to batch. Any message produced by the SBMJOB command is received and sent again so that it will appear on the programmer's menu.

The following is the command to call the menu (other parameters to initialize values could also be included).

```
STRPGMMNU       .... EXITPGM(QGPL/SBMCRTC)
```

Note that the default for the DLTOPT parameter was taken. This means the normal menu function of F11 is in effect if the object already exists. That is, the user must acknowledge the existence of the object and that the use of F11 deletes the object; or, if the object is an AS/400 environment *program, display file, ICF file,* or *print file*, the user must replace the object. The source for the CL program SBMCRTC follows:

```
SEQNBR *... ... 1 ... ... 2 ... ... 3 ... ... 4 ... ... 5 ... ... 6 ... ... 7 ..

             PGM        PARM(&OPTION &PARM &TYPE &PARM2 +
                          &SRCFILE &SRCLIB &OBJLIB &JOBD &RQSLEN +
                          &RQSDTA512 &F4 &F11 &EXIST)
             DCL        &OPTION *CHAR LEN(2)
             DCL        &PARM *CHAR LEN(10)
             DCL        &TYPE *CHAR LEN(10)
             DCL        &PARM2 *CHAR LEN(21)
             DCL        &SRCFILE *CHAR LEN(10)
             DCL        &SRCLIB *CHAR LEN(10)
             DCL        &OBJLIB *CHAR LEN(10)
             DCL        &JOBD *CHAR LEN(10)
             DCL        &RQSLEN *DEC LEN(3 0)
             DCL        &RQSDTA512 *CHAR LEN(512)
             DCL        &F4 *CHAR LEN(1)
             DCL        &F11 *CHAR LEN(1)
             DCL        &EXIST *CHAR LEN(1)
             DCL        &RQSDTA256 *CHAR LEN(256)
             DCL        &MSGID *CHAR LEN(7)
             DCL        &MSGDTA *CHAR LEN(100)
                        /* The following shows how to add  */
                        /* parameters to a command such as */
                        /* CRTDSPF if F4 (Prompt) has not  */
                        /* been requested.                 */
             IF         ((&TYPE *EQ 'DSPF') *AND (&F4 *EQ '0')) DO
             CHGVAR     &RQSLEN (&RQSLEN + 2) /* Next parm location */
             CHGVAR     %SST(&RQSDTA512 &RQSLEN 12) 'RSTDSP(*YES)'
             CHGVAR     &RQSLEN (&RQSLEN + 12)     /* Add for parm */
             ENDDO      /* DSPF and not F4                         */
                        /* SBMJOB allows only 256 bytes of RQSDTA  */
                        /* Check if the RQSLEN exceeds 256.        */
             IF         (&RQSLEN *GT 256) SNDPGMMSG MSGID(CPF9898) +
                           MSGF(QCPFMSG) MSGDTA('Command cannot be +
                           submitted--exceeds 256 bytes') +
                           MSGTYPE(*ESCAPE)
             CHGVAR     &RQSDTA256 &RQSDTA512  /* Truncate +
                        and submit the job with the RQSDTA.        */
             SBMJOB     JOB(&PARM) JOBD(&JOBD) RQSDTA(&RQSDTA256)
                        /* Receive any messages from SBMJOB and resend. */

             RCVMSG     MSGID(&MSGID) MSGDTA(&MSGDTA)
             SNDPGMMSG  MSGID(&MSGID) MSGF(QCPFMSG) MSGDTA(&MSGDTA)
             ENDPGM
```

Advanced Use of a STRPGMMNU Exit Program

The following sample program uses the STRPGMMNU command exit program to assist in replacing existing objects. It also establishes a database file to contain a record for each job submitted; this will assist in performing audit trail analysis.

This sample program makes the following assumptions:

- All objects are created in the library QTEMP. If the object is created and an old version exists, authorizations are copied from this existing old version to the new version, the old version is deleted, and the new version is moved to the specified library (see the special handling of physical files). If no old version exists, the object is moved to the specified library.
- Physical files are created in the library QTEMP. If there is no old version, the new version is moved to the specified library. If an old version exists, authorizations are copied to the new version and the new version is moved to the library TEMPCRT. The old version is not deleted. Since the old version may contain data in one or more members and may have dependent logical files built over it, it is left to the user to do the actual copying, reformatting, and deletion of data.
- If F4 is not used and the object is a file, the value of the SHARE parameter is retrieved from the old object (if it exists) and made part of the create command.

The SBMJOB command has a limit of 256 bytes for the RQSDTA parameter. To allow for the submission of create commands with large request data fields and to pass additional parameters, a database file is used to contain the requests to be submitted. The key field for the database file is the qualified job name of the job which is submitted. After the SBMJOB command is issued, the job name is retrieved and then used as the key field to add a record to the database file. This database file must be periodically cleared of all records. The file may also be used to determine how many create requests are submitted by each programmer, how many times each object is recreated, and so forth. The file is updated when the create function is started and ended to time stamp the events and provide a completion code. To use this example, perform the following steps:

1. Create the database file MNUCRTP for use in the following programs. This file contains the command to be run to create the object as well as other data from the programmer menu and date/time/completion information. Programs can be written to analyze or display this file. The source for the physical file MNUCRTP follows:

```
SEQNBR *... ... 1 ... ... 2 ... ... 3 ... ... 4 ... ... 5 ... ... 6 ... ... 7 ..

          A          R MNUCRTR
          A            JOB           10          COLHDG('Job name')
          A            USER          10          COLHDG('User name')
          A            JOBNBR         6          COLHDG('Job' 'number')
          A            MEMBER        10          COLHDG('Member name')
          A            SRCTYP         4          COLHDG('Source' 'type')
          A            SRCFIL        10          COLHDG('Source' 'file')
          A            SRCLIB        10          COLHDG('Source' 'library')
          A            OBJLIB        10          COLHDG('Object' 'library')
          A            JOBD          10          COLHDG('Job' 'description')
          A            F4             1          COLHDG('F4' 'status')
          A            CMD          512          COLHDG('Command')
          A            RQSLEN         3  0       COLHDG('Request' 'length')
          A* CRTNAM/CRTLIB may differ from MEMBER/OBJLIB if the F4
          A*   prompter function is used.
          A            CRTNAM        10          COLHDG('Create' 'name')
          A            CRTLIB        10          COLHDG('Create' 'library')
          A            OBJTYP         5          COLHDG('Object' 'type')
          A            SBMTIM         6  0       COLHDG('Submit' 'time')
          A                                      EDTWRD('0 :  :  ')
          A            SBMDAT         6  0       COLHDG('Submit' 'date')
          A                                      EDTCDE(Y)
          A            STRTIM         6  0       COLHDG('Start' 'time')
          A                                      EDTWRD('0 :  :  ')
          A            STRDAT         6  0       COLHDG('Start' 'date')
          A                                      EDTCDE(Y)
          A            CMPCDE         1          COLHDG('Cmp' 'code')
          A                                      TEXT('Completion code +
          A                                             Y = yes, N=no')
          A            CMPTIM         6  0       COLHDG('Completion' 'time')
          A                                      EDTWRD('0 :  :  ')
          A            CMPDAT         6  0       COLHDG('Completion' 'date')
          A                                      EDTCDE(Y)
          A          K JOB
          A          K USER
          A          K JOBNBR
```

2. Create the library TEMPCRT for physical file create operations when there is an existing object. The following is a sample command:

```
CRTLIB    TEMPCRT TEXT('Library used by the Programmer +
          Menu Exit program')
```

3. Create the message queue MNUMSGQ in library TEMPCRT. Messages sent to this message queue are used to ensure that database writes by MNUSBMC are completed before database accesses of those same records by MNUCRTC.

4. Create the CL program MNUSBMC to be used as the exit program for the STRPGMMNU command. The source for CL program MNUSBMC follows:

```
SEQNBR *... ... 1 ... ... 2 ... ... 3 ... ... 4 ... ... 5 ... ... 6 ... ... 7 ..

            PGM       PARM(&OPTION &PARM &TYPE &PARM2 +
                         &SRCFILE &SRCLIB &OBJLIB &JOBD &RQSLEN +
                         &RQSDTA512 &F4 &F11 &EXIST)
            DCL       &OPTION *CHAR LEN(2)
            DCL       &PARM *CHAR LEN(10)
            DCL       &TYPE *CHAR LEN(10)
            DCL       &PARM2 *CHAR LEN(21)
            DCL       &SRCFILE *CHAR LEN(10)
            DCL       &SRCLIB *CHAR LEN(10)
            DCL       &OBJLIB *CHAR LEN(10)
            DCL       &OBJTYP *CHAR LEN(5)
            DCL       &JOBD *CHAR LEN(10)
            DCL       &RQSLEN *DEC LEN(3 0)
            DCL       &RQSDTA512 *CHAR LEN(512)
            DCL       &CRTNAM *CHAR LEN(10)
            DCL       &CRTLIB *CHAR LEN(10)
            DCL       &F4 *CHAR LEN(1)
            DCL       &F11 *CHAR LEN(1)
            DCL       &EXIST *CHAR LEN(1)
            DCL       &MSGID *CHAR LEN(7)
            DCL       &MSGDTA *CHAR LEN(100)
            DCL       &RPGCDE  *CHAR LEN(1)
            DCL       &CMPCDE *CHAR LEN(1) /* Required for parm list */
            DCL       &RETURN *CHAR LEN(8) /* Required for parm list */
            DCL       &JOB *CHAR LEN(10)
            DCL       &USER *CHAR LEN(10)
            DCL       &JOBNBR *CHAR LEN(6)
            DCL       &X *DEC LEN(3 0) /* Index into RQSDTA512 */
            DCL       &Y *DEC LEN(3 0) /* First blank after library */
            DCL       &Z *DEC LEN(3 0) /* Position to shift to */
            DCL       &LFFTPOS *DEC LEN(3 0) /* Left position of name */
            DCL       &NAMLTH *DEC LEN(3 0) /* Length of name */
            DCL       &SHIFTCHAR *DEC LEN(3 0) /* Characters to shift */
            DCL       &SAVE512 *CHAR LEN(512) /* Work area */
            DCL       &BLANKCHAR *DEC LEN(3 0) /* Chars to blank out */
            DCL       &BLANK25 *CHAR LEN(25) /* 25 blanks */
                      /* Limit cmd to 500 to allow create in QTEMP */
            IF        (&RQSLEN *GT 500) SNDPGMMSG MSGID(CPF9898) +
                        MSGF(QCPFMSG) MSGTYPE(*ESCAPE) +
                        MSGDTA('Command cannot exceed 500 bytes')
                      /* The object/library names may differ if the  */
                      /*  prompter was used.  Extract actual names   */
            CHGVAR    &X 1 /* Initialize */
 LFTPAREN:  IF        (%SST(&RQSDTA512 &X 1) *NE '(') DO
            CHGVAR    &X (&X + 1) /* Increment */
            GOTO      LFTPAREN /* Loop back and try again */
            ENDDO     /* End of left Paren not found */
            CHGVAR    &LFFTPOS (&X + 1)  /* Start of library name */
 SLASH:     IF        (%SST(&RQSDTA512 &X 1) *NE '/') DO
            CHGVAR    &X (&X + 1) /* Increment */
            GOTO      SLASH    /* Loop back and try again */
            ENDDO     /* End of slash not found */
```

```
SEQNBR *... ... 1 ... ... 2 ... ... 3 ... ... 4 ... ... 5 ... ... 6 ... ... 7 ..
                  CHGVAR     &NAMLTH (&X - &LFFTPOS) /* Length of lib name */
                  CHGVAR     &CRTLIB %SST(&RQSDTA512 &LFFTPOS &NAMLTH)
                             /* Shift remaining characters to the right to */
                             /*  ensure room for QTEMP library and change  */
                             /*  the name of the object library to QTEMP   */
                  CHGVAR     &SHIFTCHAR (512 -&X +1) /* Nbr chars to move */
                  CHGVAR     &SAVE512 %SST(&RQSDTA512 &X &SHIFTCHAR) /* Sav */
                  CHGVAR     %SST(&RQSDTA512 &LFFTPOS 5) 'QTEMP' /* Chg lib */
                  CHGVAR     &Y (&LFFTPOS + 5) /* First position past QTEMP */
                  CHGVAR     %SST(&RQSDTA512 &Y &SHIFTCHAR) &SAVE512 /* Rst */
                  CHGVAR     &RQSLEN  (&RQSLEN + 5 - &NAMLTH) /* Adj dta len*/
                  CHGVAR     &LFFTPOS (&Y + 1)  /* Start ofobject name */
                  CHGVAR     &X &Y              /* Start of object name */
        RGTPAREN: IF         (%SST(&RQSDTA512 &X 1) *NE ')') DO
                  CHGVAR     &X (&X + 1) /* Increment */
                  GOTO       RGTPAREN    /* Loop back and try again */
                  ENDDO      /* End of right paren not found */
                  CHGVAR     &NAMLTH (&X - &LFFTPOS) /* Length of obj name */
                  CHGVAR     &CRTNAM %SST(&RQSDTA512 &LFFTPOS &NAMLTH)
                             /* Submit job and resend completion message */
                  SBMJOB     JOB(&PARM) JOBD(&JOBD) RQSDTA('CALL MNUCRTC')
                  RCVMSG     MSGID(&MSGID) MSGDTA(&MSGDTA)
                  SNDPGMMSG  MSGID(&MSGID) MSGF(QCPFMSG) MSGDTA(&MSGDTA)
                             /* Retrieve values from message data */
                  CHGVAR     &JOB %SST(&MSGDTA 1 10)    /* Extract JOB */
                  CHGVAR     &USER %SST(&MSGDTA 11 10) /* Extract USER */
                  CHGVAR     &JOBNBR %SST(&MSGDTA 21 6) /* Extract JOB NBR */
                             /* Write new record to MNUCRTP */
                  CHGVAR     &RPGCDE 'W' /* Write new record to MNUCRTP */
                  CALL       MNUCRT PARM(&RPGCDE &JOB &USER &JOBNBR &PARM +
                                &TYPE &SRCFILE &SRCLIB &OBJLIB &JOBD +
                                &F4 &RQSDTA512 &RQSLEN &CRTNAM &CRTLIB +
                                &OBJTYP &CMPCDE &RETURN)
                  SNDPGMMSG  MSG('DB write complete') +
                                TOMSGQ(TEMPCRT/MNUMSGQ)
                  ENDPGM
```

The program accepts the parameter list from the programmer's menu. To create the object in QTEMP, the command passed from the menu must be changed. Since the actual library name might be shorter than 5 characters (the length of QTEMP), part of the command is shifted to the right, the actual library name blanked out and the name QTEMP inserted. Because of this shifting, the request length is checked to ensure it does not exceed 500 bytes.

The program scans from the left of the passed command to find the first left parenthesis (the beginning of the library containing the object being created). It then scans for the first slash (separator character) and extracts the name of the library. It then scans for the first right parenthesis (the end of the object name) and extracts the actual object name to be used. Note that the PARM and OBJLIB fields passed from the menu may not represent the actual object if the user requested F4 (Prompt) and changed the command. The actual names are saved for later use.

To ensure adequate room for the library name QTEMP, the remainder of the command (after the qualified name) is shifted to the right. Blanks are inserted in the area that was shifted, and the literal 'QTEMP' is added before the slash (/) in the qualified name. This changes the command so that it does the create in the QTEMP library.

A job is then submitted that calls MNUCRTC (the standard program used for all create operations). After a successful SBMJOB command is run, the message CPC1221 is received which contains the qualified job name of the batch job. The JOB, USER, and JOBNBR information is extracted from the message data. A single RPG/400 program (MNUCRT) is used for all functions involving the file MNUCRTP. A code is set to cause a new record to be added and the program is called with a parameter list of all the information available. These programs are shown on the surrounding pages.

5. Create the CL program MNUCRTC, which is submitted to batch by the MNUSBMC program to perform the actual steps of object creation. The source for the CL program MNUCRTC follows:

```
SEQNBR *... ... 1 ... ... 2 ... ... 3 ... ... 4 ... ... 5 ... ... 6 ... ... 7 ..

            PGM
                       /* The QAFDBASI file is used as an example      */
                       /*  of accessing an existing object for the     */
                       /*  current value of a parameter.               */
            DCLF       QSYS/QAFDBASI
            DCL        &RPGCDE *CHAR LEN(1)
            DCL        &JOB *CHAR LEN(10)
            DCL        &USER *CHAR LEN(10)
            DCL        &JOBNBR *CHAR LEN(6)
            DCL        &MEMBER *CHAR LEN(10)
            DCL        &SRCTYP *CHAR LEN(4)
            DCL        &SRCFIL *CHAR LEN(10)
            DCL        &SRCLIB *CHAR LEN(10)
            DCL        &OBJLIB *CHAR LEN(10)
            DCL        &OBJTYP *CHAR LEN(5)
            DCL        &JOBD *CHAR LEN(10)
            DCL        &F4 *CHAR LEN(1)
            DCL        &CMD *CHAR LEN(512)
            DCL        &RQSLEN *DEC LEN(3 0)
            DCL        &CRTNAM *CHAR LEN(10)
            DCL        &CRTLIB *CHAR LEN(10)
            DCL        &CMPCDE *CHAR LEN(1)
            DCL        &RETURN *CHAR LEN(8)
            DCL        &CMD2 *CHAR LEN(768) /* Allows added parms */
            DCL        &X *DEC LEN(3 0) VALUE(1)
            DCL        &MSG *CHAR LEN(132)
                       /* WAIT FOR NEW DB RECORD TO BE WRITTEN */
GETMSG:     RCVMSG     MSGQ(TEMPCRT/MNUMSGQ) WAIT(*MAX) RMV(*YES) +
                         MSG(&MSG)
            IF         (&MSG *NE 'DB write complete') GOTO GETMSG
                       /* Retrieve command from MNUCRTP file */
            RTVJOBA    USER(&USER) JOB(&JOB) NBR(&JOBNBR)
            CHGVAR     &RPGCDE 'S' /* Read the job rcd and update str */
            CALL       MNUCRT PARM(&RPGCDE &JOB &USER &JOBNBR +
                         &MEMBER &SRCTYP &SRCFIL &SRCLIB &OBJLIB +
                         &JOBD &F4 &CMD &RQSLEN &CRTNAM &CRTLIB +
                         &OBJTYP &CMPCDE &RETURN)
 NOTFOUND:  IF         (&RETURN *EQ 'NOTFOUND') SNDPGMMSG +
                         MSGID(CPF9898) MSGF(QCPFMSG) MSGTYPE(*ESCAPE) +
                         MSGDTA('Job record was not found in the +
                         MNUCRTP file for ' *CAT &JOBNBR *TCAT +
                         '/' *CAT &USER *TCAT '/' *CAT &JOB)
            CHGVAR     &CMD2 &CMD /* Allows added parameters          */
                       /*Example of accessing an existing object      */
                       /*  by using the DSPFD OUTFILE function        */
                       /*  to retrieve the Share parameter value      */
                       /*  if F4 has not been used.                   */
            IF         ((&OBJTYP *EQ '*FILE') *AND (&F4 *EQ '0'))  DO
            CHKOBJ     OBJ(&CRTLIB/&CRTNAM) OBJTYPE(*FILE)
            MONMSG     MSGID(CPF9801) EXEC(GOTO ENDDSPFD) /* Not found */
            DSPFD      FILE(&CRTLIB/&CRTNAM) TYPE(*BASATR) +
                         OUTPUT(*OUTFILE) OUTFILE(QTEMP/DSPFDOUT)
            OVRDBF     QAFDBASI TOFILE(QTEMP/DSPFDOUT) /* OVR to QTEMP */
            RCVF       /* SHARE parm is &ATSHAR-file QSYS/QAFDBASI */
            IF         (&ATSHAR *EQ 'Y') DO
            CHGVAR     &RQSLEN (&RQSLEN + 2)
            CHGVAR     %SST(&CMD2 &RQSLEN 11) 'SHARE (*YES)'
            CHGVAR     &RQSLEN (&RQSLEN +11)
            ENDDO      /* End of SHARE = yes */
 ENDDSPFD:  ENDDO      /* End of OBJTYP = *FILE and F4 = 0 */
```

```
SEQNBR *... ... 1 ... ... 2 ... ... 3 ... ... 4 ... ... 5 ... ... 6 ... ... 7 ..

 LOGCMD:    IF         (%SST(&CMD2 &X 128) *NE ' ') DO
            SNDPGMMSG  MSG(%SST(&CMD2 &X 128))
            CHGVAR     &X (&X + 128)
            IF         (&X *NE 769) GOTO LOGCMD
            ENDDO      /* Non blank */
                       /* Run command */
            CALL       QCMDEXC PARM(&CMD2 768)
                       /* Monitor for failure on create */
            MONMSG     MSGID(CPF0000) EXEC(GOTO FAIL) /* Any CPF */
            MONMSG     MSGID(IDU0000) EXEC(GOTO FAIL) /* Any IDU */
            MONMSG     MSGID(QRC0000) EXEC(GOTO FAIL) /* Any RPG */
            MONMSG     MSGID(CBL0000) EXEC(GOTO FAIL) /* Any CBL */
            MONMSG     MSGID(BAS0000) EXEC(GOTO FAIL) /* Any BAS */
            CHGVAR     &CMPCDE 'Y' /* Set good completion code */
                       /* If old object exists, transfer authority,  */
                       /*    delete old object and move to library   */
            CHKOBJ     OBJ(&CRTLIB/&CRTNAM) OBJTYPE(&OBJTYP)
            MONMSG     MSGID(CPF9801) EXEC(GOTO MOVE) /* Not found */
              GRTOBJAUT  OBJ(QTEMP/&CRTNAM) OBJTYPE(&OBJTYP) +
                           REFOBJ(&CRTLIB/&CRTNAM) REFOBJTYPE(&OBJTYP)
            IF         (&OBJTYP *EQ '*PGM') DLTPGM &CRTLIB/&CRTNAM
            IF         (&OBJTYP *EQ '*CMD') DLTCMD &CRTLIB/&CRTNAM
            IF         (&OBJTYP *EQ '*FILE') DO
                       /* If a physical file is being replaced, move   */
                       /*   the new to TEMPCRT and do not delete the   */
                       /*   the old.  Error if it exists in TEMPCRT.   */
            IF         (&SRCTYP *EQ 'PF') DO
            CHKOBJ     OBJ(TEMPCRT/&CRTNAM) OBJTYPE(&OBJTYP)
            MONMSG     MSGID(CPF9801) EXEC(GOTO MOVPF) /* Not found */
            SNDPGMMSG  MSG('Physical file ' *CAT &CRTNAM *TCAT +
                         ' already exists in TEMPCRT - job will be +
                         ended abnormally')
            GOTO       FAIL
 MOVPF:     MOVOBJ     QTEMP/&CRTNAM OBJTYPE(&OBJTYP) TOLIB(TEMPCRT)
                       GOTO UPDATE /* Bypass delete and move */
            ENDDO      /* End of SRCTYP = PF */
                       DLTF &CRTLIB/&CRTNAM /* All other file types */
                       ENDDO  /* End of OBJTYP = FILE */
 MOVE:      MOVOBJ     QTEMP/&CRTNAM OBJTYPE(&OBJTYP) TOLIB(&CRTLIB)
 UPDATE:    CHGVAR     &RPGCDE 'E' /* End of create - update job rcd   */
            CALL       MNUCRT PARM(&RPGCDE &JOB &USER &JOBNBR +
                          &MEMBER &SRCTYP &SRCFIL &SRCLIB &OBJLIB +
                          &JOBD &F4 &CMD &RQSLEN &CRTNAM &CRTLIB +
                          &OBJTYP &CMPCDE &RETURN)
            IF         (&RETURN *EQ 'NOTFOUND') GOTO NOTFOUND
            IF         (&CMPCDE *EQ 'N') SNDPGMMSG MSGID(CPF9898) +
                          MSGF(QCPFMSG) MSGTYPE(*ESCAPE) +
                          MSGDTA('Create command failed')
            RETURN
 FAIL:      CHGVAR     &CMPCDE 'N' /* Bad completion code */
            GOTO       UPDATE
            ENDPGM
```

This program runs in batch mode. The program retrieves the job attributes that identify the qualified job name. It sets the start code for the RPG/400 program and calls the program. The RPG/400 program retrieves the record which was written during the interactive function and returns it to the CL program.

As an example of how an existing object can be used to help supply the parameters for the create command, the program accesses an existing file to retrieve the value of the SHARE parameter. This is only done if F4 is not used. The program checks for the existence of the file object and, if it exists, the DSPFD OUTFILE parameter is used to build a file in QTEMP of the existing file's attributes. The DSPFD command (for TYPE(*BASATR)) uses the file QSYS/QAFDBASI for the standard format. This file is declared in the program (DCLF command). The output file is written to QTEMP. An OVRDBF command is used prior to reading the file. The SHARE parameter value is in the &ATSHAR field of this format. The field is tested and if it contains a Y, the literal SHARE(*YES) is added at the end of the command. Note that the command to be run has been moved to a 768-byte field to allow for additional parameters.

If a program object were being replaced, you might want to use the value of the USRPRF parameter. See "Retrieving Program Attributes" on page 6-37.

To assist in debugging, the command to be run is sent as a series of messages to the job log. The create command is run by the QCMDEXC program. If the create command fails, MONMSG commands are used to handle the escape messages. If the create is successful, the actual library is checked for the existence of an object of the same name/type. If an old version exists, the GRTOBJAUT command is used to grant the existing authorities to the new version. If the object is a physical file and an old version exists, the old version will not be deleted. The new version will be moved to the library TEMPCRT. If a version already exists in TEMPCRT, it is considered an error situation. Physical files are not handled as other objects because it may be necessary to copy/reformat the data from one or more members and handle logical files which are built over the physical files.

The new object is moved to the actual library. The completion code is set to Y (successful) and the MNUCRT program is called with the parameter to note the end function. The MNUCRT program will update the job record with the date/time and completion code.

If the create command failed, the completion code is set to N and the MNUCRT program is called to update the job record. The CL program then ends by sending an escape message. This causes the job to end abnormally which will cause a message to be sent back to the originating work station.

6. Create the RPG/400 program MNUCRT, which is called by both the MNUSBMC and MNUCRTC programs to write, read or update the MNUCRTP database file. The source for the RPG/400 program MNUCRT follows:

```
*... ... 1 ... ... 2 ... ... 3 ... ... 4 ... ... 5 ... ... 6 ... ... 7 ..

     FMNUCRTP UF  E           K        DISK                      A
     E                    STP     1  30 4   OTP     5   SRCTYP/OBJTYP
     ICMD         DS                             512
     C           *ENTRY    PLIST
     C                     PARM           RPGCDE  1
     C                     PARM           JOB
     C                     PARM           USER
     C                     PARM           JOBNBR
     C                     PARM           MEMBER
     C                     PARM           SRCTYP
     C                     PARM           SRCFIL
     C                     PARM           SRCLIB
     C                     PARM           OBJLIB
     C                     PARM           JOBD
     C                     PARM           F4
     C                     PARM           CMD
     C                     PARM           RQSLEN
     C                     PARM           CRTNAM
     C                     PARM           CRTLIB
     C                     PARM           OBJTYP
     C                     PARM           COMPLT  1
     C                     PARM           RETURN  8
     C* W = Write new record to MNUCRTP file
     C           RPGCDE    IFEQ 'W'                         RPGCDE = W
     C                     TIME           SBMTIM            Time
     C                     MOVE UPDATE    SBMDAT            Date
     C* Lookup the type from the menu and convert to CPF type
     C                     Z-ADD1         X       30        Initialize
     C           SRCTYP    LOKUPSTP,X                     20Mnu type
     C   20                MOVE OTP,X     OBJTYP            CPF type
     C                     WRITEMNUCRTR                     Add new
     C                     END                              RPGCDE = W
     C* S = Create job has started in batch. Retrieve record, update
     C*     with start information and return it to CL program.
     C           RPGCDE    IFEQ 'S'                         RPGCDE = S
     C           KEY       KLIST                            Key to file
     C                     KFLD           JOB
     C                     KFLD           USER
     C                     KFLD           JOBNBR
     C           KEY       CHAINMNUCRTR             61      Not found
     C   61                MOVE 'NOTFOUND'RETURN            Error rtn
     C   61                GOTO ENDPGM                      End pgm
     C                     TIME           STRTIM            Time
     C                     MOVE UPDATE    STRDAT            Date
     C                     EXCPTSTART                       Update start
     C                     END                              RPGCDE = S
     C* E = Create job has ended in batch. Retrieve record and update
     C*      it with completion information.
     C           RPGCDE    IFEQ 'E'                         RPGCDE = E
     C           KEY       CHAINMNUCRTR             61      Not found
     C   61                MOVE 'NOTFOUND'RETURN            Error rtn
```

```
SEQNBR *... ... 1 ... ... 2 ... ... 3 ... ... 4 ... ... 5 ... ... 6 ... ... 7 ..
           C   61                 GOTO ENDPGM                     End pgm
           C                      TIME           CMPTIM           Time
           C                      MOVE UDATE     CMPDAT           Date
           C                      MOVE COMPLT    CMPCDE           Move to code
           C                      EXCPTUPDATE                     Update end
           C                      END                             RPGCDE = E
           C                ENDPGM    TAG
           C                      SETON                     LR    Set LR
           C                      RETRN                           Return
           OMNUCRTR E                 START
           O                          STRTIM
           O                          STRDAT
           OMNUCRTR E                 UPDATE
           O                          CMPTIM
           O                          CMPDAT
           O                          CMPCDE
       ** SP/OTP    SOURCE TYPE AND OBJECT TYPE
       BAS PGM
       CBL PGM
       CL  PGM
       CLP PGM
       RPG PGM
       RPT PGM
       DFU PGM
       QRY PGM
       CMD CMD
       DSPFFILE
       LF  FILE
       PF  FILE
       PRTFFILE
       ICFFILE
```

The RPG/400 program accepts the same parameter list from either MNUSBMC or MNUCRTC. It analyzes the RPGCDE value to see the type of request desired. The W code requests a new record be written. The submit date/time is retrieved. The source type is looked up in a table to return the object type which is needed in the MNUCRTC program and the new record is added.

If the RPGCDE value is S (start), the job record should be retrieved and passed back to the MNUCRTC program. This is the start of the batch create function. The record is updated with the start date and time. If the record cannot be found, an error is returned.

If the RPGCDE value is E, the create function is ended and the job record should be updated with the date/time and the completion code (Y or N).

7. Run the STRPGMMNU command, which would normally contain initial values for the programmer menu and call the MNUSBMC program for option 3.

   ```
   STRPGMMNU ..... EXITPGM(MNUSBMC) DLTOPT(*NODLT)
   ```

   Note that the DLTOPT(*NODLT) option has been requested. This specifies that F11 is to be ignored by the Programmer menu. In this example, if the object already exists, it is implicitly replaced.

Application Programming for DBCS Data

Special considerations must be made when designing application programs to process double-byte data or converting alphanumeric application programs to double-byte programs.

Designing DBCS Application Programs

Design your application programs for processing double-byte data in the same way you design application programs for processing alphanumeric data, with the following additional considerations:

- Identify double-byte data used in the database files, if any.
- Design display and printer formats that can be used with double-byte data.
- If needed, provide double-byte conversion as a means of entering data for interactive applications. Use the DDS keyword for double-byte conversion (IGCCNV) to specify DBCS conversion in display files.
- Write double-byte error messages to be displayed by the program.
- Specify extension character processing so that the system prints and displays all double-byte data.
- Determine which double-byte characters, if any, must be defined. The *CGU User's Guide* describes how to define double-byte characters for DBCS-supported countries.

Converting Alphanumeric Programs to Process DBCS Data

If an alphanumeric application program uses externally-described display files, you can change that application program to a double-byte application program by changing only the files. To convert an application program, do the following:

1. Create a duplicate copy of the source statements for the alphanumeric file you want to change.
2. Change alphanumeric constants and literals to double-byte constants and literals.
3. Change fields in the file to the open (o) keyboard shift so that you can enter both double-byte and alphanumeric data in these fields. 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 will be used. The library in which the double-byte display file is stored is then checked before the library in which the alphanumeric version of the file is stored.

Using DBCS Data in a CL Program

The following program shows the use of different keyboard shifts within a CL program. Note how the double-byte data is used only as text values in this program; the commands themselves are in alphanumeric characters.

When run, this program shows you how the different keyboard shifts for DDS display files are used. See the *DDS Reference* for information about double-byte keyboard shifts.

```
PGM
      DCLF     IGCTEST
START: CHGVAR &OUTPUTA   'ABCDEFGHIJ'
       CGVAR  &OUTPUTJ   'ABCD'
       CGVAR  &BOTHJ     'ABCD'
       CGVAR  &OUTPUTE   'EFGH'
       CGVAR  &OUTBUTO   'AB C D F'
LOOP:  SBDRCVF
       IF & IN01 RETURN
       CHGVAR &OUTPUTA  &INPUTA
       CHGVAR &OUTPUTJ  &INPUTJ
       CHGVAR &OUTPUTE  &INPUTE
       CHGVAR &BOTHE    &INPUTE
       CHGVAR &OUTPUTO  INPUTO
     GOTO LOOP
ENDPGM
```

RSLS301-0

Sample CL Programs

The following sample programs demonstrate the flexibility, simplicity, and versatility of CL programs. The following programs are described by their function and probable user.

Initial Program for Setup (Programmer)

```
PGM
CHGLIBL LIBL(TESTLIB QGPL QTEMP)
CHGJOB OUTQ(WSPRTR)
TFRCTL QPGMMENU
ENDPGM
```

The test library is placed first on the library list, an output queue is selected for a convenient printer, and the programmer menu is displayed.

Moving an Object from a Test Library to a Production Library (Programmer)

```
PGM PARM(&OBJ &OBJTYPE &OPER)
DCL &OBJ *CHAR LEN(10)
DCL &OBJTYPE *CHAR LEN(7)
DCL &OPER *CHAR LEN(1)  /* R=Replace M=Move */
IF ((&OPER *NE 'M') *AND (&OPER *NE 'R')) THEN(DO)
        SNDPGMMSG MSG('Operation code must be "R" or "M" ')
        RETURN
        ENDDO
IF ((&OBJTYPE *NE *PGM) *AND (&OBJTYPE *NE *FILE) *AND (&OBJTYPE +
    *NE *DTAARA)) THEN(DO)
        SNDPGMMSG MSG('Object' *BCAT &OBJ *BCAT ' must be *PGM, +
                  *FILE, or *DTAARA')
        RETURN
        ENDDO
CHKOBJ BLDLIB/&OBJ OBJTYPE(&OBJTYPE)
MONMSG MSGID(CPF9801) EXEC(DO)
        SNDPGMMSG MSG('Object or object type does not exist +
                  in BLDLIB')
        RETURN
        ENDDO
IF (&OPER *EQ 'M') THEN(DO)
        MOVOBJ BLDLIB/&OBJ OBJTYPE(&OBJTYPE) TOLIB(PRODLIB)
        MONMSG MSGID(CPF3208) EXEC(DO)
                        SNDPGMMSG MSG('Object' *BCAT &OBJ *BCAT ' +
                                  already exists in PRODLIB')
                        RETURN
                        ENDDO
        CHKOBJ PRODLIB/&OBJ OBJTYPE(&OBJTYPE)
        MONMSG MSGID(CPF9801) EXEC(DO)
        SNDPGMMSG MSG('Object or object type does not +
                  exist in PRODLIB')
                  RETURN
                  ENDDO
        ENDDO
RETURN
ENDPGM
```

The object name, object type, and operation code are passed from another program. Checks are performed to see that the operation code and object type are correct, and that the object exists in the test library. The object is moved unless it already exists in the production library. The move is then confirmed. More commands can be added to grant additional authority to the object or to handle additional exceptions and additional object types.

Saving Specific Objects in an Application (System Operator)

Example

```
PGM
SAVOBJ OBJ(FILE1 FILE2) LIB(LIBA) OBJTYPE(*FILE) LOC(*S12) +
  CLEAR(*YES)
SAVOBJ OBJ(DTAARA1) LIB(LIBA) OBJTYPE(*DTAARA) LOC(*S12)
SNDPGMMSG MSG('Save of daily backup of LIBA completed') +
  MSGTYPE(*COMP)
ENDPGM
```

This program ensures consistent command entry for regularly repeated procedures.

Additional Save Object (SAVOBJ) commands can, of course, be added. However, this program relies on the operator selecting the correct diskette or tape for each periodic backup of each application. This can be controlled by assigning unique names to each diskette or tape set for each save operation. If you want to save your payroll files separately each week for four weeks, for instance, you might name each diskette or tape differently and write the program to compare the name of the diskette or tape against the correct name for that week.

Recovery from Abnormal End (System Operator)

```
PGM
DCL &SWITCH *CHAR LEN(1)
RTVSYSVAL SYSVAL(QABNORMSW) RTNVAR(&SWITCH)
IF (&SWITCH *EQ '1') THEN(DO)  /*CALL RECOVERY PROGRAMS*/
        SNDPGMMSG MSG('Recovery programs in process. +
          Do not start subsystems until notified') +
          MSGTYPE(*INFO) TOMSGQ(QSYSOPR)
        CALL PGMA
        CALL PGMB
        SNDPGMMSG MSG('Recovery programs complete. +
          Startup subsystems') +
          MSGTYPE(*INFO) TOMSGQ(QSYSOPR)
        RETURN
        ENDDO
ENDPGM
```

Submitting a Job (System Operator)

```
PGM  /*DAILYAC*/
SBMJOB JOB(DAILYACCRC) JOBD(ACCRC2) +
             CMD(CALL ACCRC305 PARM(DAILY))
SNDPGMMSG    MSG('Daily Accounts Receivable job DAILYACCRC +
             submitted to batch') MSGTYPE(*COMP)
ENDPGM
```

Instead of typing in all the parameters for submitting a job, the system operator simply calls DAILYAC.

Timing Out While Waiting for Input from a Device Display

```
             PGM
             DCLF FILE(QGPL/MENU)
START:       SNDRCVF DEV(*FILE) RCDFMT(MENUFMT) WAIT(*NO)
             WAIT
             MONMSG MSGID(CPF0889) EXEC(SIGNOFF)
             CHGVAR VAR(&IN99) VALUE('0')
             IF COND(&IN01) THEN(GOTO CMDLBL(START))
OPTION1:     /* OPTION 1-ORDER ENTRY */
             IF COND(&OPTION *EQ '1') THEN(DO)
             CALL PGM(ORDENT)
             GOTO CMDLBL(START)
             ENDDO
OPTION2:     /* OPTION 2-ORDER DISPLAY */
             IF COND(&OPTION *EQ '2') THEN(DO)
             CALL PGM(ORDDSP)
             GOTO CMDLBL(START)
             ENDDO
OPTION3:     /* OPTION 3-ORDER CHANGE */
             IF COND(&OPTION *EQ '3') THEN(DO)
             CALL PGM(ORDCHG)
             GOTO CMDLBL(START)
             ENDDO
OPTION4:     /* OPTION 4-ORDER PRINT */
             IF COND(&OPTION *EQ '4') THEN(DO)
             CALL PGM(ORDPRT)
             GOTO CMDLBL(START)
             ENDDO
OPTION9:     /* OPTION 9-SIGNOFF */
             IF COND(&OPTION *EQ '9') THEN(SIGNOFF)
OPTIONERR:   /* OPTION SELECTED NOT VALID */
             CHGVAR VAR(&IN99) VALUE('1')
             GOTO CMDLBL(START)
             ENDPGM
```

This program illustrates how to write a CL program using a display file that will wait for a specified amount of time for the user to enter an option. If he does not, the user is signed off.

The display file was created with the following command:

```
CRTDSPF FILE(MENU) SRCFILE(QGPL/QDDSSRC) SRCMBR(MENU)  +
    DEV(*REQUESTER) WAITRCD(60)
```

The display file will use the *REQUESTER device. When a WAIT command is issued, it waits for the number of seconds (60) specified on the WAITRCD keyword. The following is the DDS for the display file:

```
SEQNBR *... ... 1 ... ... 2 ... ... 3 ... ... 4 ... ... 5 ... ... 6 ... ... 7 ... ... 8

 0100      A                                      PRINT CA01(01)
 0200      A          R MENUFMT                   BLINK
 0300      A                                      TEXT('Order Entry Menu')
 0400      A                                  1 31'Order Entry Menu'
 0500      A                                  2  2'Select one of the following:
 0600      A                                  3  4'1.  Enter Order'
 0700      A                                  4  4'2.  Display Order'
 0800      A                                  5  4'3.  Change Order'
 0900      A                                  6  4'4.  Print Order'
 1000      A                                  7  4'9.  Sign Off'
 1100      A                                 23  2'Option:'
 1200      A            OPTION         1   I 23 10
 1300      A  99                                  ERRMSG('Invalid option selected.')

                            * * * * * *  E N D  O F  S O U R C E  * * * * *
```

The program performs a SNDRCVF WAIT(*NO) to display the menu and request an option from the user. Then it issues a WAIT command to accept an option from the user. If the user enters a 1 through 4, the appropriate program is called. If the user enters a 9, the SIGNOFF command is issued. If the user enters an option that is not valid, the menu is displayed with an 'OPTION SELECTED NOT VALID' message. The user can then enter another valid option. If the user does not respond within 60 seconds, the CPF0889 message is issued to the program and the MONMSG command issues the SIGNOFF command.

A SNDF command using a record format containing the INVITE DDS keyword could be used instead of the SNDRCVF WAIT(*NO). The function would be the same.

Converting from and to Binary Data

Some CL commands require binary data or provide binary data output in messages, and certain command functions pass binary data to command-processing programs. However, CL programs can use only packed decimal or character data. To convert from binary data or to binary data so that the resulting value can be processed in CL programs, you may want to use a technique whereby character fields are defined to contain the binary values. Calling a high-level language (HLL) program allows the values to be converted from or to packed decimal fields.

The following sample commands provide this type of conversion. Although binary fields can be either 2 or 4 bytes in length, these examples use only 2-byte binary fields.

CVTBINDEC (Convert from Binary to Decimal) Sample Command

The CVTBINDEC sample command converts a binary value stored in a 2-byte character field into a 5-digit packed decimal field. Only values of -9999 to +9999 can be represented in a 2-byte binary value. The extra digit allows the packed field to be used internally with better efficiency.

The following is the command definition for the CVTBINDEC command:

```
SEQNBR *... ... 1 ... ... 2 ... ... 3 ... ... 4 ... ... 5 ... ... 6 ... ... 7 ..

 01.00          CMD        PROMPT('Convert from binary to decimal')
 02.00                     /* CPP is CVTBINDEC */
 03.00          PARM       KWD(FROMBIN) TYPE(*CHAR) LEN(2) MIN(1) +
 04.00                       PROMPT('From character variable:')
 05.00          PARM       KWD(TODEC) TYPE(*DEC) LEN(5 0)  RTNVAL(*YES) +
 06.00                       MIN(1) PROMPT('To decimal variable:')
```

The command should be created with the parameters PGM(CVTBINDEC) and ALLOW(*IPGM *BPGM). The ALLOW parameter must be specified as the command provides a return variable that can be used only in a CL program.

The following is the command processing program that is written RPG/400 in:

```
SEQNBR *... ... 1 ... ... 2 ... ... 3 ... ... 4 ... ... 5 ... ... 6 ... ... 7 ..

      IDS1          DS
      I                                          B    1   20VAR1B
      C            *ENTRY    PLIST
      C                      PARM               DS1       2         From var
      C                      PARM VAR1B         VAR2      50        To var
      C                      SETON                           LR
      C                      RETRN
```

CVTDECBIN (Convert Decimal to Binary) Sample Command

The CVTDECBIN sample command converts a 5-digit decimal field to a 2-byte character field. Only values of -9999 to +9999 can be converted.

The following is the command definition for the CVTDECBIN sample command.

```
SEQNBR *... ... 1 ... ... 2 ... ... 3 ... ... 4 ... ... 5 ... ... 6 ... ... 7 ..

                 CMD        PROMPT('Convert from decimal to binary')
                            /* CPP is CVTDECBIN */
                 PARM       KWD(FROMDEC) TYPE(*DEC) LEN(5 0) +
                              RANGE(-9999 9999) MIN(1) +
                              PROMPT('From decimal variable:')
                 PARM       KWD(TOBIN) TYPE(*CHAR) LEN(2)  RTNVAL(*YES) +
                              MIN(1) PROMPT('To character variable:')
```

The command should be created with the parameters PGM(CVTDECBIN) and ALLOW(*IPGM *BPGM). The ALLOW parameter must be specified because the command provides a return variable that can be used only in a CL program.

The following is the command-processing program that is written in RPG/400. A data structure is used to define the same data as both character and binary.

```
SEQNBR *... ... 1 ... ... 2 ... ... 3 ... ... 4 ... ... 5 ... ... 6 ... ... 7 ..

      IDS1          DS
      I                                          B    1   20VAR1B
      C            *ENTRY    PLIST
      C            VAR1B     PARM               VAR1      50        From var
      C                      PARM               DS1                 To var
      C                      SETON                           LR
      C                      RETRN
```

Retrieving Program Attributes

The Display Program (DSPPGM) command can be used 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, the Display Object Description (DSPOBJD) command can be used to build an output file. The output file can then be read within a CL program using the Declare File (DCLF) and Receive File (RCVF) commands. To access some of the other attributes of the DSPPGM command (such as USRPRF), the following sample command can be used.

The sample command Retrieve Program Attributes (RTVPGMATR) uses the DSPPGM command to send output to a spooled file. The file is then copied into a file in library QTEMP using the CPYSPLF command. The file is read, and the USRPRF parameter name is found within the output and the associated value.

Note: Because this sample command is reading output from the DSPPGM command print format, it may be affected by changes in a future release.

For example, the sample command is written to send an escape message if the parameter USRPRF is not found or if the returned value is not *USER or *OWNER. If this occurs, you must change the command-processing program to agree with the new printed format.

To use the sample command, perform the following steps:

1. Create the file TEMP132 in your library. This allows the DCLF command to access an existing definition to create the CL program.

```
CRTPF  FILE(JOE/TEMP132) RCDLEN(132) TEXT('File +
       used RTVPGMATR command')
```

2. Enter the following command definition for the RTVPGMATR command.

```
SEQNBR *... ... 1 ... ... 2 ... ... 3 ... ... 4 ... ... 5 ... ... 6 ... ... 7 ..

 01.00             CMD        PROMPT('Retrieve Program Attributes')
 02.00                        '/* CPP is RTVPGMATRC */
 03.00             PARM       KWD(PGM) TYPE(QUAL1) MIN(1) +
 04.00                          PROMPT('Program name:')
 05.00  QUAL1      QUAL       TYPE(*NAME) LEN(10) MIN(1) EXPR(*YES)
 06.00             QUAL       TYPE(*NAME) LEN(10) DFT(*LIBL) +
 07.00                          SPCVAL((*LIBL)) EXPR(*YES) +
 08.00                          PROMPT('Program name:')
 09.00             PARM       KWD(USRPRF) TYPE(*CHAR) LEN(6)  RTNVAL(*YES) +
 10.00                          MIN(1) PROMPT('Variable for  USRPRF value:')
```

3. Create the RTVPGMATR command specifying PGM(RTVPGMATRC) and ALLOW(*IPGM *BPGM). The ALLOW parameter is required because of the return variable used.

4. Enter the following source and create the CL program RTVPGMATRC:

```
SEQNBR *... ... 1 ... ... 2 ... ... 3 ... ... 4 ... ... 5 ... ... 6 ... ... 7 ..

             PGM        PARM(&FULLPGM &USRPRF)
             DCLF       TEMP132
             DCL        &FULLPGM *CHAR LEN(20)
             DCL        &PGM *CHAR LEN(10)
             DCL        &LIB *CHAR LEN(10)
             DCL        &USRPRF *CHAR LEN(6)
             DCL        &MSGID *CHAR LEN(7)
             DCL        &MSGDTA *CHAR LEN(100)
             DCL        &VALID *CHAR LEN(10) /* Valid prt  format switch */
             MONMSG     MSGID(CPF0000) EXEC(GOTO ERROR) /*  Any error */
             CHGVAR     &PGM %SST(&FULLPGM 1 10) /* Extract pgm */
             CHGVAR     &LIB %SST(&FULLPGM 11 10) /* Extract  library */
             OVRPRTF    QPDPGM HOLD(*YES) /* Prevent printing */
             DSPPGM     &PGM &LIB OUTPUT(*LIST) /* Output is  spooled */
             CRTPF      QTEMP/TEMP132 RCDLEN(132) /*  Temporary file */
             CPYSPLF    FILE(QPDPGM) TOFILE(QTEMP/TEMP132) +
                             SPLNBR(*LAST)
             OVRDBF     TEMP132 TOFILE(QTEMP/TEMP132) /* Use QTEMP */
 READ:       RCVF       /* Read TEMP132 */
             MONMSG     MSGID(CPF0864)  EXEC(GOTO END) /* EOF */
             IF         (%SST(&TEMP132 37 6) *EQ 'USRPRF') DO
             CHGVAR     &USRPRF %SST(&TEMP132 47 6) /*  Parameter value */
             IF         ((&USRPRF *EQ '*USER') *OR +
                           (&USRPRF *EQ '*OWNER')) CHGVAR &VALID '1'
             ENDDO      /* Parameter = 'USRPRF' */
             GOTO       READ
  END:       DLTSPLF    FILE(QPDPGM) SPLNBR(*LAST)
             DLTF       QTEMP/TEMP132 /* Delete temporary file */
             IF         (&VALID *EQ '0') SNDPGMMSG MSGID(CPF9898) +
                           MSGF(QCPFMSG) MSGTYPE(*ESCAPE) +
                           MSGDTA('The print format for the DSPPGM +
                           command has changed from the previous +
                           release')
             RETURN
  ERROR:     RCVMSG     MSGID(&MSGID) MSGDTA(&MSGDTA)
             SNDPGMMSG  MSGID(&MSGID) MSGF(QCPFMSG)  MSGTYPE(*ESCAPE) +
                          MSGDTA(&MSGDTA)
             ENDPGM
```

The RTVPGMATR command returns only the value for the USRPRF parameter. Other parameters could be added in a similar way. To use the command, the following would be specified:

```
DCL  &USRPRF *CHAR LEN(6)
.
.
RTVPGMATR  PGM(yyy/xxx) USRPRF(&USRPRF)
```

Chapter 7. Defining Messages

On the AS/400 system, communication between programs, between jobs, between users, and between users and programs occurs through messages. A message can be predefined or immediate:

- A predefined message is created and exists outside the program that uses it. Predefined messages are stored in message files and have a message number. An example of a system predefined message is:

  ```
  CPF0006  Errors occurred in command.
  ```

- An immediate message is created by the sender at the time it is sent. An immediate message is not stored in a message file because it is usually sent only once. An example of an immediate message received at a display station is:

  ```
  From . . . :   QSYSOPR      06/12/88   10:50:54
  System going down at 11:00; please sign off
  ```

Your system comes with an extensive set of predefined messages that allow communication between programs within the system and between the system and its users. Each licensed program you order has a message file that is stored in the same library as the licensed program it applies to. For example, system messages are stored in the file QCPFMSG in the library QSYS.

Each predefined message in a message file is uniquely identified by a 7-character code and is defined by a message description. The message description contains information, such as message text and message help text, severity level, valid and default reply values, and various other attributes. See the Add Message Description (ADDMSGD) command description in the *CL Reference*.

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 or program must receive the message from the queue or display it. There are several IBM-supplied message queues in the system; these message queues are described later in this chapter (see "Types of Message Queues" on page 7-18).

The system also writes some of the messages that are issued to logs. A job log contains information related to requests entered for a job, the history log contains job, subsystem, and device status information. See "Message Logging" on page 8-33 for more information on logging.

You can create your own message files and message descriptions. By creating predefined messages, you can use the same message in several 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 program that uses them. If the messages were included in the program, the 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)
- Change message descriptions (Change Message Description [CHGMSGD] command)
- Remove message descriptions (Remove Message Description [RMVMSGD] command)
- Send immediate messages (Send Message [SNDMSG], Send Break Message [SNDBRKMSG], Send Program Message [SNDPGMMSG], and Send User Message [SNDUSRMSG] commands)
- Display messages and message descriptions (Display Messages [DSPMSG], Display Message Description [DSPMSGD], and Work with Message Descriptions [WRKMSGD] commands)
- Use a CL program to:
  - Send a message to a work station 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 program's program 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 program requires corrective action or a reply from its calling program. A program can monitor for the arrival of notify messages from the programs 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 program. (For example, a CL command can be 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 program. A program can monitor for the arrival of status messages from the program it calls. Status messages sent to the external message queue (*EXT) are shown at the display station and can be used to inform the display station user of an operation in progress.
- Escape (*ESCAPE). A message that describes a condition for which a program must end abnormally. A program can monitor for the arrival of escape messages from the program it calls or from the machine.

This chapter describes:

- How to create your own message files
- How to add message descriptions to a message file
- Types of message queues
- How to create message queues

Creating a Message File

To create your own predefined messages, you must first create the message file into which the messages are to be placed. Use the Create Message File (CRTMSGF) command to create the message file. You then use the Add Message Description (ADDMSGD) command to describe your messages and place them in the message file.

On the CRTMSGF command, you can specify the maximum size in K bytes on the SIZE parameter. The following formula can be used to determine the maximum:

```
S + (I x N)
```

where:

S Is the initial amount of storage

I Is the amount of storage to add each time

N Is the number of times to add storage

The defaults for S, I, and N are 10, 2, and *NOMAX, respectively.

For example, you specify S as 5, I as 1, and N as 2. When the file reaches the initial storage amount of 5K, the system automatically adds another 1K to the initial storage. The amount added, 1K, can be added to the storage two times to make the total maximum of 7K. If you specify *NOMAX as N, the message file does not have a maximum size.

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. Therefore, it is important to calculate the size needed for your message file when you create it, or specify *NOMAX.

You can determine the size of a message by using the following formula. (The ADDMSGD command parameters are given in parentheses.)

- Message text (MSG) equals 16 bytes base plus the length of the message.
- Message help text (SECLVL), if any, equals 16 bytes base plus the length of the second-level text.
- 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 and dump list (DFTPGM and DMPLST) equal 35 plus (2 x number in DMPLST).
- ALROPT equals 12 bytes.

The smallest possible entry in a message file is 17 bytes and the largest possible entry is 5331 bytes. The following table describes the largest possible entry:

Message text	148 bytes
Message help text	3016 bytes
20 formats	74 bytes
Type and length	48 bytes
20 special values	1282 bytes
20 values	640 bytes
Default	32 bytes
Default program and dump list	79 bytes
Alert option	12 bytes

In the following example, the CRTMSGF command creates the message file USRMSG:

```
CRTMSGF   MSGF(QGPL/USRMSG) +
          TEXT('Message file for user-created messages')
```

If you are creating a message file to be used with the DSPLY operation code in RPG/400, the message file must be named QUSERMSG.

Adding Messages to a File

You use the Add Message Description (ADDMSGD) command to describe your predefined messages and to add them to the message file you created. On the ADDMSGD command, you specify the message identifier, the name of the message file into which the message is to be placed, and the message description. In the message description, you can specify:

- Message text (required) with optional substitution variables
- Message help text with optional substitution variables
- Severity code
- Description of the format of the message data to be used for the substitution variables
- Validity checking values for a reply
- Default value for a reply
- Default message handling action for escape messages
- Creation level
- Alert options
- Entry in the error log

Each of the items that can be contained in the message description is described in more detail on the following pages.

The following commands are also available for use with message descriptions:

Change Message Description (CHGMSGD)
Changes a message description.

Display Message Description (DSPMSGD)
Displays a message description. (A range of message identifiers can be specified in this command.)

Remove Message Description (RMVMSGD)
Removes a message description from a message file.

Retrieve Message (RTVMSG)
Retrieves a message from a message file.

Merge Message File (MRGMSGF)
Merges messages from one message file into another message file.

Work with Message Descriptions (WRKMSGD)
Displays a list of messages in a file and allows you to add, change, or delete message descriptions.

Assigning a Message Identifier

The message identifier you specify on the ADDMSGD command is used to refer to the message and is the name of the message description. The message identifier must be 7 characters:

```
pppmmnn
```

where `ppp` is the product or application code, `mm` is the numeric group code, and `nn` is the numeric subtype code. The number specified as `mmnn` can be used to further divide a set of product or application messages. Numeric group and subtype codes consist of decimal numbers 0 through 9 and the characters A through F. 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:

```
CPF1234
```

is message 1234 of CPF.

When you create your own messages, using the letter U as the first character in the product code is a good way to distinguish your messages from system messages. For example:

```
USR3567
```

The first character of the code must be alphabetic, the second and third characters can be alphanumeric; the group code and the subtype code must consist of decimal numbers 0 through 9 and the characters A through F. Note that although this range can be called a set of hexadecimal numbers, any sorting of the message numerics treats A through F as characters.

For example, when displaying a range of message descriptions, CPFA000 precedes CPF1000.

You should use care in using a numeric subtype code of 00 in the message identifier. If you use a numeric subtype code of 00 for a message that can be sent as an escape, notify, or status message and that can, therefore, be monitored, a subtype code of 00 in the Monitor Message (MONMSG) command causes all messages in the numeric group to be monitored. See “Monitoring for Messages in a CL Program” on page 8-16 for more information.

Defining Message Text and Message Help Text

You can define two levels of message text on the ADDMSGD command. The message text is required and should identify the condition that caused the message to be issued. The message help text is optional and should explain the condition further or explain the corrective action to be taken. To get the message help text, the display station user must move the cursor to the message line and press the Help key when the message text is displayed. Message help text can be formatted for the display station using three format control characters. These characters may be used to make the message help text (usually online help information) more readable for the user.

Each of the three format control characters must be followed by a blank to separate them from the message text.

&Nb (where b is a blank) Forces the text to a new line (column 2). If the text is longer than one line, the next lines are indented to column 4 until the end of the text or until another format control character is found.

&Pb (where b is a blank) Forces the text to a new line, indented to column 6. If the text is longer than one line, the next lines start in column 4 until the end of the text or until another format control character is found.

&Bb (where b is a blank) Forces the text to a new line, starting in column 4. If the text is longer than one line, the next lines are indented to column 6 until the end of the text or until another format control character is found.

Assigning a Severity Code

The severity code you assign to a message on the ADDMSGD command indicates how important the message is. The higher the severity code the more serious the condition is. The following lists the severity codes you can use and their meanings. (These severity codes and their meanings are consistent with the severity codes assigned to IBM-predefined messages.)

00: Information. For information purposes only; no error was detected and no reply is needed. The message could indicate that a function is in progress or that a function has completed successfully.

10: Warning. A potential error condition exists. The 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 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 the program was unable to handle data that was not valid, or possibly because the user has canceled it.

50: Abnormal end of job. The job was ended or was not started. A routing step may have ended abnormally or failed to start, a job-level function may not have been performed as required, or the job may have been canceled.

60: System status. Issued only to the system operator. It gives either the status of or a warning about a device, a subsystem, or the system.

70: Device integrity. Issued only to the system operator. It indicates that a device is malfunctioning or in some way is no longer operational. The user may be able to recover from the failure, or the assistance of a service representative may be required.

80: System alert. A message with a severity code of 80 is issued for immediate messages. It also warns of a condition that, although not severe enough to stop the system now, could become more severe unless preventive measures are taken.

90: System integrity. Issued only to the system operator. It describes a condition that renders either a subsystem or the system inoperative.

99: Action. Some manual action is required, such as entering a reply, changing printer forms, or replacing diskettes.

For a detailed discussion of the SEV parameter, see the *CL Reference*.

Defining Substitution Variables

On the FMT parameter on the ADDMSGD command, you can specify substitution variables for either first- or second-level messages. For example:

```
File &1 not found
```

contains the substitution variable &1. When the message is displayed or retrieved, the variable &1 is replaced with the name of the file that could not be found. This name is supplied by the sender of the message. For example:

```
File ORDHDRP not found
```

Compare this to:

```
File not found
```

Substitution variables can make your message more specific and more meaningful.

The substitution variable must begin with & (ampersand) and be followed by n, where n is any number from 1 through 99. For example, for the message:

```
File &1 not found
```

the substitution variable is defined as:

```
FMT((*CHAR 10))
```

When you assign numbers to substitution variables, you must begin with the number 1 and use the numbers consecutively. For example, &1, &2, &3, and so on. However, you do not have to use all the substitution variables defined for a message description in the message that is sent.

For example, the message:

```
File &3 not available
```

is valid even though &1 and &2 are not used in the messages. However, to do this, you must define &1, &2, and &3 on the FMT parameter of the ADDMSGD command. For the preceding message, the FMT parameter could be:

```
FMT((*CHAR 10) (*CHAR 2) (*CHAR 10))
```

where the first value describes &1, the second &2, and the third &3. The description for &1 and &2 must be present if &3 is used. In addition, when this message is sent, the MSGDTA parameter on the Send Program Message (SNDPGMMSG) command should include all the data described on the FMT parameter. To send the preceding message, the MSGDTA parameter should be at least 22 characters long.

For the preceding message, you could also specify the FMT parameter as:

```
FMT((*CHAR 0) (*CHAR 0) (*CHAR 10))
```

Because &1 and &2 are not used in the message, they can be described with a length of 0. Then no message data needs to be sent. (The MSGDTA parameter on the SNDPGMMSG command needs to be only 10 characters long in this case.)

An example of using &3 in the message and including &1 and &2 in the FMT parameter is when &1 and &2 are specified on the DMPLST parameter. (The

DMPLST parameter specifies that the data is to be dumped when this message is sent as an escape message to a program that is not monitoring for it.)

The substitution variables do not have to be specified in the message in the same order in which they are defined in the FMT parameter. For example, three values can be defined in the FMT parameter as:

```
FMT((*CHAR 10) (*CHAR 10) (*CHAR 7))
```

The substitution variables can be used in the message as follows:

```
Object &1 of type &3 in library &2 is not available
```

If this message is sent in a CL program, you can concatenate the values used for the message data such as:

```
SNDPGMMSG .....MSGDTA(&OBJ *CAT &LIB *CAT &OBJTYPE)
```

You must specify the format of the message data field for the substitution variable by specifying data type and, optionally, length on the ADDMSGD command. The valid data types for message data fields are:

- Quoted character string (*QTDCHAR). A string of character data to be enclosed in apostrophes. Preceding and trailing blanks are not deleted. If length is not specified in the message description, the sender determines the length of the field.
- Character string without quotation marks (*CHAR). A string of character data not to be enclosed in apostrophes. Trailing blanks are deleted. If length is not specified in the message description, the sender determines the length of the field.
- Hexadecimal (*HEX). A string to be preceded by the character X and enclosed in apostrophes; each byte of the string is to be converted into two hexadecimal characters (0 through 9 and A through F). If length is not specified in the message description, the sender determines the length of the field.
- Binary (*BIN). A binary integer (either 2 or 4 bytes long) formatted as a signed decimal integer. If length is not specified, 2 is assumed.
- 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.

The following data types are valid only in IBM-supplied message descriptions and should not be used for other messages:

- Time interval (*ITV). An 8-byte time interval that contains the time to the nearest whole second for various wait time out conditions.
- Date and time stamp (*DTS). An 8-byte system date and time stamp for which the date is to be formatted as specified in the QDATFMT and QDATSEP system values and the time is to be formatted as hh:mm:ss.

- System pointer (*SYP). A 16-byte pointer to a system object. In a message or message help, the name of the object is formatted the same as the *CHAR type data.
- Space pointer (*SPP). A 16-byte pointer to a program object. In a dump, the data in the object is formatted the same as the *HEX type data. *SPP cannot be used as substitution text in a message; it can only be used as part of the DMPLST parameter on the ADDMSGD command.

Specifying Validity Checking for Replies

On the ADDMSGD command, you can specify the type of reply that is valid for an inquiry or notify message. You can specify (parameters are given in parentheses):

- Type of reply (TYPE)
  - Decimal (*DEC)
  - Character (*CHAR)
  - Alphabetic (*ALPHA)
  - Name (*NAME)
- Maximum length of reply (LEN)
  - For decimal, 15 digits (9 decimal positions)
  - For character and alphabetic, 32 characters
  - For name, 10 characters

  Note: If you do not specify any validity checking (VALUES, RANGE, REL, SPCVAL, DFT), the maximum length of a reply is 132 characters for types *CHAR and *ALPHA.

- Values that can be used for the reply (VALUES SPCVAL RANGE REL)
  - A list of values
  - A list of special values
  - A range of values
  - A simple relationship that the reply value must meet

  Note: The special values are values that can be accepted but that do not satisfy any other validity checking values.

When a display station user enters a reply to a message, the keyboard is in lower shift which causes lowercase characters to be entered. If your program needs the reply to be in uppercase characters, you can do one of the following:

- Use the SNDUSRMSG command which supports a translation table option which defaults to converting lowercase to uppercase.
- Require the display station user to enter uppercase characters by specifying only uppercase characters for the VALUES parameter.
- Specify the VALUES parameter as uppercase and use the SPCVAL parameter to convert the corresponding lowercase characters to uppercase.
- Use TYPE(*NAME) if the characters to be entered are all letters (A-Z). The characters are converted to uppercase before being checked.

Sending an Immediate Message and Handling a Reply

In this example, the program does the following:

- Sends an immediate inquiry message to QSYSOPR
- Requests a reply of yes or no (Y or N)
- Ensures that a valid reply has been entered
- Does a time-out if the operator does not reply within 120 seconds

```
           PGM
           DCL        &MSGKEY *CHAR LEN(4)
           DCL        &MSGRPY *CHAR LEN(1)
SNDMSG:    SNDPGMMSG  MSG('.... Reply Y or N') TOMSGQ(QSYSOPR) +
                        MSGTYPE(*INQ) KEYVAR(&MSGKEY)
           RCVMSG     MSGTYPE(*RPY) MSGKEY(&MSGKEY) WAIT(120) +
                        MSG(&MSGRPY)
           IF         ((&MSGRPY *EQ 'Y') *OR (&MSGRPY *EQ 'y')) DO
           .
           .
           GOTO       END
           ENDDO      /* Reply of Y */
           IF         ((&MSGRPY *EQ 'N') *OR (&MSGRPY *EQ 'n')) DO
           .
           .
           GOTO       END
           ENDDO      /* Reply of N */
           IF         (&MSGRPY *NE ' ') DO
           SNDPGMMSG  MSG('Reply was not Y or N, try again') +
                        TOMSGQ(QSYSOPR)
           GOTO       SNDMSG
           ENDDO      /* Reply not valid */
    /* Timeout occurred */
           SNDPGMMSG  MSG('No reply from the previous message +
                        was received in 120 seconds and a ''Y'' +
                        value was assumed') TOMSGQ(QSYSOPR)
           .
           .
END:       ENDPGM
```

The SNDUSRMSG command cannot be used instead of this program because it does not support a time-out option (SNDUSRMSG waits until it receives a reply or until the job is cancelled).

The SNDPGMMSG command sends the message and specifies the KEYVAR parameter. This returns a variable 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 same message key value 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 logic checks for an upper or lower case value of the Y or N. Normally the value is entered by the operator as a lower case value. If the operator enters a non-blank value other than Y or N, the program sends a different message and then repeats the inquiry message.

If the operator had entered a blank, no reply is sent to the program. If a blank is returned to the program, the time out occurred (the operator did not reply). The program sends a message to the system operator stating that a reply *was not* received and the default was assumed (the ''Y'' value is shown as 'Y' in the message queue). Because the assumed value of 'Y' is not displayed as the reply, you cannot determine when looking at a message queue whether the message should be answered or has already timed out. The program cannot remove a message from the message queue once it has been sent. The second message should minimize this concern and provides an audit trail for what has occurred.

If the time out has already occurred and the operator replies to the message, the reply is ignored. The operator receives no indication that the reply has been ignored.

Sending Immediate Messages with Double-Byte Characters

To send an immediate message with double-byte text, limit the text to 37 double-byte characters plus the shift control characters. The limited size of the message ensures it is properly displayed.

Defining Default Values for Replies

The ADDMSGD command allows you to specify a default value for a reply to your message. A default reply must meet the same validity checking values as the other replies for the message or be specified as a special value in the message description. A default value is used when a user has indicated (using the CHGMSGQ command) that default replies should be issued for all inquiry messages sent to the user's message queue. Default replies are also sent when the unanswered inquiry messages are deleted. For example, the work station user uses the DSPMSG command to display messages, and removes unanswered inquiry messages by pressing either F13 to delete all the messages or F11 to delete a particular message.

Default replies are also used when the job attribute of INQMSGRPY is set to *DFT and may be used if set to *SYSRPYL option. You can use the system reply list to change the default reply.

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 program to which the message is sent does not monitor for or handle it.

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:
  - Program message queue name (10 characters). The name of the program message queue to which the message was sent (this is the same name as that of the program that did not monitor for the escape message).
  - Message reference key (4 characters). The message reference key of the escape message on the program message queue.
- Dump list. A list of message data field numbers (the same numbers as the substitution variables) that indicate which objects are to be dumped.

In addition, you can dump any of the following:

- The data areas for the job
- An internal machine data structure of a job
- A job

  Specifying a dump list for a job is equivalent to specifying the Display Job (DSPJOB) command with the parameters JOB(*) OUTPUT(*PRINT).

If you do not specify default actions in message descriptions, you will get a dump of the job (as if DSPJOB * *PRINT was specified).

Example of a Default Program

The following program is a sample default program that could be used when a diagnostic message is sent followed by an escape message.

```
             PGM PARM(&PGMNAME &MRK)
             DCL VAR(&PGMNAME) TYPE(*CHAR) LEN(10)
             DCL VAR(&MRK) TYPE(*CHAR) LEN(4)
             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)
GETNEXTMSG:  CHGVAR VAR(&SAVEMRK) VALUE(&DIAGMRK)
             RCVMSG PGMQ(*SAME &PGMNAME) MSGTYPE(*DIAG) +
                    RMV(*NO) KEYVAR(&DIAGMRK)
             IF  (&DIAGMRK *NE &BLANKMRK) THEN(GOTO GETNEXTMSG)
             ELSE
              DO
               RCVMSG      PGMQ(*PRV *) MSGTYPE(*DIAG) +
                             MSGKEY(&SAVEMRK) RMV(*NO) +
                             MSGDTA(&MSGDTA) MSGID(&MSGID) MSGF(&MSGF) +
                             MSGFLIB(&MSGFLIB)
               SNDPGMMSG   MSGID(&MSGID) MSGF(&MSGLIB/&MSGF) +
                             MSGDTA(&MSGDTA) TOPGMQ(*PRV &PGMNAME) +
                             MSGTYPE(*ESCAPE)
             ENDDO
            ENDPGM
```

The program receives all the diagnostic messages until it receives the diagnostic message that was issued prior to the escape message. Then it sends that diagnostic message as an escape message to allow the previous program to monitor for it.

Specifying the Alert Option

On the ADDMSGD command, you can specify an alert option to allow an alert to be created for a message. A message, for which an alert can be created, can cause an SNA alert to be created and sent to a problem management focal point. The alert created for a message can be defined using the Add Alert Description (ADDALRD) command. For more information about the OS/400 alerts support, you can refer to the *Communications and Systems Management User's Guide*.

Example of Describing a Message

In the following example, the ADDMSGD command creates a message to be used in applications such as order entry. The message is issued when a customer number entered on the display is not found. The message is:

```
Customer number &1 not found
```

The ADDMSGD command for this message is:

```
ADDMSGD   MSGID(USR4310) +
          MSGF(QGPL/USRMSG)  +
          MSG('Customer number &1 not found') +
          SECLVL('Change customer number') +
          SEV(40) +
          FMT((*CHAR 8))
```

The message is added to the USRMSG file in the library QGPL.

You can use the DSPMSGD or WRKMSGD command to print or display message descriptions.

Defining Double-Byte Messages

To define a message with double-byte text, write a CL program using the ADDMSGD command. The defined message is put into a message file and then sent normally. When writing the program, do the following:

1. Make sure the source file containing the program is a double-byte file. Specify IGCDTA(*YES) on the Create Source Physical File (CRTSRCPF) command.
2. Use the source entry utility (SEU) to enter the program. CL commands using double-byte characters can only be entered through SEU. For this reason, double-byte messages must be created in a CL program.
3. Limit the length of the message to 37 double-byte characters, so the complete message can be displayed or printed.

   When using the MONMSG command, also limit the Comparison Data (CMPDATA) parameter to 6 double-byte characters.
4. If the double-byte message file replaces an alphanumeric message file (such as files of translated messages to be sent only to double-byte display stations), enter a command similar to the following to override the alphanumeric message file:

   ```
   OVRMSGF MSGF(QCPFMSG) TOMSGF(DBCSLIB/QCPFMSG)
   ```

   Double-byte messages can be displayed only at double-byte display stations.

Overriding Message Files

You can override message files used in a program. The creation (Override Message File command), deletion (Delete Override command), and display (Display Override command) of message file overrides is similar to other types of overrides (see the *Data Management Guide*). Here, however, only the name of the message file, not the attributes, is overridden, and the rules for applying the overrides are slightly different.

To override a message file, use the Override Message File (OVRMSGF) command. The file overridden is specified in the MSGF parameter; the file overriding it is specified in the TOMSGF parameter.

For example, to override QCPFMSG with a user message file named USRMSGF, the following command would be used:

```
OVRMSGF MSGF(QCPFMSG) TOMSGF(USRMSGF)
```

When a predefined message is retrieved or displayed, the overriding file is searched for a message description. If the message description is not found in that file, the overridden file is searched.

When you use message file overrides, the following occurs when a predefined message is sent, received, or displayed:

- Predefined messages sent to program message queues reflect the message file overrides in effect at the time the message is sent.
- Predefined messages sent to a message queue other than a program message queue reflect the message file overrides in effect at the time the message is received or displayed.

OS/400 issues escape message CPF2469 if any of the following occurs while sending (SNDPGMMSG command, SNDMSG command, SNDBRKMSG command, SNDUSRMSG command, or SNDRPY command) a message:

- The receiver is not authorized to the library in which the message file is located.
- The message file is not found.
- The receiver is not authorized to the message file.
- The message file is damaged.
- OS/400 is unable to allocate the library containing the message file.

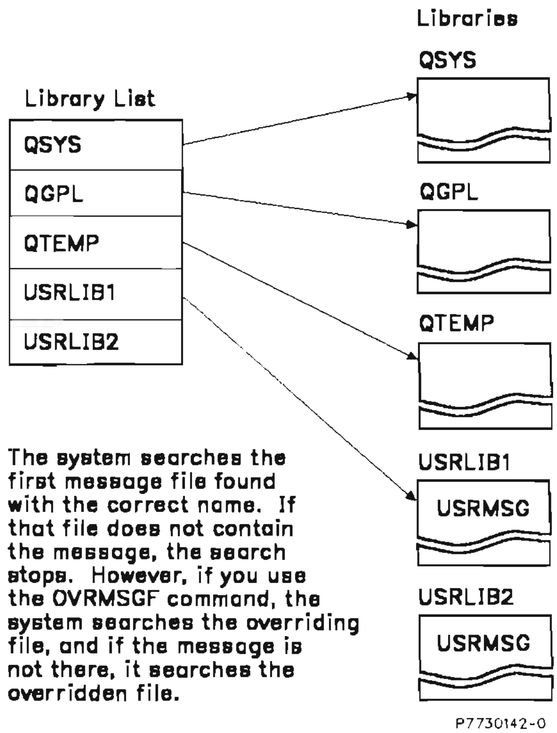
OS/400 attempts to retrieve the message from the message file found at the time the message was sent to that message queue.

There are four 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 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:



Example of Overriding a Message File

Assume that you want to change an IBM-supplied message for use in a job. For example, suppose you want to change message CPC2191, which says:

```
Object XXX in YYY type *ZZZ deleted
```

to say:

```
Object XXX in YYY deleted
```

First, you create a message file:

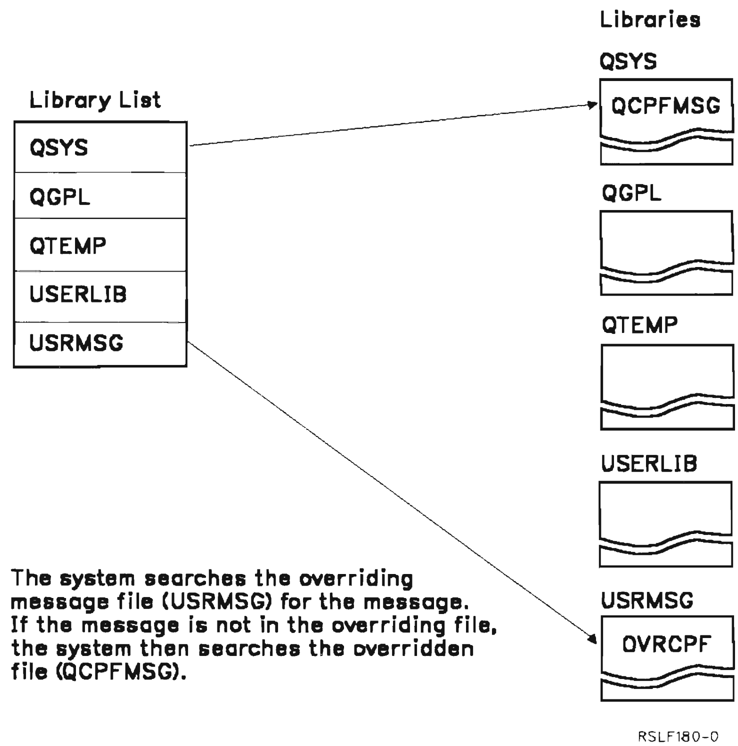
```
CRTMSGF MSGF(USRMSG/OVRCPF)
```

Then you use the message CPC2191 as a basis for your message and add it to the message file:

```
ADDMSGD  MSGID(CPC2191) MSGF(USRMSG/OVRCPF) +
         MSG('Object &1 in &2 deleted') +
         SEV(00) FMT((*CHAR 10) (*CHAR 10))
```

You then use the OVRMSGF command to override the message file when you run the job:

```
OVRMSGF MSGF(QCPFMSG) TOMSGF(USRMSG/OVRCPF)
```



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 CPF 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 that can be run at any time.

You can also override overriding files. For example, you can specify the following OVRMSGF commands during a job.

```
OVRMSGF  MSGF(MSGFILE1)  TOMSGF(MSGFILE2)
OVRMSGF  MSGF(MSGFILE2)  TOMSGF(MSGFILE3)
```

First, file MSGFILE1 was overridden with MSGFILE2. Second, MSGFILE2 was overridden with MSGFILE3. When a message is sent, the files are searched in this order:

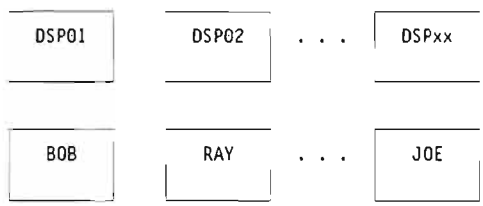
1. MSGFILE3
2. MSGFILE2
3. MSGFILE1

You can prevent message files from being overridden. To do so, you must specify the SECURE parameter on the OVRMSGF command.

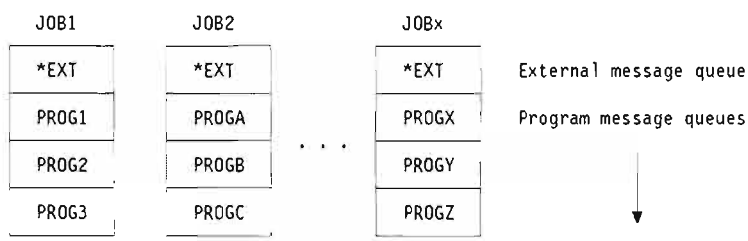
Types of Message Queues

All messages on the system are sent to a message queue. The system user or program associated with the message queue receives the message from the queue. Similarly, a reply to a message is sent back to the message queue of the user or program requesting the reply.

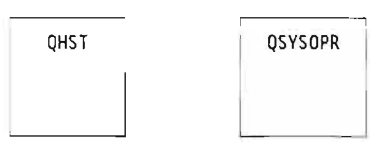
The following diagrams show the message queues supplied by IBM. A message queue is supplied for each display station (where DSP01 and DSP02 are display station names) and each user profile (where BOB and RAY are user profile names):



Job message queues are supplied for each job running on the system. Each job is given an external message queue (*EXT) and each call of a program within the job has its own message queue, with the same name as the program:



Message queues are also supplied for the system history log (QHST) and the system operator (QSYSOPR):



These message queues are used as follows:

- Display station message queues are used for sending and receiving messages between display station users and between display station users and the system operator. The name of the queue is the same as the name of the display station. The queue is created by the system when the display station 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 program

message queues. See "Job Message Queues" on page 7-23 for more information.

- System operator message queue (QSYSOPR) is used for receiving and replying to messages from the system, display station users, and application programs.
- The history log message queue is used for sending information to the history log (QHST) from any job in the system.

In addition to these message queues, you can create your own user message queues for sending messages to system users and between application programs.

Creating or Changing a Message Queue

To create your own user message queues, you use the Create Message Queue (CRTMSGQ) command. In addition, you also use the Change Message Queue (CHGMSGQ) command to change the following attributes of your message queue.

The attributes of a message queue are:

- The maximum amount of storage that can be used for the message queue. The same formula that was used to determine the size of a message file can be used for a message queue (see "Creating a Message File" on page 7-3). If an IBM-supplied or user-created message queue has been extended, it is reset to its original allocated size when the message queue is cleared.
- Whether changes to the message queue must be written immediately to the disk. Writing the changes immediately to the disk ensures that no messages are lost in cases like a system failure. Note that this will cause a decrease in system performance.
- The method of delivery for messages arriving at a message queue. When a message queue is created, the method of delivery is defined as hold delivery. When a display station is signed on, the user's message queue is set to the mode specified in the user profile. The types of delivery you can specify on the CHGMSGQ command are:
  - Break delivery. A job is interrupted and a program is called to deliver the message. If a user program is not specified on the CHGMSGQ command that requests break delivery, or if *SAME is specified, the Display Message (DSPMSG) command automatically displays the message. Break messages for a job can be controlled with the BRKMSG parameter on the CHGJOB command.
  - Notify delivery. 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 command.
  - Hold delivery. The message queue holds the messages until the display station user requests them with the DSPMSG command.
  - Default delivery. All messages are ignored, and any messages requiring a reply are sent the default reply for the message.
- How to handle messages for break delivery.

- Automatically run the DSPMSG command. For an interactive job, the messages are displayed at the display station if the severity code is high enough. For a batch job, the messages are listed to a spooled printer file if the severity code is high enough.
- Call a program to handle the messages. You must use the CHGMSGQ command to specify the program to be called and to set the method of delivery to break mode.

- 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, all messages sent to the queue are displayed even if their severity code is less than the severity code specified on the CHGMSGQ command.

  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.

Note: When a device is described to the system, the system establishes a message queue for the device to receive all action messages for the device. For work station 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 work station message queues and the QSYSOPR message queue) cannot have their delivery mode changed by a job when the message queue is in break or notify delivery mode for a different job.

When you sign off the display station, or the job ends unexpectedly, the user message queue delivery mode is changed to hold mode, if the delivery mode of the user message queue is break or notify for this job. The user message queue delivery mode is also changed from break or notify mode to hold mode when you transfer to an alternative job. You can do this using the Transfer Secondary Job (TFRSECJOB) command or by pressing the System Request key and specifying option 1 on the System Request menu.

After transferring to an alternative job, you sign on using your user profile. Your user message queue is put into the delivery mode specified in your user profile. This allows the user message queue to transfer to the alternative job. You are then able to transfer back and forth between these two jobs and have your user message queue follow you.

However, if after transferring to an alternative job, you sign on using a user profile other than your own, the user message queue for the job from which you transferred is left in hold delivery mode. The user message queue for the user profile you signed on with is put in the delivery mode specified in that user profile. Because of this, your user message queue could be put into break or notify delivery mode by another user. If another user still has your user message queue in that delivery mode when you transfer back to the first job, your user message queue delivery mode cannot be changed back to the original delivery mode.

The QSYSOPR message queue is the message queue for the system operator, unless it has been changed. The above situation can occur for a system operator as well.

Message Handling Program

A message handling program is called whenever a message of high enough severity arrives on the message queue. To request a message handling program, you must specify the name of the program and break delivery on the same CHGMSGQ command. The user-defined program called to handle messages for break delivery is passed parameters so it can receive and handle the message. The parameters are:

- The name of the message queue that received the message. The message queue name is a 10-character field and is left-justified and padded with blanks to the right, if necessary.
- The name of the library that contains the message queue. The library name is a 10-character field and is left-justified and padded with blanks to the right, if necessary.
- The message reference key of the received message. The key is a 4-character field.

If the message handling program is called, it interrupts the job in which the message occurred and runs. When the message handling program ends, the original program resumes processing. The message handling program must receive the message with the Receive Message (RCVMSG) command so the message is marked as handled and the program is not called again. An example of a message 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 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
```

Warning: In the above example break message handling program, if a CPA5316 message should arrive at the queue while the DSPMSG command is running, the DSPMSG display will show the original message that caused the break and the CPA5316 message and will wait for the operator to reply to the CPA5316 message before proceeding. In other words, the program will never have a chance to automatically reply to the CPA5316 message under certain circumstances.

Note: This program cannot open a display file if the interrupted program is waiting for input data from the device display.

You can use the system reply list to specify that the system issue the reply to specified predefined inquiry messages so that the display station user does not need to reply. See “Using the System Reply List” on page 8-30 for more information.

Example of Changing the Delivery Mode

When the system is started, it puts the QSYSOPR message queue in break delivery when the controlling subsystem is started. However, if the system operator signs off, the message queue is put in hold delivery. When the system operator signs on again, QSYSOPR is placed in the mode specified in the QSYSOPR user profile.

The following program can be used to place the QSYSOPR message queue in break mode. Similar programs can be used as initial programs to monitor message queues other than the one specified in a user's own user profile.

```
PGM /* Initial program 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 program 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 ends, the initial menu is displayed. A severity level of 50 is used to decrease the number of break messages that interrupts the work station user. A common reason for failure is when another user has QSYSOPR in break mode already.

Job Message Queues

Job message queues are created for each job on the system to handle all the message requirements of the job. Job message queues for a single job consist of an external message queue (*EXT) and a set of program message queues.

The external message queue (*EXT) is used to communicate with the external requester (such as a display station user) of the job. Messages (except status messages) sent to the external message queue of a job are also written to the job log for printing at end of job (see "Job Log" on page 8-33 for more information).

If an informational, inquiry, or notify message is sent to the external message queue for an interactive job, the message is displayed on the Display Program Messages display, and the program waits for a reply to inquiry or notify messages from the display station user. If the user does not enter a reply and presses 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 an inquiry or notify message is sent to the external message queue for a batch job, the default reply is sent to the sender of the message. If there is no default message, *N is the reply. The system reply list may override the displaying of inquiries or the sending of default replies to inquiries to *EXT.

If a status message is sent to the external message queue of an interactive job, the message is displayed on the message line of the display station. You can use status messages like this to inform the display station user of the progress of a long-running operation. For example, the system sends status messages when running the CPYF command if you copy a file with several members.

Note: When your application completes the long-running operation, you must send another message to clear the message line at the display. You can use message CPI9801, which is a blank message, for this purpose. For example:

```
PGM
.
.
.
SNDPGMMSG  MSGID(CPF9898)  MSGF(QCPFMSG)  MSGDTA('Status 1')  +
           TOPGMQ(*EXT)  MSGTYPE(*STATUS)
.
.
.
SNDPGMMSG  MSGID(CPF9898)  MSGF(QCPFMSG)  MSGDTA('Status 2')  +
           TOPGMQ(*EXT)  MSGTYPE(*STATUS)
.
.
.
SNDPGMMSG  MSGID(CPI9801)  MSGF(QCPFMSG)  TOPGMQ(*EXT)  +
           MSGTYPE(*STATUS)
.
.
.
ENDPGM
```

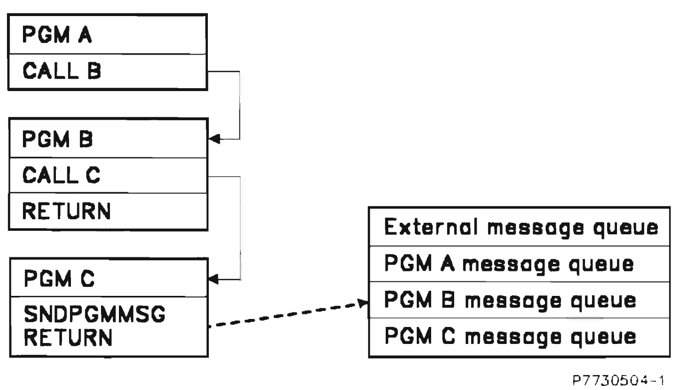
Program message queues are used to send messages between program calls of a job. These messages can be informational, request, completion, diagnostic, status, escape, or notify messages.

A program message queue is created for each call of a program that has messages sent to it and is given the same name as the program. However, if a program is called more than once, only the program message queue of the most current call can be used for message handling. That is, when program A sends a message to program B, the message goes to the program message queue for the last call of program B.

You can send a message to the calling program of a program without specifying the program name. The calling program is always the previous program in the program stack. In addition, you can send a message to the calling program of a specified program; in which case, you must specify the program name.

A program message queue for a program is no longer available for use when the program ends. Because the program message queue is no longer available, a program higher in the program stack cannot access the messages sent to a lower level program's message queue once the lower level program has ended.

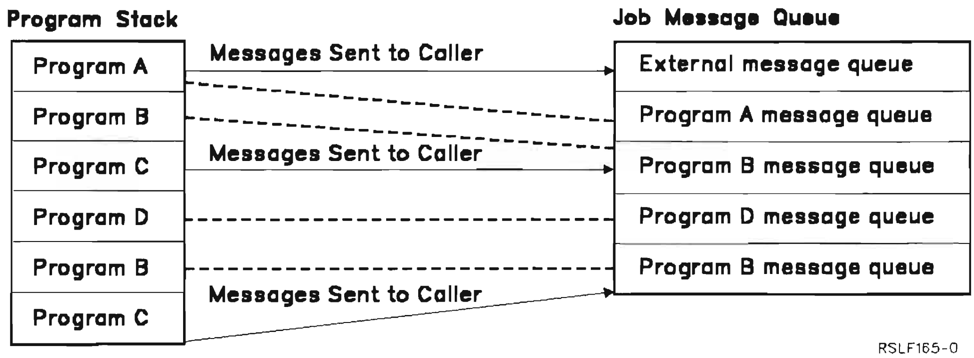
For example, as shown in the following figure, assume program A calls program B which calls program C. Program C sends a message to program B and ends. The message is available to program B. However, when program B ends, its program message queue is no longer available and cannot be accessed by program A, even though the message is shown in the job log.



If program A needs to access the specific messages, you could:

- Have program C send specific messages to program A
- Have program B resend the messages to program A

The following figure shows the relationship of program calls, the job message queue, and the program message queues. A dashed line (-----) indicates which message queue is associated with which call of a program.



In the preceding figure, program B has two program message queues, one for each call of the program. There are no message queues for program C because no messages were sent to program C. When program C sends a message to program B, the message goes to the program message queue for the last call of program B.

Note: When you are using the command entry display, you can display all the messages sent to the job message queue by pressing F10 (Include detailed messages). Once the messages are displayed, you can roll through them using one of the roll keys.

You can also display the messages for a job by using the Display Job Log (DSPJOBLOG) command.

Chapter 8. Working with Messages

This chapter discusses some of the ways that messages can be used to communicate between users and programs. Messages can be sent:

- From one system user to another system user, even if the receiver of the messages is not currently using the system
- From one program to another program
- From a program to a system user, even if the receiver of the messages is not currently using the system

Interactive system users can send only immediate messages and replies.

Programs can send immediate messages or predefined messages with user-defined data. In addition, programs can:

- Receive messages
- Retrieve a message from a message file and place it into a program variable
- Remove messages from a message queue
- Monitor for messages

See the Handling Messages section in the *System Operations: Operator's Guide*, SC21-8082 for more information on handling messages.

Sending Messages to a System User

Several commands can be used to send messages to system users:

- Send Message (SNDMSG)
- Send Break Message (SNDBRKMSG)
- Send Program Message (SNDPGMMSG)
- Send User Message (SNDUSRMSG)

SNDPGMMSG and SNDUSRMSG can only be used in batch or interactive programs. These commands cannot be entered for interactive jobs. The SNDMSG command sends an informational or inquiry message to the system operator message queue (QSYSOPR), a display station message queue, or a user message queue. You can send an informational message to more than one message queue at a time. But you can send an inquiry message to only one message queue at a time. The message is delivered by the delivery type specified for the message queue. The message does not interrupt the user unless the message queue is in break mode.

The following SNDMSG command is sent by a display station user to the system operator:

```
SNDMSG   MSG('Mount tape on device TAP1') TOUSR(*SYSOPR)
```

The SNDBRKMSG command sends an immediate message from a work station, a program, or a job to one or more display stations to be delivered in the break mode regardless of what delivery mode the receiver's message queue is set to. This command can be used to send a message only to display station message queues. You should use the SNDBRKMSG command when sending any message that requires immediate attention of a display station user.

If you send an inquiry message, you can specify that the reply be sent to a message queue other than that of your display station.

The following SNDBRKMSG command is sent by the system operator to all the display station message queues:

```
SNDBRKMSG MSG('System going down in 15 minutes')
          TOMSGQ(*ALLWS)
```

Sending Messages from a CL Program

Use the Send Program Message (SNDPGMMSG) command or the Send User Message (SNDUSRMSG) command to send a message from a CL 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 program to the following types of queues:

- External message queue of the requester of the job (see "Job Message Queues" on page 7-23)
- Program message queue of a program called by the job (see "Job Message Queues" on page 7-23)
- System operator message queue
- Work station message queue
- User message queue

To send a message from a program, you can specify the following on the SNDPGMMSG command:

- Message identifier or an immediate message. The message identifier is the name of the message description for a predefined message.
- Message file. The name of the message file containing the message description when a predefined message is sent.
- Message data fields. If a predefined message is sent, these fields contain the values for the substitution variables in the message. The format of each field must be described in the message description. If an immediate message is sent, there are no message data fields.
- Message queue or user to receive the message.
- Message type. The following indicates which types of messages can be sent to which types of queues (V = valid).

Table 8-1. Valid Message Types for Message Queue Types

Message Type	Message Queue Type: External	Program	QSYSOPR	Work Station	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			

- 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 program message queue of the program that sent the inquiry message.
- Key variable name. The name of the CL variable to contain the message reference key for a message.

To send the message created in "Example of Describing a Message" on page 7-14, 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, in your program you declare a program variable for the customer number &CUSNO and 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 program, so you cannot specify the exact display station message queue that the message is to be sent to (TOPGMQ parameter); therefore, you specify the external message queue *EXT.

Inquiry and Informational Messages

Using the SNDUSRMSG command, you can send an inquiry message or an informational message to a display station user, the system operator, or a user-defined message queue. If you use the SNDUSRMSG command to send an inquiry message to the user, the 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 program using the SNDUSRMSG command, you can specify the following on the SNDUSRMSG command:

- Message identifier or an immediate message. The message identifier is the name of the message description for a predefined message.
- Message file. The name of the message file containing the message description when a predefined message is sent.
- Message data fields. If a predefined message is sent, these fields contain the value for the substitution variables in the message. The format of each field must be described in the message description. If an immediate message is sent, there are no message data fields.
- Valid replies to an inquiry message.
- Default reply value to an inquiry message.
- Message type.
- Message queue to which the message is to be sent.
- Message reply. A CL variable, if any, that is to contain the reply received in response to an inquiry message.
- Translation table. The translation table to be used, if any, to translate the reply value. This is normally used for translating lowercase to uppercase.

Completion and Diagnostic Messages

Using the SNDPGMMSG command, you can send diagnostic and completion messages. From your CL program, you can send these message types to any message queue. Diagnostic messages tell the calling program about errors detected by the program. Completion messages tell the results of work done by the program.

Normally, an escape message is sent to the message queue of the calling program to tell the calling program what the problem was or that diagnostic messages were also sent. For a completion message, an escape message is usually not sent because the requested function was performed.

For an example of sending a completion message, assume that the system operator uses the system operator menu to call a CL program SAVPAY to save certain objects. The CL program saves the objects and issues a completion message:

```
PGM
SAVOBJ OBJ(PAY1 PAY2) LIB(PAYROLL) CLEAR(*YES)
SNDPGMMSG MSG('Payroll objects have been saved') MSGTYPE(*COMP)
ENDPGM
```

If the SAVOBJ command fails, the CL program function checks and the system operator has to display the detailed messages to locate the specific escape message explaining the reason for the failure as described later in this chapter. If the SAVOBJ command completes successfully, the completion message is sent to the message queue associated with the program that displays the system operator menu.

Status Messages

You can send status messages from your CL program, using the SNDPGMMSG command, to its caller's program message queue or to the external message queue (*EXT) for the job. When a status message is sent to a program message queue, the receiving program can monitor for the arrival of the status message and can handle the condition it describes. If the receiving program does not monitor for the message, control returns to the sender to resume processing. See "Monitoring for Messages in a CL Program" on page 8-16.

Escape and Notify Messages

Using the SNDPGMMSG command, you can send escape messages from your CL program to the message queue of the calling program. An escape message tells the calling program that the program ended abnormally and why. The calling program can monitor for the arrival of the escape message and handle the condition it describes. Control does not return to the sender of an escape message, and the sending program immediately ends. If the calling program does not monitor for an escape message, default system action is taken. See "Monitoring for Messages in a CL Program" on page 8-16 for more information about using escape messages.

You can send notify messages from your CL program to the message queue of the calling program or to the external message queue. A notify message tells the calling program about a condition under which processing can continue. The calling program can monitor for the arrival of the notify message and handle the condition it describes. This ends processing of the sender. If the calling program is not monitoring for the message, a default reply is returned to the sender, who resumes processing. See "Monitoring for Messages in a CL Program" on page 8-16.

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:

```
SNDUSRMSG MSGID(CPF9898) MSGF(QCPFMSG) MSGDTA('Error condition') +
          MSGTYPE(*ESCAPE)
```

Examples of Sending Messages

Example 1: The following CL program allows the display station user to submit a job by calling this program instead of entering the Submit Job (SBMJOB) command. The program sends a completion message when the job has been submitted.

```
PGM
SBMJOB JOB(WKLYPAY) JOBD(USERA) RQSDTA('CALL WKLY PARM(PAY1)')
SNDPGMMSG MSG('WKLPAY job submitted') MSGTYPE(*COMP)
ENDPGM
```

Example 2: The following CL program changes a message based on a parameter received from a program and sends the message as a completion message. (The ERRCNT field is defined as characters in PGMA.)

```
PGM
DCL &ERRCNT TYPE(*CHAR) LEN(3)
CALL PGMA PARM(&ERRCNT)
SNDPGMMSG MSG('PGMA completed' *BCAT &ERRCNT *BCAT 'error') +
          MSGTYPE(*COMP)
ENDPGM
```

Example 3: The following program 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 program does not use the reply value.

```
PGM
DCL &MSGKEY TYPE(*CHAR) LEN(4)
SNDPGMMSG   MSG('Load special form') TOUSR(*SYSOPR) +
            KEYVAR(&MSGKEY) MSGTYPE(*INQ)
RCVMSG MSGTYPE(*RPY) MSGKEY(&MSGKEY) WAIT(120)
.
.
.
ENDPGM
```

The WAIT parameter must be specified on the RCVMSG command so that the program waits for the reply. If the WAIT parameter is not specified, the program continues with the instruction following the RCVMSG command, without receiving the reply. The MSGKEY parameter is used in the RCVMSG command to allow the program to receive the reply to a specific message. The variable &MSGKEY in the SNDPGMMSG command is returned to the program for use in the RCVMSG command.

Example 4: The following program 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 program accepts either an uppercase or lowercase Y or N. (The lowercase values are translated to uppercase by the translation table (TRNTBL parameter) to make program logic easier.) If the value entered is not one of these four, the operator is issued a message indicating the reply is not valid.

```
PGM
DCL &REPLY *CHAR LEN(1)
.
.
SNDUSRMSG MSG('Update YTD Information Y or N') VALUES(Y N) +
     MSGRPY(&REPLY)
IF (&REPLY *EQ Y)
 DO
 .
 .
 .
 ENDDO
ELSE
 DO
 .
 .
 ENDDO
 .
 .
 .
ENDPGM
```

Example 5: The following program uses the message CPF9898 to send an escape message. The text of the message is 'Program 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('Program detected failure')
   .
   .
   ENDPGM
```

Example 6: The following program allows the system operator to send a message to several display stations. When the system operator calls the program, the program displays a prompt on which the system operator can enter the type of message to be sent and the text for the message. The program concatenates the date, time, and name of the message.

```
          PGM
          DCLF WSMSGD
          DCL &MSG TYPE(*CHAR) LEN(150)
          DCL &HOUR TYPE(*CHAR) LEN(2)
          DCL &MINUTE TYPE(*CHAR) LEN(2)
          DCL &MONTH TYPE(*CHAR) LEN(2)
          DCL &DAY TYPE(*CHAR) LEN(2)
          DCL &WORKHR TYPE(*DEC) LEN(2 0)
          SNDRCVF RCDFMT(PROMPT)
          IF &IN91 RETURN /* Request was ended  */
          RTVSYSVAL QMONTH RTNVAR(&MONTH)
          RTVSYSVAL QDAY RTNVAR(&DAY)
          RTVSYSVAL QHOUR RTNVAR(&HOUR)
          IF (&HOUR *GT '12') DO
          CHGVAR &WORKHR &HOUR
          CHGVAR &WORKHR (&WORKHR - 12)
          CHGVAR &HOUR &WORKHR /* Change from military time */
          ENDDO
          RTVSYSVAL QMINUTE RTNVAR(&MINUTE)
          CHGVAR    &MSG ('From Sys Opr ' *CAT &MONTH *CAT '/' +
                    *CAT &DAY +
                    *BCAT &HOUR *CAT ':' *CAT &MINUTE +
                    *BCAT &TEXT)
          IF (&TYPE *EQ 'B') GOTO BREAK
NORMAL:   SNDPGMMSG MSG(&MSG) TOMSGQ(WS1 WS2 WS3)
          GOTO ENDMSG
BREAK:    SNDBRKMSG MSG(&MSG) TOMSGQ(WS1 WS2 WS3)
ENDMSG:   SNDPGMMSG MSG('Message sent to display stations') +
                    MSGTYPE(*COMP)
          ENDPGM
```

The DDS for the display file, WSMSGD, used in this program follows:

```
     A                                      DSPSIZ(24 80)
     A          R PROMPT                    TEXT('Prompt')
     A                                      BLINK
     A                                      CA01(91 'Return')
     A                                  1  2'SEND MESSAGES TO WORK STATIONS'
     A                                      DSPATR(HI)
     A                                  3  2'Type'
     A            TYPE           1   I   +2VALUES('N' 'B')
     A                                      CHECK(ME)
     A                                      DSPATR(MDT)
     A                                    +3'(N - No break;  B - Break)'
     A                                  5  2'Text'
     A            TEXT         100   I   +2LOWER
```

RSLF174-2

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
```

Receiving Messages in a CL Program

For your program to receive messages from a message queue, you use the Receive Message (RCVMSG) command. Messages can be received in the following ways:

- By message type. You can specify that any type or specific types can be received (MSGTYPE parameter). For new messages (messages that have not been received in the program), the messages 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 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 program (DCL command). You must specify on the RCVMSG command the CL variable through which the key is to be passed (MSGKEY parameter).
  - Receive the next message on a message queue following the message with a specified message reference key. In addition to specifying the MSGKEY parameter, you must specify MSGTYPE(*NEXT).
  - Receive the message on a message queue that is before a message with a specified message reference key. In addition to specifying the MSGKEY parameter, you must specify MSGTYPE(*PRV).
- By its location on the message queue. You must specify MSGTYPE(*FIRST) for the first message on the message queue; specify MSGTYPE(*LAST) for the last.
- By both message type and message key (MSGTYPE and MSGKEY parameters).

To receive a message, you can specify:

- Message queue. Where the message is to be received from.
- Message type. Either a specific message type can be specified or all types can be specified.
- Whether to wait for the arrival of a message. After the wait is over and no message is received, the CL variables requested to be returned are filled with blanks (or zeros if numeric) and control returns to the program running the RCVMSG command.
- Whether to remove the message from the message queue after it is received. If it is not removed, it becomes an old message on the message queue and can only be received again (by a program) through its message

reference key. However, if messages on the message queue are reset to new messages through the CHGMSGQ command, you do not have to use the message reference key to receive the message. Note that inquiry messages that have already been replied to are not reset to a new status. (See "Removing Messages from a Message Queue" on page 8-15 *for more information.*)

- A group of CL variables into which the following information is placed (each corresponds to one variable):
  - Message reference key of the message in the message queue (character variable, 4 characters)
  - Message (character variable, length varies)
  - Length of message, including length of substitution variable data (decimal variable, 5 decimal positions)
  - Message help text (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 must be 80 characters)
  - Type of message received (character variable, 2 characters long)
  - Alert option of the message received (character variable, 9 characters)
  - Message file that contains the predefined message (character variable, 10 characters)
  - Message file library name that contains the message file used to receive the message (character variable, 10 characters)
  - Message file library name that contains the message file used to send the message (character variable, 10 characters)

The following RCVMSG command specifies that any new message on the inventory application message queue INVN is to be received:

```
RCVMSG MSGQ(QGPL/INVN) MSGTYPE(*ANY) MSG(&MSG)
```

The message received is placed in the variable &MSG. *ANY is the default value on the MSGTYPE parameter.

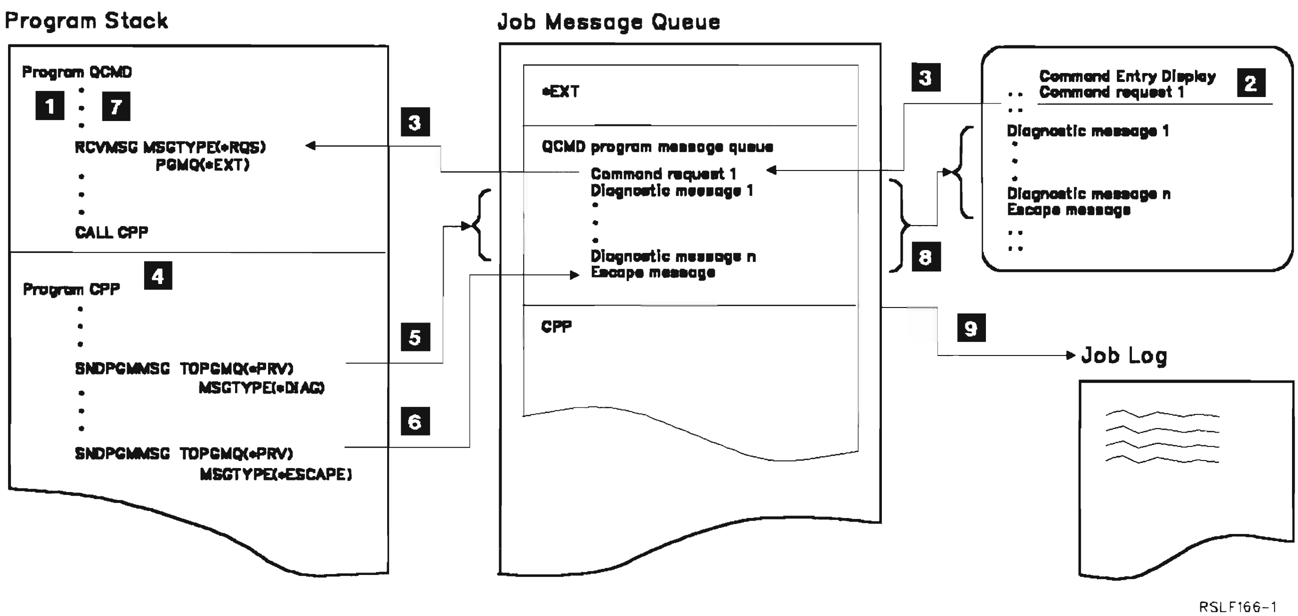
Request Messages

Receiving request messages is a simple method for your CL program to process CL commands. For example, your program can obtain input from a display station and handle the messages that result from the analysis and processing of the program. Usually, request messages are received from the external message queue (*EXT) of the job. For batch jobs, the requests received are those read from the input stream. For interactive jobs, the requests received are those the display station user enters one at a time on the Command Entry display. For example, CL commands are requests that are received by the IBM-supplied CL processor.

Your 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 program message queue for the program. The 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 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 program to indicate this.

More than one program 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.

The following diagram shows how request messages are processed by QCMD:



1 The CL processor QCMD receives a request message from *EXT.

2 If there is no request message on *EXT, the Command Entry display is displayed. The display station user enters a command on the display. When the command is entered, it is placed on *EXT as a request message.

3 The command is then moved to the end of the QCMD program message queue and is passed from there to QCMD.

4 The command is analyzed and its command processing program (CPP) is called.

5 The command processing program sends diagnostic messages to the program message queue for QCMD.

6 Then the command processing program sends an escape message to the program message queue for QCMD. The escape message notifies QCMD that diagnostic messages are on the queue and that QCMD should end processing of the CPP.

7 QCMD is monitoring for the arrival of a request-check (CPF9901) or function-check (CPF9999) escape message. QCMD then tries to receive the next request message. If a request processor receives message CPF9901 or CPF9999, it should run a Reclaim Resources (RCLRSC) command. The

request processor should also monitor for messages CPF1907 (end request) and CPF2415 (which indicates that the user pressed F3 or F12 on the Command Entry display).

8 Because a request message was being processed, all the messages on the program 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 written to the job log by the message logging level specified for the job. For more information, see "Message Logging" on page 8-33.

Writing Request-Processing Programs

To become a request-processing program, your program must include the Send Program Message (SNDPGMMSG) and Receive Message (RCVMSG) commands. For example, the following commands would allow a program to become a request processor (request-processing program):

```
SNDPGMMSG  MSG('Request Message') TOPGMQ(*EXT) MSGTYPE(*RQS)
RCVMSG     PGMQ(*EXT) MSGTYPE(*RQS) RMV(*NO)
```

RMV(*NO) must be specified on the RCVMSG command because the program is not a request processor if the request message is removed from the program's message queue.

The program becomes a request processor when it receives a request message. While the program is a request processor, other programs that are called can be ended using option 2 (End request) on the System Request menu. The request-processing program should include a monitor for message CPF1907 (MONMSG command) 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-processing program.

The program remains a request processor until the program ends (either normally or abnormally) or until an RMVMSG command is run to remove all the request messages from the request processor's program message queue. For example, the following command removes all request messages from the message queue and, therefore, ends request processing:

```
RMVMSG  CLEAR(*ALL)
```

Determining if a Request-Processing Program Exists

To determine if a job has a request-processing program, display the job's program 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 program stack, the program associated with the number is a request-processing program. In the following example, both QCMD and QTEVIREF are request-processing programs:

```
                              Display Program Stack
                                                          System:   S0000000
Job:   WS31          User:   QSECOFR        Number:   000173

Job status:   ACTIVE

Request
 Level  Program        Library        Statement      Instruction
        QCMD           QSYS                            01DC
   1    QCMD           QSYS                            016B
        QTECADTR       QSYS                            0001
   2    QTEVIREF       QSYS                            02BA

                                                                    Bottom
Press Enter to continue.

F3=Exit   F5=Refresh   F12=Cancel
```

The following is an example of a request-processing program:

```
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 program make it a request processor. The program remains a request processor until the RMVMSG command is run. A Monitor Message command is placed after the call to program PGMONE because an end request may be sent from PGMONE to the request-processing program. If monitoring were not used, a function check would occur for an end request. No message monitor is specified after the call to PGMTWO because the RMVMSG command ends request processing.

Note that if an end request is attempted when no request-processing program is called, an error message is issued and the end is not performed.

Retrieving Messages in a CL Program

You can use the Retrieve Message (RTVMSG) command to retrieve the text of a message from a message file into a program variable. This differs from RCVMSG in that RTVMSG operates on predefined message descriptions that have not been sent. By doing this, you can write the message to an output list or handle it in other ways. Besides specifying the message identifier and message file name, you can specify:

- Message data fields. The message data for the substitution variables.
- 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 text (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)

For example, the following command adds the message description for the message USR1001 to the message file USRMSG:

```
ADDMSGD   MSGID(USR1001) MSGF(QGPL/USRMSG) +
          MSG('File &1 not found in library &2') +
          SECLVL('Change file name or library name') +
          SEV(40) FMT((*CHAR 10) (*CHAR 10))
```

The following commands result in the substitution of the file name INVENT in the 10-character variable &FILE and the library name QGPL in the 10-character variable &LIB in the retrieved message USR1001.

```
DCL &FILE TYPE(*CHAR) LEN(10) VALUE(INVENT)
DCL &LIB TYPE(*CHAR) LEN(10) VALUE(QGPL)
DCL &A TYPE(*CHAR) LEN(20)
DCL &MSG TYPE(*CHAR) LEN(50)
CHGVAR VAR(&A) VALUE(&FILE||&LIB)
RTVMSG   MSGID(USR1001) MSGF(QGPL/USRMSG) +
         MSGDTA(&A) MSG(&MSG)
```

The data for &1 and &2 is contained in the program variable &A, in which the values of the program variables &FILE and &LIB have been concatenated. The following message is placed in the CL variable &MSG:

```
File INVENT not found in library QGPL
```

If the MSGDTA parameter is not used in the RTVMSG command, the following message is placed in the CL variable &MSG:

```
File not found in library
```

After the message is placed in the variable &MSG, you could do the following:

- Send the message using the SNDPGMMSG command
- Use the variable as the text for a message line in DDS (M in position 38)
- Use a message subfile
- Print or display the message

Removing Messages from a Message Queue

Messages are held on a message queue until they are removed by a Remove Message (RMVMSG) command, Clear Message Queue (CLRMSGQ) command, the RMV parameter on the Receive Message (RCVMSG) and Send Reply (SNDRPY) commands, the remove function keys of the Display Messages display, or the clear message queue option on the Work with Message Queue display. You can remove:

- A single message
- All messages
- All except unanswered messages
- All old messages
- All new messages
- All messages from all inactive programs

To remove a single message using the RMVMSG command or a single old message using the RCVMSG command, you specify the message reference key of the message to be removed.

Note: The message reference key can also be used to receive a message and to reply to a message.

If you remove an inquiry message that you have not answered, a default reply is sent to the sender of the message and the inquiry message and the default reply are removed. If you remove an inquiry message that you have already answered, both the message and your reply are removed.

To remove all messages for all inactive programs from a user's job message queue, specify *ALLINACT for the PGMQ parameter and *ALL for the CLEAR parameter on the RMVMSG command. If you wish 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.

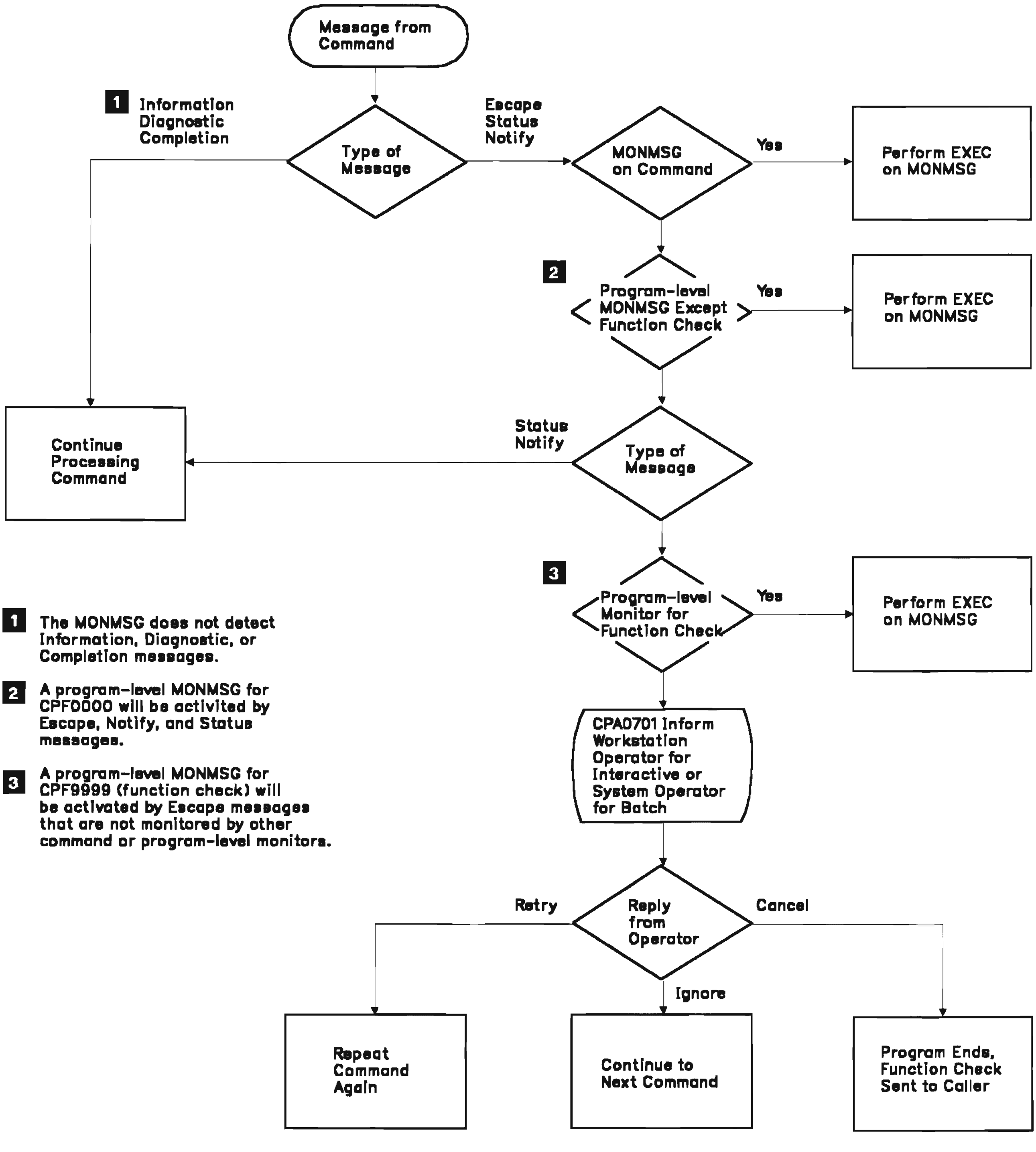
The following RMVMSG command removes a message from the user message queue JONES. The message reference key is in the CL variable &MRKEY.

```
DCL &MRKEY TYPE(*CHAR) LEN(4)
RCVMSG MSGQ(JONES) RMV(*NO) KEYVAR(&MRKEY)
RMVMSG MSGQ(JONES) MSGKEY(&MRKEY)
```

Note: The maximum number of messages that have never been handled on a message queue is 32 767 for each type of message sent; that is, 32 767 new diagnostic messages can be on the queue; 32 767 new request messages can be on the queue, and so on.

Monitoring for Messages in a CL Program

You can monitor for escape, notify, and status messages that are sent to your CL program's program message queue by the commands in your program. The Monitor Message (MONMSG) command monitors the messages sent to the program message queue for the conditions specified in the command. If the condition exists, the CL command specified on the MONMSG command is run. The logic involved with the MONMSG command is as follows:



The *CL Reference* contains a list of the escape, notify, and status messages that are issued for CL commands. You should also keep a list of all messages that you have defined.

Escape Messages: Escape messages are sent to tell your 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 program.

Status or Notify Messages: Status and notify messages are sent to tell your program of an abnormal condition that is not serious enough for the sender to end. By monitoring for status or notify messages, your program can detect this condition and not allow the function to continue.

You can monitor for messages using two levels of MONMSG commands:

- Program level: You can monitor for an escape, notify, or status message sent by any command in your program by specifying the MONMSG command immediately following the last declare command in your CL program. This is called a program-level MONMSG command. You can use as many as 100 program-level MONMSG commands in a program. (A CL 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 program by specifying the MONMSG command immediately following the command. This is called a command level MONMSG command. You can use as many as 100 command-level MONMSG commands for a single command. This lets you handle different escape messages in different ways.

To monitor for escape, status, or notify messages, you must specify, on the MONMSG command, generic message identifiers for the messages in one of the following ways:

- `pppmmnn`

  Monitors for a specific message. For example, MCH1211 is the message identifier of the zero divide escape message.
- `pppmm00`

  Monitors for any message with a generic message identifier that begins with a specific licensed program (`ppp`) and the digits specified by mm. For example, CPF5100 indicates that all notify, status, and escape messages beginning with CPF51 are monitored.
- `ppp0000`

  Monitors for every message with a generic message identifier that begins with a specific licensed program (`ppp`). For example, `CPF0000` indicates that all notify, status, and escape messages beginning with `CPF` are monitored.

  Note: Do not use `MONMSG CPF0000` when doing system function, such as install or saving or restoring your entire system, since you may lose important information.
- CPF9999

  Monitors for function check messages for all generic message identifiers. If an error message is not monitored, it becomes a CPF9999 (function check).

Note: Generally, when monitoring, your monitor also gets control when notify and status messages are sent.

In addition to monitoring for escape messages by message identifier, you can compare a character string, which you specify on the MONMSG command, to data sent in the message. For example, the following command monitors for an escape message (CPF5101) for the file MYFILE. The name of the file is sent as message data.

```
MONMSG MSGID(CPF5101) CMPDTA(MYFILE) EXEC(GOTO EOJ)
```

The compare data can be as long as 28 characters, and the comparison starts with the first character of the first field of the message data. If the compare data matches the message data, the action specified on the EXEC parameter is run.

The EXEC parameter on the MONMSG command specifies how an escape message is to be handled. Any command except PGM, ENDPGM, IF, ELSE, DCL, DCLF, ENDDO, and MONMSG can be specified on the EXEC parameter. You can specify a DO command on the EXEC parameter, in which case, the commands in the do group are run. When the command or do group (on the EXEC parameter) has been run, control returns to the command in your 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 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 program. When this happens, the value of &A is changed to 1 (as specified on the EXEC parameter).

In the following example, the program 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 program contains two CALL commands and a program-level MONMSG command for CPF0001. (This escape message occurs if a CALL command cannot be completed successfully.) If either CALL command fails, the program 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 program-level MONMSG command, any escape message that is handled by the MONMSG command is ignored. If the escape message occurs on any command except an IF command, the program continues processing with the command that would have been run next if the escape message had not occurred. If the escape message occurs on an IF command, the program continues processing as if the condition on the IF command were false. The following example illustrates what happens if an escape message occurs at different points in the program:

```
PGM
DCL &A TYPE(*DEC) LEN(5 0)
DCL &B TYPE(*DEC) LEN(5 0)
MONMSG MSGID(CPF0001 MCH1211)
CALL PGMA PARM(&A &B)
IF (&A/&B *EQ 5) THEN(CALL PGMB)
ELSE CALL PGMC
CALL PGMD
ENDPGM
```

Depending on where an escape message occurs, the following happens:

- If CPF0001 occurs on the call to PGMA, the program resumes processing on the IF command.
- If MCH1211 (divide by 0) occurs on the IF command, the IF condition is considered false, and the program resumes processing with the call to PGMC.
- If CPF0001 occurs on the call to PGMB or PGMC, the program resumes processing with the call to PGMD.
- If CPF0001 occurs on the call to PGMD, the program resumes processing with the ENDPGM command, which causes a return to the calling program.

You can also monitor simultaneously for the same escape message to be sent by a specific command in your program *and* by any 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.

Default Handling

Many escape messages can be sent to your program by commands and programs it calls. Probably, you will not want to monitor for and handle all of these. Normally, only a few escape messages pertain to the function of your program; the remaining escape messages should never actually be sent. The system provides default monitoring and handling of any messages you do not monitor.

Default handling assumes that an error has been detected in your program. If you are debugging the program, the message is sent to your display station. Then you can enter commands to analyze and correct the error. If you are not debugging the program, a dump is taken and the function-check escape message (CPF9999) is sent to your program. You can monitor for function-check escape messages so that you can either:

- Clean up and end the program
- Continue with some other aspect of your program

If you do not monitor for a function-check escape message, your program is ended and the function-check message is sent to the calling program of the program.

Notify Messages

Besides monitoring for escape messages, you can monitor for notify messages that are sent to your CL program's program message queue by the commands in your program or by the programs it calls. Notify messages are sent by these programs to tell your 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.

Monitoring for and handling notify messages are similar to monitoring for and handling escape messages. The difference is in what happens if you do not monitor for and handle notify messages. Unlike escape messages, unmonitored notify messages are not considered an indication of an error in your program. Instead, for notify messages, the default reply stored in the message description is used to tell the sender of the message what to do.

Status Messages

Another type of message you can monitor for is the status message. You can monitor for status messages that are sent by the commands in your CL program or by the programs your program calls. Status messages tell your program the status of the work performed by the sending program. By monitoring for status messages, you can prevent the sending program from proceeding with any more processing.

No message information is stored in a message queue for status messages. Therefore, if the status message is monitored for, the status message cannot be received. If the status message is not monitored for, no information is placed on the message queue and control is returned to the sending program for more processing. Status messages are often sent to communicate normal conditions that have been detected where processing can continue.

Status messages can also be used to send a message to the interactive user, informing the user of a function in progress. Status messages sent to the external message queue are shown on the interactive display. For example, the Copy File (CPYF) command sends a message informing the user that a copy is in progress.

Only messages contained in message files can be sent as status messages; immediate messages cannot be sent. You can use the system-supplied blank 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 program should remove the status message. The message cannot be removed using a command, but sending another status message with a blank message gives the appearance of removing the message. The system-supplied message ID CPI9801 can be used for this purpose. 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 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.

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. The name of the program and break delivery must both be specified on the same Change Message Queue (CHGMSGQ) command. The program must run a Receive Message (RCVMSG) command to receive the message.

The following program (PGMA) 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)
```

For more information, see "Message Handling Program" on page 7-21.

QSYSMSG Message Queue

The QSYSMSG message queue is an optional queue that you can create in the QSYS library. If it exists and is not damaged, certain messages are directed to it instead of, or in addition to, the QSYSOPR message queue. This allows a user-written program to gain control when certain messages are sent. You should not create the QSYSMSG queue unless you want it to receive specific messages.

Enter the following command to create the QSYSMSG queue:

```
CRTMSGQ  QSYS/QSYSMSG +
         TEXT('Optional MSGQ to receive specific system messages')
```

Once the QSYSMSG message queue is created, all of the specific messages (shown below in "Messages Sent to QSYSMSG Message Queue") are directed to it. You can write a program to receive messages for which you can perform special action and send other messages to the QSYSOPR message queue or another message queue. This program should be written as a break-handling program.

Messages Sent to QSYSMSG Message Queue

The following describes the specific messages sent to the QSYSMSG queue:

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 message data for CPF0907 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.

You can use the Work with System Status (WRKSYSSTS) command to monitor the amount of storage used. You can reduce user storage by doing the following:

1. Save objects by specifying STG(*FREE), and then delete any unused objects.
2. Save the old log versions of QHST that are not currently used and then delete them; either write or delete spooled files on the system.
3. If checksum protection is in effect, you can increase protected storage by reducing the amount of unprotected storage.

The system service tools function can be used to display and change the threshold value. For more information, see the *Backup and Recovery Guide*.

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. You may want to check two of the reason codes (704: password not valid; 705: user not authorized to the device) in your program for security reasons. The message data for this message contains the reason codes and other values (for example, unit of work identifier and device mode) which can be used to assist your program in taking action. For a detailed explanation of each reason code, see the message description for CPF1269 in the *Communications Programmer's Guide*.

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.

CPF1397: `Subsystem varied off work station.`

This message is sent if the threshold value assigned by the system value QMAXSIGN is exceeded 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

CPI0920 `Disk unit error occurred.`

This message is sent if a disk unit has detected errors and is requesting service. No data has been lost but a loss may occur if this condition is not corrected.

To correct the condition, you could consider one of the following:

- If you have a current copy of the existing system on tape, press F14 to run problem analysis.
- If you do not have a current copy of the existing system on tape, save the current copy to tape. For information on how to save a current copy of the system data, see the *Backup and Recovery Guide*.

CPI0945 `Disk mirroring protection is suspended on a disk unit.`

This message is sent if a disk unit has detected errors and is requesting service. An error occurred when the system was copying data from one disk device to another. Data has not been lost. To run problem analysis, press F14.

CPI0946 `Disk mirroring protection is suspended on a disk unit.`

This message is sent if a disk unit has detected errors and is requesting service. A disk unit did not complete an operation in the time allowed. Data has not been lost.

To report the hardware problem to your service representative on a magnetic disk unit, you can use the Analyze Problem (ANZPRB) command.

CPI0947 Disk mirroring protection is suspended on a disk unit.

The storage device controller was not able to complete an operation in the time allowed. Data has not been lost.

You can use the Analyze Problem (ANZPRB) command to report the problem on the disk unit to your service representative.

CPI0948 Disk unit is listed as missing.

The system is not able to locate a disk unit. Data has not been lost.

To locate the missing disk unit, you could consider the following:

- Use the system Resource Configuration List display to see the disk unit that was identified as missing.
- Ensure that the power cables on the disk unit are connected.
- Ensure that the power switches on the disk device are set to the On position.
- Ensure that the cables are connected between all devices on the storage device controller card, because one of these disk devices attached to the device controller card is listed as missing.
- Ensure that each device attached to the storage controller card, which has a device listed as missing, has a unique address.
- If the missing disk unit has a type of 9335, make sure the Enable/Disable switches on the front of the disk unit are set to the Enable position.

If you correct the problem using one of the steps listed above, do the following:

1. Type the Start System Service Tools (STRSST) command and press the Enter key.
2. Type the option to Work with disk units and press the Enter key.
3. Type the option to Work with disk unit recovery and press the Enter key.
4. Type the option to Suspend or resume mirroring protection and press the Enter key.
5. Type a 2 in the option field for the device with the specified address and press the Enter key. Mirroring protection on the disk unit then resumes.

If you cannot correct the problem, use the Analyze Problem (ANZPRB) command to report the problem to your service representative.

CPI0949 Disk mirroring protection is suspended on a disk unit.

This message is sent if a disk unit has detected errors and is requesting service. Data has not been lost. The disk mirror protection is suspended for one of the following reasons:

- The service representative is repairing the disk unit.
- A disk unit is not operating, but there is not enough information about the failure to run problem analysis.
- A disk unit found errors, but there is not enough information about the failure to run problem analysis.

If your service representative is repairing the disk unit, wait until the service representative is finished repairing the disk unit, and then do the following:

1. Type the Start System Service Tools (STRSST) command and press the Enter key.
2. Type the option to Work with disk units and press the Enter key.
3. Type the option to Work with disk unit recovery and press the Enter key.
4. Type the option to Suspend or resume mirroring protection and press the Enter key.
5. Type a 2 in the option field for the device with the specified address. Press the Enter key. Mirroring protection on the disk unit is then resumed.

Otherwise use the Analyze Problem (ANZPRB) command to report the problem to your service representative.

CPI0950 A disk unit is now available.

A disk unit, which was missing from the system configuration, is now available. Data has not been lost.

After the service representative is finished repairing the disk unit, do the following:

1. Type the Start System Service Tools (STRSST) command and press the Enter key.
2. Type the option to Work with disk units and press the Enter key.
3. Type the option to Work with disk unit recovery and press the Enter key.
4. Type the option to Suspend or resume mirroring protection and press the Enter key.
5. Type a 2 in the option field for the device with the specified address and press the Enter key. Mirroring protection on the disk unit will then be resumed.

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.

You can use the system service tools function to monitor the amount of storage used in the ASP. You could reduce the use of storage by saving objects and specifying STG(*FREE), and deleting any unused objects. For more information on the system service tools function, see the *Backup and Recovery Guide*.

CPI0954 ASP storage limit exceeded.

This message is sent if all available storage in the specified ASP has been used.

You can delete any objects that are not used in the user ASP. Use the system service tools function to verify that there is enough storage available in the ASP. If there is not enough storage, move data to another ASP or add disk storage to this ASP. Save any objects remaining on the ASP. Delete the objects on the ASP. Use the dedicated service tools function to request a reset of the overflow status. Data can then be restored to the ASP. For more information on the system service tools and dedicated service tools functions, see the *Backup and Recovery Guide*.

CPI0955 System ASP unprotected storage limit exceeded.

This message is sent if all available storage in the system ASP has been used.

You can do an initial program load (IPL) to release some unprotected storage. After the IPL, use the system service tools function to monitor the amount of unprotected storage available. For more information on the system service tools function, see the *Backup and Recovery Guide*. If this error condition occurs again, consider configuring the checksum protection again, specifying a larger amount of unprotected storage. Use the Work with System Status (WRKSYSSTS) command to display your unprotected storage requirements.

CPI0956 Disk mirroring protection is suspended on a disk unit.

This message is sent if a disk unit has detected errors and is requesting service. Errors were found on a disk unit. Data has not been lost.

You can press F14 to run problem analysis.

CPI0957 Disk mirroring protection is suspended on a disk unit.

An error is found on a storage device controller. Data has not been lost. Every disk unit attached to the storage device controller is suspended from disk mirroring protection. For more information, see the previous messages.

After the service representative is finished repairing the storage device controller, do the following:

1. Type the Start System Service Tools (STRSST) command and press the Enter key.
2. Type the option to Work with disk units and press the Enter key.
3. Type the option to Work with disk unit recovery and press the Enter key.
4. Type the option to Suspend or resume mirroring protection and press the Enter key.
5. Type a 2 in the option field for the device with the specified address and press the Enter key. Mirroring protection on the disk unit is then resumed.

CPI0958 Disk mirroring protection is suspended on a disk unit.

An error was found on a bus. Data has not been lost. Every disk unit attached to the bus is suspended from disk mirroring protection. For more information, see the previous messages.

After the service representative is finished repairing the bus, do the following:

1. Type the Start System Service Tools (STRSST) command and press the Enter key.
2. Type the option to Work with disk units and press the Enter key.

3. Type the option to Work with disk unit recovery and press the Enter key.
4. Type the option to Suspend or resume mirroring protection and press the Enter key.
5. Type a 2 in the option field for the device with the specified address and press the Enter key. Mirroring protection on the disk unit is then resumed.

CPI0959 Disk mirroring protection is suspended on a disk unit.

This message is sent if a disk unit has detected errors and is requesting service. A disk unit is not operating.

To run problem analysis, press F14.

CPI0988 Disk mirroring protection is being resumed on a disk unit.

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 Disk mirroring protection is resumed on a disk unit.

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.

CPI0992 Disk error occurred.

This message is sent if a disk unit has determined that it has too many errors and is starting to fail. The system has recovered from the error. However, machine checks and possible data loss may occur if this condition is not corrected. You can press F14 to run problem analysis.

CPI0997 Nearly all available machine addresses used.

This message is sent if nearly all available machine addresses have been used for the reason specified. If the percentage of addresses used reaches 100 percent, the system ends abnormally. If reason 1 is given, at least the specified percentage of the maximum possible addresses for permanent objects have been used. If reason 2 is given, at least the specified percentage of the maximum possible addresses for temporary objects have been used.

You can use the Work with System Status (WRKSYSSTS) command to monitor the percentage of addresses used for permanent and temporary objects. Do an initial program load (IPL) to make more addresses available to the system. The IPL automatically releases available addresses to the system.

CPI2209 User profile deleted because it was damaged.

This message is sent when a user profile is deleted because it was damaged. This user profile may have owned objects prior to being deleted. These objects now have no owner. A Reclaim Storage

(RCLSTG) command can be used to transfer the ownership of these objects to the QDFTOWN user profile.

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.

Sample Program to Receive Messages from QSYSMSG

The following is a sample program which receives messages from the QSYSMSG message queue. The program 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 sample command CVTBINDEC (described under "CVTBINDEC (Convert from Binary to Decimal) Sample Command" on page 6-36) is used for this conversion. The program issues the ENDMOD command to prevent new jobs from being started until the situation is understood. It then sends the same message to a user-defined message queue to be reviewed by the security officer. It also sends a message to the system operator informing him of what occurred. If a different message is received, it is sent to the system operator.

A separate job would be started to call this sample program. The job would remain active, waiting for a message to arrive. The job could be ended using the ENDJOB command.

```
SEQNBR *... ... 1 ... ... 2 ... ... 3 ... ... 4 ... ... 5 ... ... 6 ... ... 7 ..

         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        &REASONB *CHAR LEN(2) /* Reason code in binary */
         DCL        &REASOND *DEC LEN(5 0) /* Reason code in dec */
         DCL        &RMTLOC *CHAR 8
         MONMSG     MSGID(CPF0000) EXEC(GOTO PROBLEM)
LOOP:    RCVMSG     MSGQ(QSYS/QSYSMSG) WAIT(*MAX) MSGID(&MSGID) +
                      MSG(&MSG) MSGDTA(&MSGDTA)
         IF         (&MSGID *EQ 'CPF1269') DO /* Start failed */
         CHGVAR     &REASONB %SST(&MSGDTA 45 2) /* Reason code */
         CVTBINDEC  FROMBIN(&REASONB) TODEC(&REASOND)
         IF         ((&REASOND *EQ 704) *OR (&REASOND *EQ 705)) DO
         CHGVAR     &MODE %SST(&MSGDTA 11 8) /* Extract mode */
         CHGVAR     &RMTLOC %SST(&MSGDTA 48 8) /* Get rmt loc name */
         ENDMOD     RMTLOCNAME(&RMTLOC) MODE(&MODE)
         SNDPGMMSG  MSGID(&MSGID) MSGF(QCPFMSG) MSGDTA(&MSGDTA) +
                      TOMSGQ(QSECOFR) /* Send to user created MSGQ */
         CHGVAR     &DEVICE %SST(&MSGDTA 1 10) /* Extract device */
         SNDPGMMSG  MSG('Device ' *CAT &DEVICE *TCAT ' Mode ' *CAT +
                      &MODE *TCAT ' had security failure - session +
                      max changed to zero') TOMSGQ(QSYSOPR)
         GOTO       LOOP
         ENDDO      /* End of reason code 704/705 */
         ENDDO      /* End of Start request that was not valid */
                    /* 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))
         GOTO       LOOP
PROBLEM: SNDPGMMSG  MSG('QSYSMSG job has abnormally ended') +
                      TOMSGQ(QSYSOPR)
         SNDPGMMSG  MSGID(CPF9898) MSGF(QCPFMSG) MSGTYPE(*ESCAPE) +
                      MSGDTA('Unexpected error occurred')
         ENDPGM
```

Using the System Reply List

The system reply list allows you to specify that the system issue the reply to specified predefined inquiry messages so the display station user does not need to reply.

The system reply list contains message identifiers, optional compare data, a reply value for each message, and a dump attribute. The system reply list applies only to predefined inquiry messages that are sent by a job that uses the system reply list. You specify that a job is to use the system reply list for inquiry messages on the INQMSGRPY(*SYSRPYL) parameter on the following commands:

- Job (JOB)
- Submit Job (SBMJOB)
- Change Job (CHGJOB)
- Create Job Description (CRTJOBD)
- Change Job Description (CHGJOBD)

When a predefined inquiry message is sent by a job that uses the system reply list, the system searches the reply list in ascending sequence number order for an entry that matches the message identifier and, optionally, the compare data of the reply message. If an entry is found, the reply specified is issued and the user is not required to enter a reply. If an entry is not found, the message is sent to the display station user for interactive jobs or system operator for batch jobs.

The system reply list is shipped with the system with the following initial entries defined:

Sequence Number	Message Identifier	Compare Value	Reply	Dump
10	CPA0700	*NONE	D	*YES
20	RPG0000	*NONE	D	*YES
30	CBE0000	*NONE	D	*YES
40	PLI0000	*NONE	D	*YES

These entries indicate that a reply of D is to be sent and a job dump is to be taken if the message CPA0700-CPA0799, RPG0000-RPG9999, CBE0000-CBE9999, or PLI0000-PLI9999 (which indicate a program failure) is sent by a job using the system reply list. For the system to use these entries, you must specify that the jobs are to use the system reply list.

To add other inquiry messages to the system reply list, use the Add Reply List Entry (ADDRPYLE) command. On this command you can specify the sequence number, the message identifier, optional compare data, reply action, and the dump attribute.

The following reply actions can be specified for the inquiry messages that are placed on the system reply list (the parameter value is given in parentheses):

- Send the default reply for the inquiry messages (*DFT). In this case, the default reply for the message is sent. The message is not displayed, and no default handling program is called.
- Require the work station user or system operator to respond to the message (*RQD). If the message queue to which the message is sent (work station message queue for interactive jobs and QSYSOPR for batch jobs) is in break mode, the message is displayed, and the work station 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('QPSPLPRTbbQSYSbbbbbbQSYSPRT')` (where b is a blank)
- `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	'QPSPLPRTbbQSYSbbbbbbQSYSPRT'	1	*RQD	*NO
25	CPA4002		1	G	*NO
27	CPA5316	'QSYSPRT'	21	I	*NO
30	CBE0000		1	D	*YES
40	PLI0000		1	D	*YES
9999	*ANY		1	*DFT	*NO

For a job that uses this system reply list, the following occurs when the messages that were added to the reply list are sent by the job:

- For sequence number 15, whenever an RPG1241 message is sent by a job that uses the system reply list, a reply of C is sent and the job is not dumped.
- For sequence number 18, a generic message identifier is used so whenever an RPG1200 inquiry message is sent by the job, the default reply is sent. The default reply can be the default reply specified in the message description or the system default reply. Before the default reply is sent, the job is dumped. The previous entry that was added overrides this entry for message RPG1241.
- For sequence number 22, if the inquiry message CPA4002 is sent with the compare data of QPSPLPRTbbQSYSbbbbbbQSYSPRT (where b is a blank) 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. For example, message CPA4002 has the following substitution data:

&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

Sequence number 22 tests for an ODP file name of QPSPLPRT, an ODP library name of QSYS, and an ODP device name of QSYSPRT. For an example of testing only one substitution variable, see sequence number 27.

- For sequence number 25, if the inquiry message CPA4002 is sent with the compare not equal to QPSPLPRTbbQSYSbbbbbbQSYSPRT (where b is a blank), a reply of G is sent. The job is not dumped.
- For sequence number 27, if the inquiry message CPA5316 is sent with the compare data of TESTEDFILETSTLIBRARYQSYSPRT, a reply of I is sent.

  When a compare value and a start position are specified, the compare value is compared with the message data beginning with the start position. In this case, position 21 is the beginning of the third substitution variable. For message CPA5316, the substitution variables are as follows:

&1	ODP file name	*CHAR	10
&2	ODP library name	*CHAR	10
&3	ODP device name	*CHAR	10
&4	Line number for first line	*BIN	2

Therefore, sequence number 27 tests for an ODP device name of QSYSPRT before sending a reply.

- For sequence number 9999, the message identifier of *ANY applies to any predefined inquiry message that is not matched by an entry with a lower sequence number, and the default reply for these inquiry messages is sent. If this entry were not included in the system reply list, the display station user would have to respond to all other inquiry messages that were not included in the system reply list.

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.

Message Logging

The two types of logs for messages are:

- Job log
- History log

A job log contains information related to requests entered for a job. The QHST log contains system data, such as a history of job activity on your system.

Job Log

Each job has an associated job log that can contain the following for the job:

- The commands in the job.
- The commands in a CL program if the CL program was created with the LOG(*YES) option or with the LOG(*JOB) option and a Change Job (CHGJOB) command was run with the LOGCLPGM(*YES) option (for more information on logging CL program commands, see "Logging CL Program Commands" on page 2-49).
- All messages and message help sent to the requester and not removed from the program message queues.

At the end of the job, the job log can be written to the output file QPJOBLOG so that it can be printed. After the job log is written to the output file, the job log is deleted. You can prevent a job log from being written to QPJOBLOG for successful completions. See the discussion on preventing job logs later in this chapter.

You can control what information is written in the job log. To do this, you specify the LOG parameter on the Create Job Description (CRTJOBD) command. You can change these levels using the Change Job (CHGJOB) command or the Change Job Description (CHGJOBD) command. The LOG parameter is made up of 3 values: message level, message severity, and message text level. For more information on these commands, see the *Work Management Guide*.

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:

- Logging level 1 information.
- Any requests or commands being logged from a CL program for which high-level messages are issued with a severity greater than or equal to the severity specified. If the request or command is logged, all its associated messages are also logged.

3 The following information is logged:

- Logging level 1 information.
- All requests or commands being logged from a CL program.
- If any high-level messages associated with a request or command have a severity greater than or equal to the severity specified, then all the messages associated with the request or command are logged.

4 The following information is logged:

- All requests or commands being logged from a CL program.
- Only messages with a severity code greater than or equal to the severity specified.

  Note: A high-level message is one that is sent to the program message queue of the program that receives the request or command being logged from a CL program.

The second value, message severity, specifies the minimum severity level that causes error messages to be entered in the job log. Values 0 through 99 are allowed. Only those messages with a severity greater than or equal to this value are entered in the job log.

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:

***MSG** Only message text is written to the job log (message help is included).

***SECLVL** The first-level message text and the second-level message text are written to the job log.

***NOLIST** No job log is produced if the job ends normally. If the job end code is 20 or greater, a job log is produced with the first-level message and second-level message text included.

The following example shows how the logging level affects the information that is stored in the job message queue and, therefore, written to the job log, if one is requested:

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 program message queue and to the program message queue of the program called before the specified call (*PRV). The messages that PGMA sends to its own program message queue are called *detailed messages*. Detailed messages are those messages that are sent to the program message queue of the lower level program call.

   PGMB sends two informational messages with severity codes of 40 and 50 to its own program message queue. These are detailed messages. PGMB also sends one informational message with a severity code of 10 to *PRV.

   Note that the CHGJOB command no longer appears on the display. According to logging level 2, only requests for which a message has been issued with a severity equal to or greater than that specified are saved for the job log, and no messages were issued for this request. If such a message had been issued, any detailed messages that had been issued would be saved for the job log and could be displayed by pressing F10.

```
                              Command Entry                         SYSTEM1
                                                           Request level:   1
Previous commands and messages:
      > CALL PGMA
        Message sev 20 - PGMA
        Message sev 50 - PGMA
        Message sev 60 - PGMA
      > CALL PGMB
        Message sev 10 - PGMB







                                                                        Bottom
Type command, press Enter.
===> ________________________________________________________________________
_____________________________________________________________________________
_____________________________________________________________________________
_____________________________________________________________________________
 F3=Exit   F4=Prompt   F9=Retrieve   F10=Include detailed messages
 F11=Display full   F12=Cancel   F13=User support   F16=System main menu
```

3. When F10=Include detailed messages is pressed from the Command Entry display, all the messages associated with the request are displayed.

```
                              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=Include detailed messages
 F11=Display full  F12=Cancel   F13=User support   F16=System main menu
```

4. When another command is entered (in this example, CHGJOB), the `CALL PGMB` command and all messages (including detailed messages) are removed because the severity code for the high-level message associated with this request was not equal to or greater than the severity code specified in the CHGJOB command. The `CALL PGMA` command and its associated messages remain because at least one of the high-level messages issued for that request has a severity code equal to or greater than that specified.

   On the following display, the CHGJOB command specifies a logging level of 3, a message severity of 40, and that both the first- and second-level text of a message are to be written to the job log. When another command is entered, the CHGJOB command remains on the display because logging level 3 logs all requests.

PGMC sends two messages with severity codes of 30 and 40 to the program message queue of the program called before the specified call (*PRV).

PGMD sends a message with a severity of 10 to *PRV.

```
                              Command Entry                         SYSTEM1
                                                             Request level:   1
 Previous commands and messages:
      > CALL PGMA
        Message sev 20 - PGMA
        Message sev 50 - PGMA
        Message sev 60 - PGMA
      > CHGJOB LOG(3 40 *SECLVL)
      > CALL PGMC
        Message sev 30 - PGMC
        Message sev 40 - PGMC
      > CALL PGMD
        Message sev 10 - PGMD

                                                                        Bottom
 Type command, press Enter.
 ===> ____________________________________________________________________
 _________________________________________________________________________
 _________________________________________________________________________

  F3=Exit   F4=Prompt   F9=Retrieve   F10=Exclude detailed messages
  F11=Display full      F12=Cancel    F13=User support    F16=System main menu
```

5. When another command is entered after the CALL PGMD command was entered, the CALL PGMD command remains on the display, but its associated message is deleted. The message is deleted because its severity code is not equal to or greater than the severity code specified on the LOG parameter of the CHGJOB command.

   The command SIGNOFF *LIST is entered to print the job log.

```
                              Command Entry                         SYSTEM1
                                                             Request level:   1
 Previous commands and messages:
      > CHGJOB LOG(3 40 *SECLVL)
      > CALL PGMC
        Message sev 30 - PGMC
        Message sev 40 - PGMC
      > CALL PGMD
      > CALL PGME

                                                                        Bottom
 Type command, press Enter.
 ===> SIGNOFF *LIST_______________________________________________________
 _________________________________________________________________________
 _________________________________________________________________________

  F3=Exit   F4=Prompt   F9=Retrieve   F10=Exclude detailed messages
  F11=Display full      F12=Cancel    F13=User support    F16=System main menu
```

The job log, which follows, contains all the requests and all the messages that have remained on the Command Entry display. In addition, the job log contains the message help associated with each message, as specified by the CHGJOB command. Notice that the job log contains the message help of any message issued during the job, not just for the messages issued since the second CHGJOB command was entered.

```
5728SS1 R03 M00 900824                          Job Log                                    02/22/90 15:27:08          Page    1

 Job name . . . . . . . . . . :   DSP07          User  . . . . :   MTN             Number . . . :   028510
 Job description  . . . . . . :   QDFTJOBD       Library . . . :   QGPL

TIME   MSGID   SEV TYPE  MESSAGE TEXT                                                     FROM PGM    INST TO PGM     INST

152708 CPF1124 00 INFO  Job 028510/MTN/DSP07 started on 02/22/90 at 15:27:08 in subsystem QINTER in  QWTPIIPP    0000 *EXT       0000
                        QSYS. Job entered system on 02/22/90 at 15:27:08.
                  RQS  -call pgma                                                          QMHGSD      0000 QCMD       0000
152728 USR1001 20 INFO  Detailed message sev  20 - PGMA                                    PGMA        0000 PGMA       0000
                           PGMA second-level text
152728 USR1002 50 INFO  Detailed message sev  50 - PGMA                                    PGMA        0000 PGMA       0000
                           PGMA second-level text
152728 USR1003 60 INFO  Detailed message sev  60 - PGMA                                    PGMA        0000 PGMA       0000
                           PGMA second-level text
152728 USR1004 20 INFO  Message sev  20 - PGMA                                             PGMA        0000 QCMD       0000
                           PGMA second-level text
152728 USR1005 50 INFO  Message sev  50 - PGMA                                             PGMA        0000 QCMD       0000
                           PGMA second-level text
152728 USR1006 60 INFO  Message sev  60 - PGMA                                             PGMA        0000 QCMD       0000
                           PGMA second-level text
                  RQS  -chgjob log(3 40 *seclvl)                                           QMHGSD      0000 QCMD       0000
                  RQS  -call pgmc                                                          QMHGSD      0000 QCMD       0000
152729 USR100A 30 INFO  Message sev  30 - PGMC                                             PGMC        0000 QCMD       0000
                           PGMC second-level text
152729 USR100B 40 INFO  Message sev  40 - PGMC                                             PGMC        0000 QCMD       0000
                           PGMC second-level text
                  RQS  -call pgmd                                                          QMHGSD      0000 QCMD       0000
                  RQS  -call pgme                                                          QMHGSD      0000 QCMD       0000
                  RQS  -signoff *list                                                      QMHGSD      0000 QCMD       0000
152832 CPF1164 00 COMP  Job 028510/MTN/DSP07 ended on 02/22/90 at 15:28:32; 1 seconds used; end code QWTMCEOJ    0000 *EXT       0000
                         0 .
                           Cause . . . . . :   Job 028510/MTN/DSP07 completed on 02/22/90 at 15:28:
                        32 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.  Th
                        e job ending codes and their meanings are as follows:  0 - The job completed
                         normally. 10 - The job completed normally during controlled ending or contr
                        olled subsystem ending. 20 - The job exceeded end severity (ENDSEV job attri
                        bute). 30 - The job ended abnormally. 40 - The job ended before becoming act
                        ive. 50 - The job ended while the job was active. 60 - The subsystem ended a
                        bnormally while the job was active. 70 - The system ended abnormally while t
                        he 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 Guide, SC21-8078.
```

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 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 date and time the job started.
- The time each message was sent.
- The message identifier.
- The message severity.
- The message type.
- The message. If the logging level specifies that second-level text is to be included, the second-level text appears on subsequent lines below the message.
- The program from which the message or request was sent.

- The machine interface instruction number or the offset into the program from which the message was sent.
- The program to which the message or request was sent.
- The machine interface instruction number or the offset into the program to which the message was sent.

Note: The machine interface instruction numbers appear only for escape, notify, and diagnostic messages. For all other message types, the machine interface instruction number is set to zero.

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. See the *APPC Programmer's Guide* for more information on APPC.

Displaying the Job Log

The way to display a job log depends on the status of the job.

- If the job has ended and the job log is not yet printed, use the Display Spooled File (DSPSPLF) command, as follows:

  ```
  DSPSPLF FILE(QPJOBLOG) JOB(001293/FRED/WS3)
  ```

  to display the job logs for job number 001293 associated with user FRED at display station WS3.
- If the job is still active (batch or interactive jobs) or is on a job queue and has not yet started, use the Display Job Log (DSPJOBLOG) command. For example, to display the job log of the interactive job for user JSMITH at display station WS1, enter:

  ```
  DSPJOBLOG JOB(nnnnnn/JSMITH/WS1)
  ```

  where nnnnnn is the job number.

To display the job log of your own interactive job, do one of the following:

- Enter the following command:

  ```
  DSPJOBLOG
  ```
- Enter the WRKJOB command and select option 10 (Display job log) from the Work with Job display.
- Press F10=Include detailed messages from the Command Entry display (this key displays the messages that are shown in the job log).
- If the input inhibited light on your display station is on and remains on, do the following:
  1. Press the System Request key, then press the Enter key.
  2. On the System Request menu, select option 3 (Display current job).
  3. On the Display Job menu, select option 10 (Display Job log, if active or on job queue).

4. On the Job Log display, DSPJOB appears as the processing request. Press F10 (Display detailed messages).
5. On the Display All Messages display, press the Roll Down key to see messages that were received before you pressed the System Request key.

- Sign off the work station, specifying LOG(*LIST) on the SIGNOFF command.

When you use the Display Job Log (DSPJOBLOG) command, you see the Job Log display. This display shows program names with special symbols, as follows:

>>	The running command or the next command to be run. For example, if a CL or HLL program was called, the call to the program is shown.
>	The command has completed processing.
. .	The command has not yet been processed.
?	Reply message. This symbol marks both those messages needing a reply and those that have been answered.

On the Job Log display, you can do the following:

- Press F10 to display detailed messages. This display shows the commands or operations that were run within an HLL program or within a CL program for which LOGCLPGM 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).

Preventing the Production of Job Logs

To prevent a job log from being produced at the completion of a batch job, you can specify *NOLIST for the message text-level value of the LOG parameter on the Batch Job (BCHJOB), Submit Job (SBMJOB), Change Job (CHGJOB), Create Job Description (CRTJOBD), or Change Job Description (CHGJOBD) command. If you specify *NOLIST for the message level value of the LOG parameter, the job log is not produced at the end of a job unless the job end code is 20 or greater. If the job end is 20 or greater, the job log is produced.

For an interactive job, the value specified for the LOG parameter on the SIGNOFF command takes precedence over the LOG parameter value specified for the job.

Job Log Considerations

The following suggestions apply to using job logs:

- To change the output queue for all jobs on the system, use the OUTQ or DEV parameter on the Change Printer File (CHGPRTF) command to change the file QSYS/QPJOBLOG. The following are two examples using each of the parameters:

```
CHGPRTF FILE(QSYS/QPJOBLOG)
       DEV (USRPRT)
```

and

```
CHGPRTF FILE(QSYS/QPJOBLOG)
       OUTQ(USROUTQ)
```

- 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 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.

Considerations for Interactive Job Logs

The IBM-supplied job descriptions QCTL, QINTER, and QPGMR all have a log level of LOG(4 0 *NOLIST); therefore, all messages and both first- and second-level text for the messages are written to the job log. However, the job logs are not printed unless you specify *LIST on the SIGNOFF command. To change the log level for interactive jobs, you can use the CHGJOB or CHGJOBD command.

If a display station user uses an IBM-supplied menu or the command entry display, all error messages are displayed. If the display station user uses a user-written initial program, any unmonitored message causes the initial program to end and a job log to be produced. However, if the initial program monitors for messages, it receives control when a message is received. In this case, it may be important to ensure that the job log is produced so you can determine the specific error that occurred. For example, assume that the initial program displays a menu that includes a sign-off option, which defaults to *NOLIST. The initial program monitors for all exceptions and includes a Change Variable (CHGVAR) command that changes the sign-off option to *LIST if an exception occurs:

```
          PGM
          DCLF MENU
          DCL &SIGNOFFOPT TYPE(*CHAR) LEN(7) VALUE(*NOLIST)
          .
          .
          .
          MONMSG MSG(CPF0000) EXEC(GOTO ERROR)
PROMPT:   SNDRCVF RCDFMT(PROMPT)
          CHGVAR &IN41 '0'
          .
          .
          .
          IF (&OPTION *EQ '90') SIGNOFF LOG(&SIGNOFFOPT)
          .
          .
          .
          GOTO PROMPT
ERROR:    CHGVAR &SIGNOFFOPT '*LIST'
          CHGVAR &IN41 '1'
          GOTO PROMPT
          ENDPGM
```

If an exception occurs in the previous example, the CHGVAR command changes the option on the SIGNOFF command to *LIST and sets on an indicator. This indicator could be used to condition a constant that displays a message informing the display station user that an unexpected event has occurred and telling him what to do.

If the interactive job is running a CL program, the CL program commands are logged only if the log level is 3 or 4 and one of the following is true:

- LOG(*YES) was specified on the Create Control Language Program (CRTCLPGM) command.
- LOG(*JOB) was specified on the Create Control Language Program (CRTCLPGM) command and (*YES) is the current LOGCLPGM job attribute.

  The LOGCLPGM job attribute can be set and changed by using the LOGCLPGM parameter on the SBMJOB, CRTJOBD, CRTJOBD, and CHGJOBD commands.

Considerations for Batch Job Logs

For your batch applications, you may want to change the amount of information logged. The log level (LOG(4 0 *NOLIST)) specified in the job description for the IBM-supplied subsystem QBATCH supplies a complete log if the job abnormally ends. If the job completes normally, no job log will be produced.

If you want to print the job log in all cases, use the Change Job Description (CHGJOBD) command to change the job description, or specify a different LOG value on the BCHJOB or SBMJOB command. See "Job Log" on page 8-33 for a description of logging levels.

If the batch job is running a CL program, the CL program commands are logged only if LOG(*YES) is specified on the Create Control Language Program (CRTCLPGM) command or LOGCLPGM(*YES) is specified on the Change Program (CHGPGM) command.

QHST History Log

The history log (QHST) consists of a message queue and a physical file known as a log-version. Messages sent to the history-log message queue are written by the system to the current log-version physical file.

- History log (QHST). Contains a high-level trace of system activities such as system, subsystem, and job information, device status, and system operator messages. Its message queue is QHST.

When a log-version is full, a new version of the log is automatically created. Each version is a physical file that is named in the following way:

```
Qxxxyydddn
```

where:

xxx
: Is a 3-character description of the log type (HST)

yyddd
: Is the Julian date on which the log-version was created

n
: Is a sequence number within the Julian date (0 through 9 or A through Z)

Note: The number of records in the version of the history log is specified in the system value QHSTLOGSIZ.

You can also write a program to process history log records. Because several versions of each log may be available, you must select the log-version to be processed. To determine the available log-versions, use the Display Object Description (DSPOBJD) command. For example, the following DSPOBJD command displays what versions of the history log are available:

```
DSPOBJD   OBJ(QSYS/QHST*)   OBJTYPE(*FILE)
```

You can delete logs on your system using the delete option from the display presented on the Work with Objects (WRKOBJ) command. You can display or print the information in a log using the Display Log (DSPLOG) command. You can select the information you want displayed or printed by specifying any combination of the following:

- Period of time
- Name of job that sent the log entry
- Message identifiers of entries

The following DSPLOG command displays all the available entries for the job OEDAILY in the current day:

```
DSPLOG JOB(OEDAILY)
```

The resulting display is:

```
                        Display History Log Contents

 Job OEDAILY started
 Database file OEMSTR in library OELIB expired
 Job OEDAILY abnormally ended
 Job OEDAILY started
 Job OEDAILY ended

                                                                     Bottom
 Press Enter to continue.

 F3=Exit   F10=Display all   F12=Cancel
```

If you have reset the system date or time to an earlier setting, a system log-version can contain entries that are not in chronological order and, therefore, when you try to display the log-version, some entries may be missed. For example, if the log-version contains entries dated 1988 followed by entries dated 1987, and you want to display those 1987 entries, you specify the 1987 dates on the PERIOD parameter on the DSPLOG command but the expected entries are not displayed. You should always use the system date (QDATE) and the system time (QTIME), or you should specify the PERIOD parameter as follows:

```
PERIOD((start-time start-date) (*AVAIL *END))
```

The system writes the messages sent to a log message queue to the current version physical file when the message queue is full or when the DSPLOG command was used. If you want to ensure the current version is up-to-date, specify a fictitious message identifier, such as ###0000, on the DSPLOG command. No messages are displayed, but the log-version physical file is made current.

If you print the information in a log using the Display Log (DSPLOG) command, only 105 characters of message text is shown. Any characters after 105 characters are truncated at the right.

Format of the History Log

A database file is used to store the messages sent to a log on the system. Because all records in a physical file have the same length and messages sent to a log have different lengths, the messages can span more than one record. Each record for a message has three fields:

- System date and time (a character field of length 8). This is an internal field. The converted date and time also are in the message.

- Record number (a 2-byte field). For example, the field contains hex 0001 for the first record, hex 0002 for the second record, and so on.
- Data (a character field of length 132).

The third field (data) of the first record has the following format:

Contents	Type	Length	Positions in Record
Job name	Character	26	11-36
Converted date and time[1]	Character	13	37-49
Message ID	Character	7	50-56
Message file name	Character	10	57-66
Library name	Character	10	67-76
Message type[2]	Character	2	77-78
Severity code	Character	2	79-80
Sending program name	Character	12	81-92
Sending program instruction number	Character	4	93-96
Receiving program name	Character	10	97-106
Receiving program instruction number	Character	4	107-110
Message text length	Binary	2	111-112
Message data length	Binary	2	113-114
Reserved	Character	28	115-142

[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.

The third field (data) of the remaining records has the following format:

Contents	Type	Length
Message	Character	Variable[1]
Message data	Character	Variable[2]

[1] This length is specified in the first record (positions 111 and 112) and cannot exceed 132.

[2] This length is specified in the first record (positions 113 and 114).

A message is never split when a new version of a log is started. The first and last records of a message are always in the same QHST version.

For a description of the message data for a message, see "Defining Substitution Variables" on page 7-8 .

Processing the QHST File

If you use an HLL program to process the QHST file, keep in mind that the length of the message can vary with each occurrence of a message. Because the message includes replaceable variables, the actual length of the message varies; therefore, the message data begins at a variable location for each use of the same message.

However, for message CPF1124 (job start) and message CPF1164 (job completion), the message data always begins in position 11 of the third record.

Performance information is not displayed as text on message CPF1164. Because the message is in the QHST log, users can write application programs to retrieve this data. The format of this performance information is as follows.

The performance information is passed as a variable length replacement text value. This means that the data is in a structure with the first entry being the length of the data. The size of the length field is not included in the length. The first data fields in the structure are the times and dates that the job entered the system and when the first routing step for the job was started. The times are in the format 'hh:mm:ss'. The separators are always colons. The dates are in the format defined in the system value QDATFMT and the separators are in the system value QDATSEP. The time and date the job entered the system precede the job start time and date in the structure.

The time and date the job entered the system are when the system becomes aware of a job to be initiated (a job structure is set aside for the job). For an interactive job, the job entry time is the time the password is recognized by the system. For a batch job, it is the time the BCHJOB or SBMJOB command is processed. For a monitor job, reader or writer, it is the time the corresponding start command is processed, and for autostart jobs it is during the start of the subsystem.

Following the times and dates are the total response time and the number of transactions. The total response time is in seconds and contains the accumulated value of all the intervals the job was processing between pressing the Enter key at the work station and when the next display is shown. This information is similar to that shown on the WRKACTJOB display. This field is only meaningful for interactive jobs.

It is also possible in the case of a system failure or abnormal job end that the last transaction will not be included in the total. The job end code in this case would be a 40 or greater. The transaction count is also only meaningful for interactive jobs other than the console job and is the number of response time intervals counted by the system during the job.

The number of synchronous auxiliary I/O operations follows the number of transactions. This is the same as the AUXIO field that appears on the WRKACTJOB display except that this value is the total for the job. If the job ends with a end code of 70, this value may not contain the count for the final routing step. Additionally, if a job exists across an IPL (using a TFRBCHJOB command) it is ended before becoming active following an IPL, the value will be 0.

The final field in the performance statistics is the job type. Values for this field are:

A Automatically started job

B Batch job

I Interactive job

M Subsystem monitor

R Spooling reader

S System job

W Spooling writer

X Start job

For messages in which the message data begins in a variable position, you can access the message data by doing the following:

- Determine the length of the variables in the message. For example, assume that a message uses the following five variables:

  ```
  Job name    *CHAR 10
  User name   *CHAR 10
  Job number  *CHAR 6
  Time        *CHAR 8
  Date        *CHAR 8
  ```

  These variables are fixed in the first 42 positions of the message data.

- To find the location of the message data, consider that:

  - The message always begins in position 11 of the second record.

  - The length of the message is stored in a 2-position field beginning at position 111 of the first record. This length is stored in a binary value so if the message length is 60, the field contains hex 003C.

  Then, by using the length of the message and the start position of the message, you can determine the location of the message data.

Example of Processing the QHST File

The following RPG/400 program processes the QHST file for the job completion message CPF1164. The QHST file in the program must be overridden to one of the actual QHST files. The program reads each record from the file into an array. A data structure is defined to match the message data for CPF1164. You can determine the structure of the message data for a message by using the Display Message Description (DSPMSGD) command.

For message CPF1164, there are always three records and the message data always begins in position 11 of the third record. The program checks for the first record of a message (RCDNBR = 1) and then checks for CPF1164 in positions 50 through 56. If the message is CPF1164, the program sets on indicator 22 so that when the third record is read, the message data can be moved to the data structure. Once the message data is in the data structure, the fields can be printed, as in this example, or written to a database file. Note that the RTGSTP field contains the number of routing steps used by the job.

Some of the information in the message data is not in the text for CPF1164. This includes:

- Number of routing steps.
- Time and date the job was entered (this is the submitted time for batch jobs; for interactive jobs, it is the same as the time and date started).
- Time and date started.
- Total response time (*).
- Number of transactions (*) (number of display station interactions).
- Number of auxiliary storage I/O operations (*).
- Job type.

Note: The asterisked (*) fields are the same as those on the WRKACTJOB display.

```
*... ... 1 ... ... 2 ... ... 3 ... ... 4 ... ... 5 ... ... 6 ... ... 7 ..

     FQHST    IP  F     142          DISK
     FQPRINT  O   F     132     OF     PRINTER
     E                    ARR       132  1               INPUT ARRAY
     IQHST    AA  01
     I                                    B   9  100RCDNBR
     I                                       11 142 ARR
     I                                       50  56 MSGNBR
     IMSG         DS
     I                                        1  10 JOB
     I                                       11  20 USER
     I                                       21  26 JOBNBR
     I                                       27  34 JOBTIM
     I                                       35  42 JOBDAT
     I                                    B  43  460CPUTIM
     I                                    B  47  480RTGSTP
     I                                    B  49  500CMPCOD
     I                                       73  80 TIMENT
     I                                       81  88 DATENT
     I                                       89  96 STRTIM
     I                                       97 104 STRDAT
     I                                    B 105 1080TOTRSP
     I                                    B 109 1120NBRTRN
     I                                    B 113 1160NBRAUX
     I                                      117 117 JOBTYP
     C           1         COMP RCDNBR                   20
     C   20      'CPF1164' COMP MSGNBR                   22
     C   20                GOTO END
     C           3         CABNERCDNBR    END            12
     C  N22                GOTO END
     C                     MOVEAARR       MSG
     C                     EXCPTDETAIL
     C           END       TAG
     OQPRINT  E  1             DETAIL
     O                         JOB
     O                         USER
     O                         JOBNBR
     O                         JOBTIM  +  1
     O                         JOBDAT  +  1
     O                         CPUTIM  +  1
     O                         RTGSTP  +  1
     O                         CMPCOD  +  1
     O                         TIMENT  +  1
     O                         DATENT  +  1
     O                         STRTIM  +  1
     O                         STRDAT  +  1
     O                         TOTRSP  +  1
     O                         NBRTRN  +  1
     O                         NBRAUX  +  1
     O                         JOBTYP  +  1
```

Deleting Log Files

Log-version physical files accumulate on a system and you should periodically delete old logs that are not needed. A log-version is created such that only the security officer is authorized to delete it.

To delete a log-version, use the Delete File (DLTF) command and specify the object name or generic name to be deleted.

Another alternative is to use the following sample command and program to simplify deleting logs on your system. The Delete Log (DLTLOG) command specifies the log and the number of days to keep logs. The default log is QHST, and the default number of days is five. For example, if you keep the defaults, all QHST files that are older than five days from the current date are deleted. The last date in which a message was sent to the log is used for the comparison, *not* the creation date of the log.

Because the security officer is the only user who can delete the log files, you may want to create the program with USRPRF(*OWNER) and authorize the command and program to specific users.

This function is dependent on the accurate entry of the system date during IPL You may want to add this command to other standard processing functions of QHST and include them as an autostart job.

The command accepts the name of the log to be deleted and the number of days that the logs are to be kept. The current date is retrieved and converted to Julian format. The program subtracts the number of days entered on the command from the last date in which a message was sent to the log, and that number is used. The text description of each log file contains the date that the last message was sent to it. The program also tests for year-end. The names of the log files are put into an output file in library QTEMP by the DSPOBJD command. The number of files deleted and kept is counted and placed in the completion message.

Enter the following command definition statements and create the DLTLOG command with the parameter PGM(DLTLOGC) specified.

```
SEQNBR *... ... 1 ... ... 2 ... ... 3 ... ... 4 ... ... 5 ... ... 6 ... ... 7 ..

 01.00          CMD        PROMPT('Delete Log')  /* CPP IS DLTLOGC */
 02.00          PARM       LOG *CHAR LEN(4) DFT(QHST) RSTD(*YES) +
 03.00                        VALUES('QHST') +
 04.00                        PROMPT('Log (QHST):')
 05.00          PARM       DAYS *DEC LEN(2 0) DFT(5) +
 06.00                        RANGE(1 50) +
 07.00                        PROMPT('Nbr of days to keep log')
```

Enter the following CL program source statements and create the DLTLOGC program:

```
SEQNBR *... ... 1 ... ... 2 ... ... 3 ... ... 4 ... ... 5 ... ... 6 ... ... 7 ..

                PGM         PARM(&LOG &DAYS)
                DCLF        QSYS/QADSPOBJ /* IBM file to be overridden */
                DCL         &LOG *CHAR LEN(4)
                DCL         &DAYS *DEC LEN(2 0)
                DCL         &WRKDAT *CHAR LEN(6)
                DCL         &JULIANA *CHAR LEN(5)
                DCL         &YRD *DEC LEN(2 0)
                DCL         &DAYSD *DEC LEN(3 0)
                DCL         &NUM3 *DEC LEN(3 0)
                DCL         &QHSTLAST *CHAR LEN(6)
                DCL         &QHSTLASTJ *CHAR LEN(5)
                DCL         &DLTCNT *DEC LEN(5 0)
                DCL         &DLTCNTA *CHAR LEN(5)
                DCL         &RMNCNT *DEC LEN(5 0)
                DCL         &RMNCNTA *CHAR LEN(5)
                DCL         &MSGID *CHAR LEN(7)
                DCL         &MSGDTA *CHAR LEN(100)
                DCL         &BLANKS *CHAR LEN(6) VALUE('      ') /* 6 blanks value */
                MONMSG      MSGID(CPF0000) EXEC(GOTO ERROR)
                            /* Subtract nbr of days from the current date */
                RTVJOBA     DATE(&WRKDAT) /* Rtv current date */
                CVTDAT      DATE(&WRKDAT) TOVAR(&JULIANA) TOFMT(*JUL) +
                                 TOSEP(*NONE) /* Convert to Julian */
                CHGVAR      &YRD %SST(&JULIANA 1 2)
                CHGVAR      &DAYSD %SST(&JULIANA 3 3)
                CHGVAR      &NUM3 (&DAYSD - &DAYS) /* Sub for cmd days */
                IF          (&NUM3 *LE 0) DO   /* Went passed Jan 1 */
                CHGVAR      &YRD (&YRD - 1)
                CHGVAR      &DAYSD (365 + &NUM3) /* Assume 365 per year */
                ENDDO
                ELSE        CHGVAR    &DAYSD &NUM3 /* Same year */
                CHGVAR      %SST(&JULIANA 1 2) &YRD  /* Cnvt to Julian */
                CHGVAR      %SST(&JULIANA 3 3) &DAYSD  /* Cnvt to Julian */
                            /* Create OUTFILE of log descriptions */
                 DSPOBJD    OBJ(QSYS/(&LOG *CAT '*')) OBJTYPE(*FILE) +
                              OUTPUT(*OUTFILE) OUTFILE(QTEMP/DSPOUTP)
                            /* Delete log loop */
                OVRDBF      QADSPOBJ TOFILE(QTEMP/DSPOUTP)
      LOOP:     RCVF        RCDFMT(QLIDOBJD) /* DSPOBJD format */
                MONMSG      MSGID(CPF0864) EXEC(GOTO END)
                            /* Extract the date of the last msg sent */
                            /*   to the log file.  This is written   */
                            /*   as part of the text description of  */
                            /*   the object beginning in position 15 */
                CHGVAR      &QHSTLAST %SST(&ODOBTX 15 6)
                IF          (&QHSTLAST *EQ &BLANKS) DO /* Is to date blank? */
                CHGVAR      &RMNCNT(&RMNCNT+1) /* Count remaining */
                GOTO        LOOP
                ENDO
                CVTDAT      DATE(&QHSTLAST) TOVAR(&QHSTLASTJ) +
                                 FROMFMT(*YMD) TOFMT(*JUL) +
                                 TOSEP(*NONE) /* Convert to Julian */
                IF          (&QHSTLASTJ *LE &JULIANA) DO /* Comp dates */
                DLTF        QSYS/&ODOBNM /* Delete the log file */
                CHGVAR      &DLTCNT (&DLTCNT + 1)  /* Count decisions */
                ENDDO
                ELSE        +
                CHGVAR      &RMNCNT (&RMNCNT + 1) /* Count remaining */
                GOTO        LOOP
                            /* Send completion message */
```

```
      END:                      CHGVAR &DLTCNTA &DLTCNT /* Convert to alpha */
                   CHGVAR       &RMNCNTA &RMNCNT  /* Convert to alpha */
                   SNDPGMMSG    MSG('Number of files deleted-' *CAT +
                                      &DLTCNTA *CAT +
                                      '    Number remaining-' *CAT +
                                      &RMNCNTA)
                   DLTF         QTEMP/DSPOUTP
                   RETURN
      ERROR:       RCVMSG       MSGID(&MSGID) MSGDTA(&MSGDTA) MSGTYPE(*EXCP)
                   SNDPGMMSG    MSGID(&MSGID) MSGDTA(&MSGDTA) +
                                     MSGF(QCPFMSG) MSGTYPE(*ESCAPE)
                   ENDPGM
```

You could change this program to include the following functions:

- Copy the QHST file to another file.
- Save the QHST file.
- Produce standard reports on the file.

Chapter 9. Defining Commands

A CL command is a statement that requests that the system perform a function. The function is performed by a program that is run when the command is entered. CL commands allow you to request a broad range of functions. You can use these commands as they have been supplied by IBM, change the default values supplied by IBM, or you can define your own commands. This chapter describes how you can define and create your own commands.

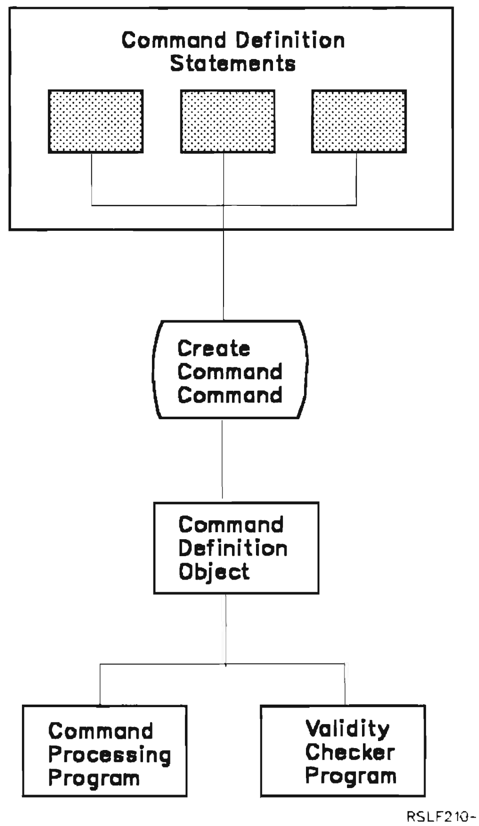
Overview of How to Define Commands

By defining your own CL commands, you can:

- Call application programs or REXX procedures
- Set default values for parameters
- Perform validity checking outside your application programs
- Define prompt text for the command to be used interactively
- Simplify system operator or programmer tasks by having your command run one or more frequently used commands
- Change certain attributes of the IBM-supplied commands, and then define a command to do one or more of the following:
  - Restrict the possible parameter values
  - Prompt for key parameters before other parameters
  - Change the default values of some of the parameters
  - Change the command name to require fewer keystrokes

Steps for Creating CL Commands

The following illustrates the steps involved in creating a command. The text that follows the illustration further describes each step.



Writing your own validity checking program is an optional step.

Command Definition Statements

The command definition statements contain the information that is necessary to prompt the work station user for input, to validate that input, and to define the values to be passed to the program that is called when the command is run.

Command definition statements may exist in any file supported as input to the CRTCMD command. For example, source entry utility (SEU) source files, diskette files, and other device files may contain command definition statements. They are usually entered in a source file by SEU. Table 9-1 on page 9-3 contains the statements used for defining commands.

Table 9-1. Statements for Defining CL Commands

Name of Command	Name of Keyword	Description
Command	CMD	Specifies the prompt text, if any, for the command name
Parameter	PARM	Defines a parameter or key parameter for a command
Element	ELEM	Defines an element in a list used as a parameter value
Qualifier	QUAL	Defines a qualifier of a name used as a parameter
Dependent	DEP	Defines the relationship among parameters
Prompt Control	PMTCTL	Defines the prompting given for parameters

Create Command (CRTCMD) Command

The CRTCMD command processes the command definition statements to create the command definition object. The CRTCMD command may be run interactively or in a batch job.

Command Definition Object

The command definition object is the object that is checked by a system program to ensure that the command is valid and that the proper parameters were entered.

Validity Checking

The system performs validity checking on commands. You may also write your own validity checking program although it is not required.

The validity checking performed by the system ensures that:

- 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 or the QCMDCHK program by a call from a HLL. See Chapter 6 for more information on the QCMDEXC program.
- A CL program is created.
- Commands are run by a CL program.

The validity checking program is optional.

If you need more validity checking than the system performs, you can write a program called a *validity checking* (see "Writing a Validity Checking Program" on page 9-54) or you can include the checking in the command processing program. Some validity checking you may want to do is check for the existence of an object or check the contents of a data area. You specify the names of both the command processing and validity checking programs on the CRTCMD command.

If you write a validity checking program, the command parameter values are passed to the validity checking before being passed to the command processing program. If all required parameters have constants specified, the program you write is also called during syntax checking of spooled input streams and when SEU is used to enter your commands into a database file. If an error is found, a message is issued to the user so errors can be corrected immediately. If you write a validity checking program, the command processing program can assume that the data passed to it is correct.

Command Processing Program

The command processing program (CPP) is the program that the command calls to perform the function requested. The CPP can be a CL, REXX, or an HLL program. For example, it can be an application program that your command calls, or it can be a CL or REXX program that contains a system command or series of commands.

The CPP must accept the parameters as defined by the command definition statements.

Authority Needed for the Commands You Define

For users to use a command you create, they must have operational authority to the commands and data authority to the command processing program and optional validity checking program. They also must have read authority to the library containing the command, to the command processing program, and to the validity checking program. If other commands are run in the command processing program or if files are opened, the user must also have the proper authority to these command processing programs or files.

Example of Creating a Command

If you want to create a command to allow the system operator to call a program to start the system, you would do the following. (This example assumes you are using IBM-supplied source files.)

1. Enter the command definition source statement into the source file QCMDSRC using the member name of STARTUP.

   ```
   CMD PROMPT('S Command for STARTUP')
   ```

2. Create the command by entering the following command.

   ```
   CRTCMD CMD(S) PGM(STARTUP) SRCMBR(STARTUP)
   ```

3. Enter the source statements for the STARTUP program (the command processing program).

```
PGM
STRSBS QINTER
STRSBS QBATCH
STRSBS QSPL
STRPRTWTR DEV(QSYSPRT) OUTQ(QPRINT) WTR(WTR)
STRPRTWTR DEV(WSPR2) OUTQ(WSPRINT) WTR(WTR2)
SNDPGMMSG MSG('STARTUP procedure completed') MSGTYPE(*COMP)
ENDPGM
```

4. Create the program using the Create CL Program (CRTCLPGM) command.

```
CRTCLPGM STARTUP
```

In the previous example, S is the name of the new command (specified by the CMD parameter). STARTUP is the name of the command processing program (specified by the PGM parameter) and also the name of the source member that contains the command definition statement (specified by the SRCMBR parameter). Now the system operator can either enter `S` to call the command or `CALL STARTUP` to call the command processing program.

How to Define Commands

To create a command, you must first define the command through command definition statements, which are described in detail in the *CL Reference*. The general format of the command definition statements and a summary of the coding rules follow.

Statement	Coding Rules
CMD	One and only one CMD statement must be used. The CMD statement can be placed anywhere in the source file.
PARM	A maximum of 75 PARM statements is allowed. The order in which you enter the PARM statements into the source file determines the order in which the parameters must be specified at processing time. 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 are not allowed with prompt control.
ELEM	A maximum of 300 ELEM statements is allowed in one list. The order in which you enter the ELEM statements into the source file determines the order of the elements in the list. The first ELEM statement must have a statement label that matches the statement label on the TYPE parameter on the PARM or ELEM statement for the list.
QUAL	A maximum of 300 qualifiers is allowed for a qualified name. The order in which you enter the QUAL statements into the source file determines the order in which the qualifiers must be specified and the order in which they are passed to the validity checking program and command processing program.
DEP	The DEP statement must be placed after all PARM statements it refers to. Therefore, the DEP statements are normally placed near the end of the source file.
PMTCTL	The PMTCTL statement must be placed after all PARM statements it refers to. Therefore, the PMTCTL statements are normally placed at the end of the source file.

At least one PARM statement must precede any ELEM or QUAL statements in the source file. The source file in which you enter the command definition statements is used by the CRTCMD command when you create a command. For information on entering statements into a source file, see the *Database Guide*.

Using the CMD Statement

When you define a command, you must include one and only one CMD statement with your command definition statements.

When you define a command, you can provide command prompt text for the user. If the user chooses to be prompted for the command instead of entering the entire command, the user types in the command name and presses F4 (Prompt). The command prompt is then displayed with the command name and the heading prompt text on line 1 of the display.

If you want to specify prompt text for the command, use the PROMPT parameter on the CMD statement to specify the heading for the prompt. You then specify the prompts for the parameters, elements of a list, and qualifiers on the PROMPT parameters for the PARM, ELEM, and QUAL statements.

On the PROMPT parameter for the CMD statement, you can specify the actual prompt heading text as a character string 30 characters maximum, or you can specify the message identifier of a message description. In the following example, a character string is specified for the command ORDENTRY.

```
CMD PROMPT('Order Entry')
```

Line 1 of the prompt looks like this after the user types in the command name and presses F4.

```
Order Entry (ORDENTRY)
```

If you do not provide prompt text for the command you are defining, you only need to use the word CMD for the CMD statement. However, you may want to use the PROMPT keyword for documentation purposes.

Defining Parameters

You can define as many as 75 parameters for each command. To define a parameter, you must use the PARM statement.

On the PARM statement, you specify the following:

- Name of the keyword for the parameter
- Whether or not the parameter is a key parameter
- Type of parameter value that can be passed
- Length of the value
- If needed, the default value for the parameter. (On the CRTCMD command, you can specify the maximum number of parameters that can be specified.)

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 and/or validity checking program if the parameter being defined is not specified (PASSVAL).
- What the message identifier is or what the prompt text for the parameter is (PROMPT).
- What valid values are shown in the possible choices field on the prompt display (CHOICE).
- Whether the choice values are provided by a program (CHOICEPGM).
- Whether prompting for a parameter is controlled by another parameter (PMTCTL).
- Whether values for a PMTCTL statement are provided by a program (for parameters referred to in CTL keywords) (PMTCTLPGM).
- Whether the value is to be hidden in the job log or hidden when the command is being prompted (DSPINPUT).

Naming the Keyword for the Parameter

The name of the keyword you choose for a parameter should be descriptive of the information being requested in the parameter value. For example, USER for user name, CMPVAL for compare value, and OETYPE for order entry type. The keyword can be as long as 10 alphanumeric characters, the first of which must be alphabetic.

Parameter Types

The basic parameter types are (parameter TYPE value given in parentheses):

- Decimal (*DEC). The parameter value is a decimal number, which is passed to the command processing program as a packed decimal value of the length specified on the LEN parameter.
- Logical (*LGL). The parameter value is a logical value, '1' or '0', which is passed to the command processing program as a character string of length 1 (F1 or F0).
- Character (*CHAR). The parameter value is a character string, which can be enclosed in apostrophes and which is passed to the command processing program as a character string of the length specified on the LEN parameter. The value is passed with its apostrophes removed, is left-justified, and is padded with blanks.
- Name (*NAME). The parameter value is a character string of which the first character is alphabetic (A-Z), $, #, or @, and the remaining characters are alphanumeric or (***) or a string of special characters enclosed in single quotation marks (') or double quotation marks ("). Periods and underscores

are also valid for type *NAME. The value is passed to the command processing program as a character string of the length specified in the LEN parameter. The value is left-justified and is padded with blanks. The *NAME type is normally used for object names. If a special value such as *LIBL or *NONE can be entered for the name parameter value, you must describe the special value on the SPCVAL parameter. Then, if the display station user enters a special value for the name parameter values, the rules for name verification are bypassed.

- 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.
- 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, which is passed to the command processing program in the format cyymmdd (c = century digit, y = year, m = month, d = day). The system sets the century digit based on the year in the specified date. The century digit is 0 if yy equals a number from 40 through 99; it is 1 if yy equals a number from 00 through 39. The user must enter the date for the command in the format specified by the date format (DATFMT) job attribute. The date separator (DATSEP) job attribute determines the optional separator character to be used when the date is entered. The DATFMT and DATSEP job attributes can be changed with the Change Job (CHGJOB) command.
- Time (*TIME). The parameter value is a character string, which is passed to the command processing program in the format hhmmss (h = hour, m = minute, s = second).
- Hexadecimal (*HEX). The parameter value is a hexadecimal value. The characters specified must be 0 through F. The value is passed to the CPP as hexadecimal (EBCDIC) characters (2 hexadecimal digits per byte), and is right adjusted and padded with zeros. If the value is enclosed in apostrophes, an even number of digits is required.
- Zero elements (*ZEROELEM). The parameter value is considered to be a list of zero elements for which no value can be specified in the command. This parameter type is used to prevent a value from being entered for a parameter that is a list even though the command processing program (CPP) expects a value. For example, if two commands use the same CPP, one command could pass a list for a parameter, and the other command may not have any values to pass. The parameter for the second command would be defined with TYPE(*ZEROELEM).
- Integer (*INT2 or *INT4). The parameter value is an integer passed as a 2-byte or 4-byte signed binary number. CL programs do not support binary values in variables.

- 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. It is passed to the CPP as a command string.
- Statement label. The statement label identifies the first of a series of QUAL or ELEM statements that further describe the qualified name or the mixed list being defined by this PARM statement.

The following parameter types are for IBM-supplied commands only.

- Expression (*X). The parameter value is a character string, variable name, or numeric value. The value is passed as a numeric value if it contains only digits, a plus or minus sign, and/or a decimal point; otherwise, it is passed as a character string.
- Variable name (*VARNAME). The parameter value is a variable name, which is passed to the command processing program as a character string. The value is left-justified and is padded with blanks. A variable is a name that refers to an actual data value during processing. A variable name can be as long as 10 alphanumeric characters (the first of which must be alphabetic) preceded by an ampersand (&); for example, &PARM.
- Command (*CMD). The parameter value is a command. For example, the CL command IF has a parameter named THEN whose value must be another command.

Length of Parameter Value

You can also specify a length (LEN parameter) for parameter values of the following types. For parameter types of date or time, date is always 7 characters long and time is always 6 characters long. The following shows the maximum length for each parameter type and the default length for each parameter type for which you can specify a length.

Data Type	Maximum Length	Default Length
*DEC	24 (9 decimal positions)	15 (5 decimal positions)
*LGL	1	1
*CHAR	3000	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	6000	256

The maximum length shown here is the maximum length allowed for these parameter types when the command is run. However, the maximum length 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.

Default Values

If you are defining an optional parameter, you can define a value on the DFT parameter to be used if the user does not enter the parameter value on the command. This value is called a *default value*. The default value must meet all the value requirements for that parameter (such as type, length, and special values). If you do not specify a default value for an optional parameter, the following default values are used.

Data Type	Default Value
*DEC	0
*INT2	0
*INT4	0
*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

Example of Defining a Parameter

The following example defines a parameter OETYPE for a command to call an order entry application.

```
PARM   KWD(OETYPE)  TYPE(*CHAR)  RSTD(*YES) +
       VALUES(DAILY WEEKLY MONTHLY)  MIN(1) +
       PROMPT('Type of order entry:')
```

The OETYPE parameter is required (MIN parameter is 1) and its value is restricted (RSTD parameter is *YES) to the values DAILY, WEEKLY, or MONTHLY. The PROMPT parameter contains the prompt text for the parameter. Since no LEN keyword is specified and TYPE(*CHAR) is defined, a length of 32 is the default.

Data Type and Parameter Restrictions

The following figure shows the valid combinations of parameters according to the parameter type. An X indicates that the combination is valid, a number refers to a restriction noted at the bottom of the table.

	LEN	RTNVAL	CONSTANT	RSTD	DFT	VALUES	REL	RANGE	SPCVAL	SNGVAL	MIN	MAX	ALWUNPRT	ALWVAR	PGM	DTAARA	FILE	FULL	EXPR	VARY	PASSATR	PASSVAL	DSPINPUT	CHOICE	CHOICEPGM	PMTCTL	PMTCTLPGM	PROMPT
*DEC	X	4	X	X	X	X	X	X	6	1	X	X		X					X		5	X	X	X	X	X	X	X
*LGL	X	4	X	X	X	X			6	1	X	X		X				X	X	5	5	X	X	X	X	X	X	X
*CHAR	X	4	X	X	X	X	X	X	6	1	X	X	X	X	X	X	X	X	X	5	5	X	X	X	X	X	X	X
*NAME	X		X	X	X	X	X	X	6	1	X	X		X	X	X	X	X	X	5	5	X	X	X	X	X	X	X
*SNAME	X		X	X	X	X	X	X	6	1	X	X		X	X	X	X	X	X	5	5	X	X	X	X	X	X	X
*CNAME	X		X	X	X	X	X	X	6	1	X	X		X	X	X	X	X	X	5	5	X	X	X	X	X	X	X
*GENERIC	X		X	X	X	X	X	X	6	1	X	X		X	X	X	X	X	X	5	5	X	X	X	X	X	X	X
*DATE			X	X	X	X	X	X	6	1	X	X		X					X		5	X	X	X	X	X	X	X
*TIME			X	X	X	X	X	X	6	1	X	X		X					X		5	X	X	X	X	X	X	X
*HEX	X		X	X	X	X	X	X	6	1	X	X		X				X	X		5	X	X	X	X	X	X	X
*ZEROELEM											X	X																
*INT2			X	X	X	X	X	X	6	1	X	X		X					X		5	X	X	X	X	X	X	X
*INT4			X	X	X	X	X	X	6	1	X	X		X					X		5	X	X	X	X	X	X	X
*CMDSTR	X		X		X						2	3		X						5	5		X	X	X	X	X	X
*NULL	X										2	3																
stmt label			X		X					X	X	X			X	X	X					7		X	X	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 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 Not valid when the command CPP is a REXX procedure.
5 Parameter is ignored when CPP is a REXX procedure.
6 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.
7 PASSVAL passes a keyword with no blanks or other characters between parentheses when CPP is a REXX procedure.

The next figure shows the valid parameter combinations and restrictions for the PARM, ELEM, and QUAL statements. For example, the intersection of the row for LEN and the column for DFT are blank; therefore, there are no restrictions and combination of LEN(XX) and DFT(XX) is valid. However, the intersection of the row for DFT and the column for CONSTANT contains a 4 which refers to a note at the bottom of the table describing the restriction.

	LEN	RTNVAL	CONSTANT	RSTD	DFT	VALUES	REL	RANGE	SPCVAL	SNGVAL	MIN	MAX	ALWUNPRT	ALWVAR	PGM	DTAARA	FILE	FULL	EXPR	VARY	PASSATR	PASSVAL	DSPINPUT	CHOICE	CHOICEPGM	PMTCTL	PMTCTLPGM	PROMPT
LEN																												
RTNVAL			1	1	1	1	1	1	1	1		2		13	1	1	1	1	1	3	3	15						
CONSTANT		1			4							2							5				20	16		18	18	6
RSTD		1				7	9	9	7	7																		
DFT		1	4								8																	
VALUES		1		7																								
REL		1		9				9																				
RANGE		1		9			9															12						

	LEN	RTNVAL	CONSTANT	RSTD	DFT	VALUES	REL	RANGE	SPCVAL	SNGVAL	MIN	MAX	ALWUNPRT	ALWVAR	PGM	DTAARA	FILE	FULL	EXPR	VARY	PASSATR	PASSVAL	DSPINPUT	CHOICE	CHOICEPGM	PMTCTL	PMTCTLPGM	PROMPT
SPCVAL		1		7																								
SNGVAL		1		7								11																
MIN					8							10										15				19		
MAX		2	2							11	10																	
ALWUNPRT																												
ALWVAR		13																										
PGM		1														14	14											
DTAARA		1													14		14											
FILE		1													14	14												
FULL		1																										
EXPR		1	5																									
VARY		3																			3							
PASSATR		3																		3								
PASSVAL		15						12			15																	
CHOICE			16																						17			
CHOICEPGM																								17				
PMTCTL			18								19																	
PMTCTLPGM			18																									
PROMPT			6																									

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 PASSATR(*YES) and RTNVAL(*YES) are specified, VARY(*YES) must also be specified.
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 The value specified for the MIN parameter must not exceed the value specified for the MAX parameter.
11 Either the MAX value must be greater than 1 or the parameter type must be a statement label, or both.
12 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).
13 If RTNVAL(*YES) is specified, ALWVAR(*NO) cannot be specified.
14 PGM(*YES), DTAARA(*YES), and a value other than *NO for the FILE parameters are mutually exclusive.
15 PASSVAL(*NULL) is not allowed with RTNVAL(*YES) or a value greater than 0 for MIN.
16 The CHOICE and CONSTANT parameters are mutually exclusive.
17 CHOICE(*PGM) requires a name for CHOICEPGM.
18 CONSTANT is mutually exclusive with the PMTCTL and PMTCTLPGM parameters.

19 PMTCTL is not allowed with a value greater than 0 for MIN.
20 CONSTANT is mutually exclusive with DSPINPUT(*NO) and DSPINPUT(*PROMPT).

Defining Lists for Parameters

You can define a parameter to accept a list of values instead of just a single value. You can define the following types of lists:

- A simple list, which allows one or more values of the same type to be specified for a parameter
- A mixed list, which allows a set of separately defined values to be specified for a parameter
- A list within a list, which allows a list to be specified more than once for a parameter or which allows a list to be specified for a value within a mixed list

The following sample command source illustrates the different types of lists:

```
                CMD         PROMPT('Example of lists command')



   /* THE FOLLOWING PARAMETER IS A SIMPLE LIST. IT WILL ACCEPT UP TO   */
   /* 5 NAMES.                                                         */
                PARM        KWD(SIMPLST) TYPE(*NAME) LEN(10) DFT(*ALL) +
                             SPCVAL((*ALL)) MAX(5) PROMPT('Simple list +
                             of up to 5 names')


   /* THE FOLLOWING PARAMETER IS A MIXED LIST OF 3 VALUES, EACH OF A   */
   /* DIFFERENT TYPE AND/OR LENGTH. EACH ELEMENT MAY NOT BE REPEATED.  */
                PARM        KWD(MXDLST) TYPE(MLSPEC) PROMPT('This is a +
                        mixed list of 3 val')
MLSPEC:      ELEM        TYPE(*CHAR) LEN(4) PROMPT('Elem 1 of 3')
             ELEM        TYPE(*DEC) LEN(3 0) PROMPT('Second of three')
             ELEM        TYPE(*CHAR) LEN(10) PROMPT('Last of three +
                           elements')


/* THE FOLLOWING PARAMETER IS A LIST WITHIN A LIST. IT CONTAINS A   */
/* LIST OF UP TO 2 ELEMENTS, WHICH MAY BE REPEATED UP TO 3 TIMES.   */
                PARM        KWD(LWITHINL1) TYPE(LWLSPECA) MAX(3) +
                              PROMPT('Repeatable list of 2 elements')
LWLSPECA:       ELEM          TYPE(*CHAR) LEN(10) PROMPT('1st part of +
                                repeatable list')
                ELEM          TYPE(*DEC) LEN(5 0) PROMPT('2nd part of +
                                repeatable list')


/* THE FOLLOWING PARAMETER IS A LIST WITHIN A LIST. IT CONTAINS A   */
/* LIST OF UP TO 2 ELEMENTS, THE FIRST OF WHICH MAY BE REPEATED     */
/* UP TO 3 TIMES.                                                   */
             PARM        KWD(LWITHINL2) TYPE(LWLSPECB) MAX(1) +
                           PROMPT('Repeated simple within mixed')
LWLSPECB:    ELEM        TYPE(*CHAR) LEN(10) MAX(3) PROMPT('Simple +
                           list within a list')
             ELEM        TYPE(*DEC) LEN(5 0) PROMPT('Single parm +
                           within a list')
```

The following display shows the prompt for the preceding sample command:

```
                          Example of lists command (LSTEXAMPLE)

 Type choices, press Enter.

 Simple list of up to 5 names . . SIMPLST        *ALL
                          + for more values
 This is a mixed list of 3 val    MXDLST
   Elem 1 of 3  . . . . . . . . .
   Second of three  . . . . . . .
   Last of three elements . . . .
 Repeatable list of 2 elements    LWITHINL1
   1st part of repeatable list  .
   2nd part of repeatable list  .
                          + for more values
 Repeatable simple within mixed   LWITHINL2
   Simple list within a list  . .
                          + for more values
   Single parm within a list  . .


                                                                        Bottom
F3=Exit   F4=List   F5=Refresh   F12=Cancel   F13=Prompter help
F24=More keys
```

Defining a Simple List

A simple list can accept one or more values of the type specified by the parameter. For example, if the parameter is for the user name, a simple list means that more than one user name can be specified on that parameter. If a parameter's value is a simple list, you specify the maximum number of elements the list can accept using the MAX parameter on the PARM statement. For a simple list, no command definition statements other than the PARM statement need be specified.

The following example defines a parameter USER for which the display station user can specify up to five user names (a simple list).

```
PARM      KWD(USER) TYPE(*NAME) LEN(10) MIN(0) MAX(5) +
          SPCVAL(*ALL) DFT(*ALL)
```

The parameter is an optional parameter as specified by MIN(0) and the default value is *ALL as specified by DFT(*ALL).

When the elements in a simple list are passed to the command processing program, the format varies depending on whether you are using CL or HLL, or REXX. The following section describes how the elements used in the previous example are passed using CL and HLL. For an explanation of the differences when using REXX, see “Using REXX for Simple Lists” on page 9-18.

Using CL or HLL for Simple Lists

When the command is run using CL or HLL, the elements in a simple list are passed to the command processing program in the following format.

Number of Values Passed	Value	Value	Value	Value

P7730149-0

The number of values passed is specified by a binary value that is two characters long. This number indicates how many values were actually entered (are being passed), not how many can be specified. The values are passed by the type of parameter just as a value of a single parameter is passed (as described under "Defining Parameters" on page 9-7). For example, if two user names (BJONES and TBROWN) are specified for the USER parameter, the following is passed.

0002	BJONES	TBROWN

P7730150-0

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.

Lists are not conveniently processed in CL programs because CL programs do not support binary values, and a binary value needs to be tested to determine how many values are being passed. One alternative is to use the sample commands CVTBINDEC and CVTDECBIN described under "Converting from and to Binary Data" on page 6-36. Another alternative is to process a list in a CL program using the technique shown in the following example. In this example, the parameter value is placed into a character variable that is large enough to contain the largest possible value. (To determine this value, multiply the value of the MAX parameter by the length of the value that can be specified and add 2.) The substring built-in function is used to extract the number of parameters passed. This value is then compared to a character variable that is the number, expressed in hexadecimal format, of all possible values that could be received.

```
        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(&ST) TYPE(*DEC) LEN(3 0) VALUE(1)
        DCL VAR(&CT) TYPE(*DEC) LEN(3 0) VALUE(0)
        DCL VAR(&HEX) TYPE(*CHAR) LEN(12)
           VALUE(X'000000010002000300040005')
        .
        .
        .
LOOP:   IF COND(%SST(&USER 1 2) *EQ %SST(&HEX &ST 2))
           THEN(GOTO CMDLBL(FOUND))
        CHGVAR VAR(&CT) VALUE(&CT + 1)
        CHGVAR VAR(&ST) VALUE(&ST + 2)
        IF COND(&CT *LE 5) THEN(GOTO CMDLBL(LOOP))
        /* If &CT is greater than 5, the value passed is */
        /* greater than the program expects, and error   */
        /* logic should be performed                     */
        .
        .
        .
FOUND:  /* When control is passed here, the value in &CT */
        /* is equal to the number of parameters passed.     */
        /* the value could be zero.                         */
        .
        .
        .
        ENDPGM                                             *
```

Figure 9-1. Simple Lists Example

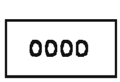
This same technique can be used to process other lists in a CL program.

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.

P7730151-0

*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:



P7730152-0

Using REXX for Simple Lists

When the same command is run, the elements in a simple list are passed to the REXX procedure in the argument string in the following format:

```
. . . USER(value1 value2 . . . valueN) . . .
```

where valueN is the last value in the simple list.

For example, if two user names (BJONES and TBROWN) are specified for the USER parameter, the following is passed:

```
. . . USER(BJONES TBROWN) . . .
```

When a simple list is passed, only the number of elements specified on the command are passed. Therefore, when the CPP processes a simple list, it uses the number of elements passed to determine how many elements can be processed.

Unlike CL programs, lists in REXX programs are conveniently processed. The REXX program in the following example produces the same result as the CL program in Figure 9-1 on page 9-17, but without the complexity:

```
.
.
.
PARSE ARG . 'USER(' user ')' .
.
.
CT = WORDS(user)
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
```

This same procedure can be used to process other lists in a REXX program.

For a simple list, a single value such as *ALL or *NONE can be entered on the command instead of the list. Single values are passed as an individual value. Similarly, if no values are specified for a parameter, the default value, if any is defined, is passed as the only value in the list. For example, if the default value *ALL is used for the USER parameter, the following is passed:

```
. . . USER(*ALL) . . .
```

If no default value is defined for an optional simple list parameter, the following is passed:

```
. . . USER() . . .
```

For more information about REXX procedures, see the *Procedures Language (REXX) Programmer's Guide* and the *Procedures Language (REXX) Reference*.

Defining a Mixed List

A mixed list accepts a set of separately defined values that usually have different meanings, are of different types, and are in a fixed position in the list. For example, LOG(4 0 *SECLVL) could specify a mixed list. The first value, 4, identifies the message level to be logged; the second value, 0, is the lowest message severity to be logged. The third value, *SECLVL, specifies the amount of information to be logged (both first- and second-level messages). If a parameter's value is a mixed list, the elements of the list must be defined separately using an Element (ELEM) statement for each element.

The TYPE parameter on the associated PARM statement must have a label that refers to the first ELEM statement for the list.

```
         PARM    KWD(LOG)    TYPE(LOGLST) ...

LOCLST:  ELEM    TYPE(*INT2)       ...
         ELEM    TYPE(*INT2)       ...
         ELEM    TYPE(*CHAR)       LEN(7)
```

The first ELEM statement is the only ELEM statement that can have a label. Specify the ELEM statements in the order in which the elements occur in the list.

Note that when the MAX parameter has a value greater than 1 on the PARM statement, and the TYPE parameter refers to ELEM statements, the parameter being defined is a list within a list.

Parameters that you can specify on the ELEM statement include TYPE, LEN, CONSTANT, RSTD, DFT, VALUES, REL, RANGE, SPCVAL, SNGVAL, MIN, MAX, ALWUNPRT, ALWVAR, PGM, DTAARA, FILE, FULL, EXPR, VARY, PASSATR, CHOICE, CHOICEPGM, and PROMPT.

In the following example, a parameter CMPVAL is defined for which the display station user can specify a comparison value and a starting position for the comparison (a mixed list).

```
      PARM   KWD(CMPVAL) TYPE(CMP) SNGVAL(*ANY) DFT(*ANY) +
              MIN(0)
CMP:  ELEM   TYPE(*CHAR) LEN(80) MIN(1)
      ELEM   TYPE(*DEC) LEN(2 0) RANGE(1 80) DFT(1)
```

When the elements in a mixed list are passed to the command processing program, the format varies depending on whether you are using CL or HLL, or REXX. The following section describes how the elements used in the previous example are passed using CL and HLL. For an explanation of the differences when using REXX, see "Using REXX for Mixed Lists" on page 9-21.

Using CL or HLL for Mixed Lists

When a command is run using CL or HLL, the elements in a mixed list are passed to the command processing program in the following format:

Number of Values in the Mixed List	Value of Element 1	Value of Element 2		Value of Element n

P7730153-0

The number of values in the mixed list is passed as a binary value of length 2. This value always indicates how many values have been defined for the mixed list, not how many were actually entered on the command. This value may be 1 if the SNGVAL parameter is entered or is passed as the default value. If the user does not enter a value for an element, a default value is passed. The elements are passed by their types just as single parameter values are passed (as described under "Defining Parameters" on page 9-7). 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

P7730154-0

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

P7730155-0

*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, you must use the substring built-in function (see Chapter 2).

In one case, only a binary value of 0000 is passed as the number of values for a mixed list. If no default value is defined on the PARM statement for an optional parameter and the first value of the list is required (MIN(1)), then the parameter itself is not required; but if any element is specified the first element is required. In this case, if the command is entered without specifying a value for the parameter, the following is passed.

0000

P7730152-0

An example of such a parameter is:

```
     PARM  KWD(KWD1)    TYPE(E1) MIN(0)
E1:  ELEM  TYPE(*CHAR)  LEN(10)  MIN(1)
     ELEM  TYPE(*CHAR)  LEN(2)   MIN(0)
```

If this parameter were to be processed by a CL program, the parameter value could be received into a 14-character CL variable. The first 2 characters could be compared to a 2-character variable initialized to hex 0000 using the %SUBSTRING function.

Using REXX for Mixed Lists

When a command is run using REXX, the elements in a mixed list are passed to the command processing program in the following format:

```
. . . CMPVAL(value1 value2 . . . valueN) . . .
```

where `valueN` is the last value in the mixed list.

If the user does not enter a value for an element, a default value is passed. For example, if in the previous example, the user enters a comparison value of QCMDI for the CMPVAL parameter, but does not enter a value for the starting position, whose default value is 1, the following is passed:

```
. . . CMPVAL(QCMDI 1) . . .
```

Note that trailing blanks are not passed with REXX values.

When a display station user enters a single value or when a single value is the default for a mixed list, the value is passed as the first element in the list. For example, if the display station user enters *ANY as a single value for the parameter, the following is passed:

```
. . . CMPVAL(*ANY) . . .
```

Again note that trailing blanks are not passed with REXX values.

If no default value is defined on the PARM statement for an optional parameter and the first value of the list is required (MIN(1)), then the parameter itself is not required. But if any element is specified, the first element is required. In this case, if the command is entered without specifying a value for the parameter, the following is passed:

```
. . . CMPVAL() . . .
```

Defining Lists within Lists

A list within a list can be:

- A list that can be specified more than once for a parameter (simple or mixed list)
- A list that can be specified for a value within a mixed list

The following is an example of lists within a list.

```
STMT((START RESPND) (ADDDSP CONFRM))
```

The outside set of parentheses enclose the list that can be specified for the parameter (the outer list) while each set of inner parentheses encloses a list within a list (an inner list).

In the following example, a mixed list is defined within a simple list. A mixed list is specified, and the MAX value on the PARM statement is greater than 1; therefore, the mixed list can be specified up to the number of times specified on the MAX parameter.

```
        PARM KWD(PARM1) TYPE(LIST1) MAX(5)
LIST1:  ELEM TYPE(*CHAR) LEN(10)
        ELEM TYPE(*DEC) LEN(3 0)
```

In this example, the two elements can be specified up to five times. When a value is entered for this parameter, it could appear as follows:

```
PARM1((VAL1 1.0) (VAR2 2.0) (VAR3 3.0))
```

In the following example, a simple list is specified as a value in a mixed list. In this example, the MAX value on the ELEM statement is greater than 1; therefore, the element can be repeated up to the number of times specified on the MAX parameter.

```
        PARM KWD(PARM2) TYPE(LIST2)
LIST2:  ELEM TYPE(*CHAR) LEN(10) MAX(5)
        ELEM TYPE(*DEC) LEN(3 0)
```

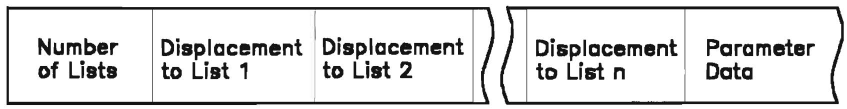
In this example, the first element can be specified up to five times, but the second element can be specified only once. When a value is entered for this parameter, it could appear as follows.

```
PARM2((NAME1 NAME2 NAME3) 123.0)
```

When lists within lists are passed to the command processing program, the format varies depending on whether you are using CL or HLL, or REXX. The following section describes how elements are passed using CL and HLL. For an explanation of the differences when using REXX, see "Using REXX for Lists with Lists" on page 9-25.

Using CL or HLL for Lists within Lists

When a command is run using CL or HLL, a list within a list is passed to the command processing program in the following format:



RSLF177-0

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.

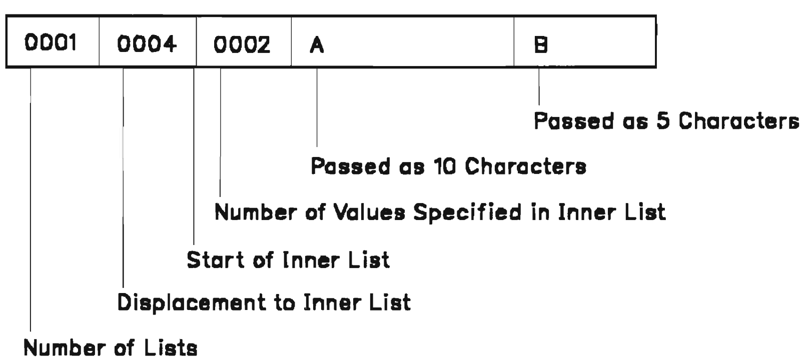
The following example shows a definition for a parameter KWD2 (which is a mixed list within a simple list) how the parameter can be specified by the display station user, and what is passed. The parameter definition is:

```
      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:

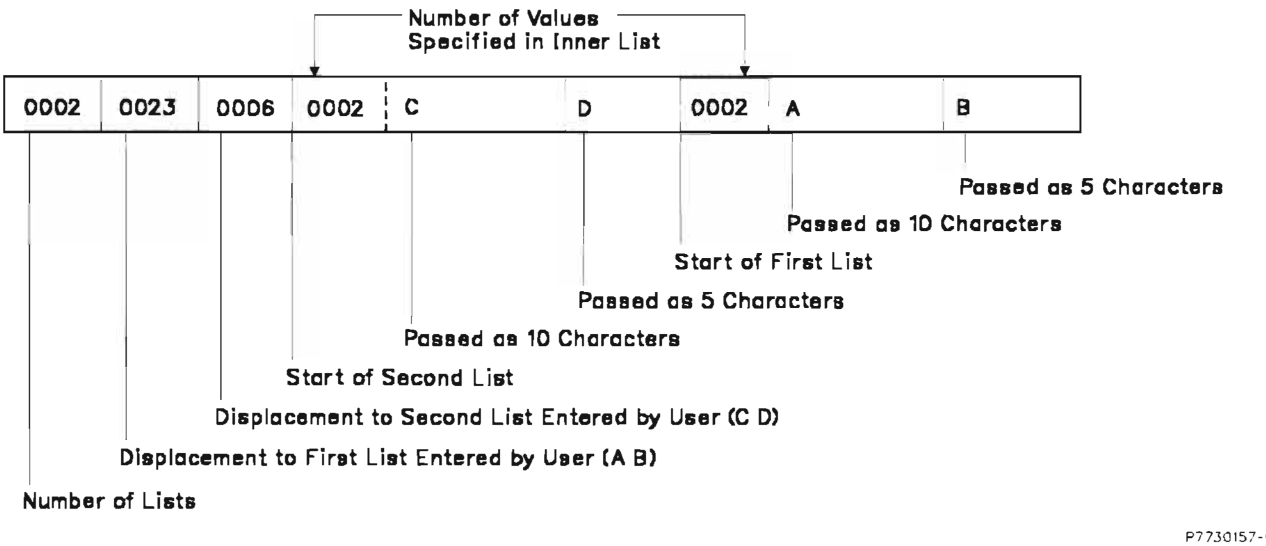


P7730156-0

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.

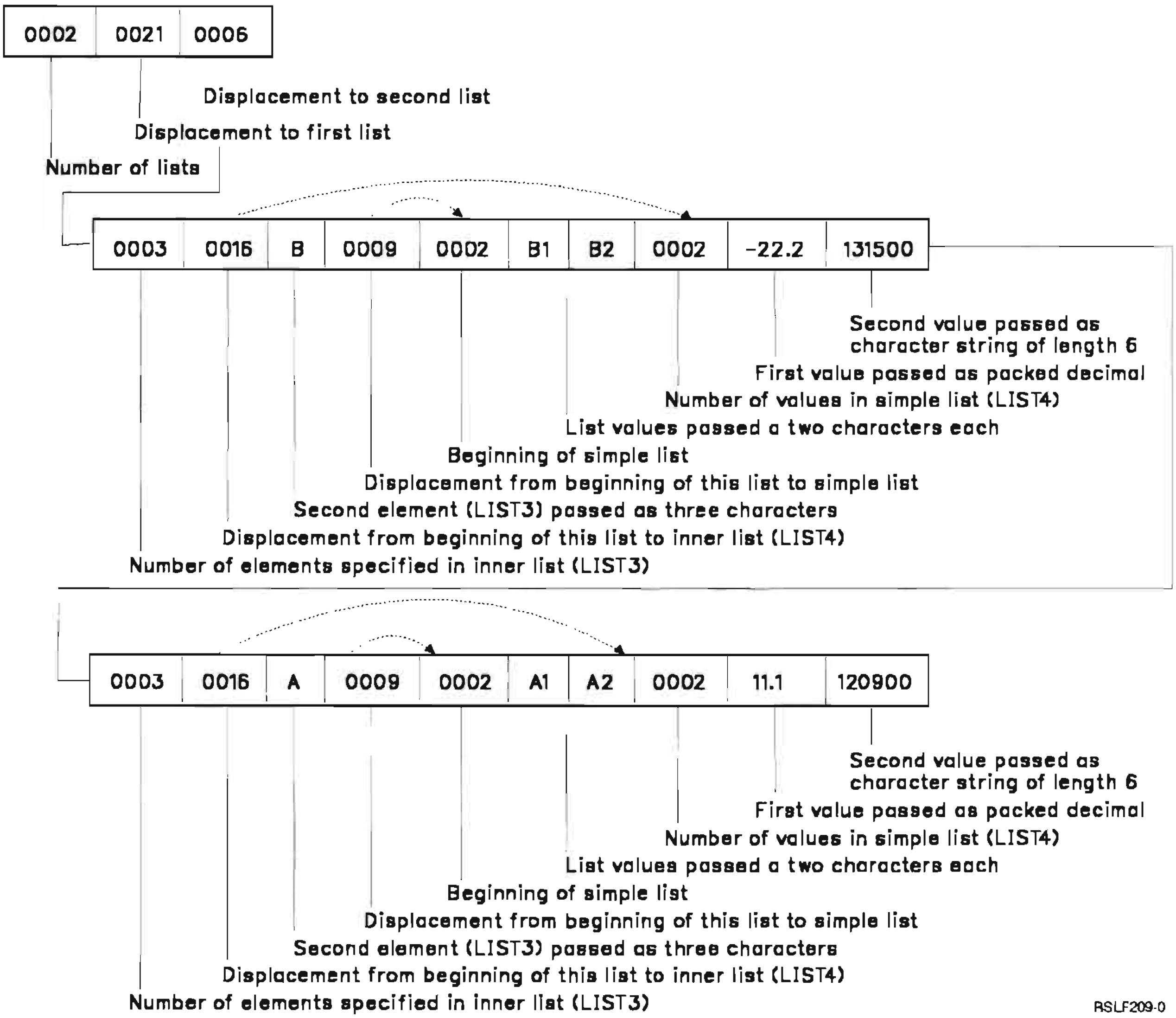
The following is a more complex example of lists within lists. The parameter definition is:

```
        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:

```
PARM1(((11.1 12D900) A (A1 A2)) ((-22.2 131500) B (B1 B2)))
```

the following is passed to the command processing program:



Using REXX for Lists with Lists

When a command is run using REXX, a list within a list is passed to the command processing program just as the values are entered for the parameters. Trailing blanks are not passed.

The following example shows a definition for a parameter KWD2, which is a mixed list within a simple list, how the parameter can be specified by the display station user, and what is passed. The parameter definition is:

```
        PARM   KWD(KWD2)     TYPE(LIST) MAX(20) MIN(0) +
               DFT(*NONE)  SNGVAL(*NONE)
LIST:   ELEM   TYPE(*CHAR)  LEN(10) MIN(1)       /*From value*/
        ELEM   TYPE(*CHAR)  LEN(5) MIN(0)        /*To value*/
```

The display station user enters the KWD2 parameter as:

```
KWD2((A B))
```

The following is passed to the command processing program:

```
KWD2(A B)
```

If the display station user enters the following instead:

```
KWD2((A B) (C D))
```

The following is passed to the command processing program:

```
KWD2((A B) (C D))
```

The following is a more complex example of lists within lists. The parameter definition is:

```
        PARM    KWD(PARM1)  TYPE(LIST3)  MAX(25)
LIST3:  ELEM    TYPE(LIST4)
        ELEM    TYPE(*CHAR)  LEN(3)
        ELEM    TYPE(*NAME)  LEN(2)  MAX(5)
LIST4:  ELEM    TYPE(*DEC) LEN(7 2)
        ELEM    TYPE(*TIME)
```

The display station user enters the PARM1 parameter as:

```
PARM1(((11.1  12D900)  A  (A1  A2))  ((-22.2  131500)  B  (B1  B2)))
```

The following is passed to the command processing program:

```
PARM1(((11.1  12D900)  A  (A1  A2))  ((-22.2  131500)  B  (B1  B2)))
```

Defining a Qualified Name

A qualified name is the name of an object preceded by the name of the library in which the object is stored. If a parameter value or list element is a qualified name, you must define the name separately using Qualifier (QUAL) statements. Each part of the qualified name must be defined with a QUAL statement. The parts of a qualified name must be described in the order in which they occur in the qualified name. You must specify *NAME or *GENERIC in the first QUAL statement. The associated PARM or ELEM statement must identify the label that refers to the first QUAL statement for the qualified name.

The following command definition statements define the most common qualified name. A qualified object consists of the library name which contains an object followed by the name of the object itself. The QUAL statements must appear in the order in which they are to occur in the qualified name.

```
              PARM    KWD(NAME) TYPE(NAME1) SNGVAL(*NONE)...

  NAME1:  QUAL    TYPE(*NAME)
          QUAL    TYPE(*NAME)
```

P7730433-0

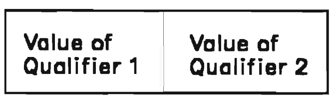
Many of the parameters that can be specified for the QUAL statement are the same as those described for the PARM statement (see “Defining Parameters” on page 9-7). However, only the following values can be specified for the TYPE parameter:

- *NAME
- *GENERIC
- *CHAR
- *INT2
- *INT4

When a qualified name is passed to the command processing program, the format varies depending on whether you are using CL or HLL, or REXX. The following section describes how qualified names are passed using CL and HLL. For an explanation of the differences when using REXX, see "Using REXX for a Qualified Name" on page 9-29.

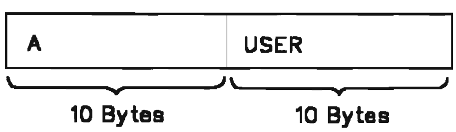
Using CL or HLL for a Qualified Name

A qualified name is passed to the command processing program in the following format when using CL or HLL:



P7730158-0

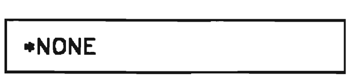
For example, if the display station user enters NAME(USER/A) for the previously defined QUAL statements, the name is passed to the command processing program as follows:



P7730159-0

Qualifiers are passed to the command processing program consecutively by their types and length just as single parameter values are passed (as described under "Defining Parameters" on page 9-7). 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:



P7730160-0

Qualified names can be processed in CL programs using the Substring built-in function as shown in the following example.

The substring built-in function (%SUBSTRING or %SST) is used to separate the qualified name into two values.

```
PGM PARM(&QLFDNAM)
DCL &QLFDNAM TYPE(*CHAR) LEN(20)
DCL &OBJ TYPE(*CHAR) LEN(10)
DCL &LIB TYPE(*CHAR) LEN(10)
CHGVAR &OBJ %SUBSTRING(&QLFDNAM 1 10) /* First 10 */
CHGVAR &LIB %SST(&QLFDNAM 11 10) /* Second 10 */
.
.
.
ENDPGM
```

You can then specify the qualified name in the proper CL syntax. For example, `OBJ(&LIB/&OBJ)`.

You can also separate the qualified name into two values using the following method:

```
PGM PARM(&QLFDNAM)
DCL &QLFDNAM TYPE(*CHAR) LEN(20)
CHKOBJ (%SST(&QLFDNAM 11 10)/%SST(&QLFDNAM 1 10)) *PGM
.
.
.
ENDPGM
```

A simple list of qualified names is passed to the command processing program in the following format:

Number of Qualified Names	**Value 1 Qualifier 1**	**Value 1 Qualifier 2**	**Value 2 Qualifier 1**	**Value 2 Qualifier 2**	. . .

P7730161-0

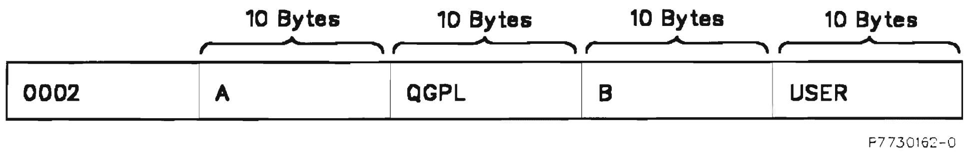
For example, assume that MAX(3) were added as follows to the PARM statement for the NAME parameter.

```
        PARM   KWD(NAME) TYPE(NAME1) SNGVAL(*NONE) MAX(3)
NAME1:  QUAL   TYPE(*NAME)
        QUAL   TYPE(*NAME)
```

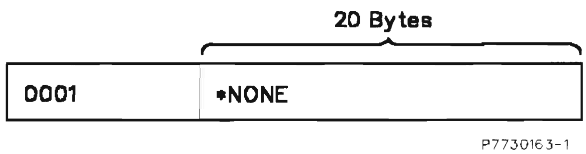
If the display station user enters the following:

```
NAME(QGPL/A USER/B)
```

then the name parameter would be passed to the command processing program as follows.



If the display station user enters the single value NAME(*NONE), the name parameter is passed as follows.



Using REXX for a Qualified Name

When a command is run using REXX, a qualified name is passed to the command processing program just as the value is entered for the parameter. Trailing blanks are not passed.

For example, if a display station user enters the following for the QUAL statements defined previously in this section:

```
NAME(USER/A)
```

the qualified name is passed to the command processing program in the following format:

```
NAME(USER/A)
```

Qualifiers are passed to the command processing program consecutively by their types and length just as single parameter values are passed (as described under "Defining Parameters" on page 9-7).

If the display station user enters *NONE as the single value, the following 20-character value is passed:

```
NAME(*NONE)
```

The following example shows how a display station user would enter a simple list of qualified names:

```
NAME(QGPL/A USER/B)
```

Using REXX, the name parameter would be passed to the command processing program as the following:

```
NAME(QGPL/A USER/B)
```

Defining a Dependent Relationship

If a required relationship exists between parameters and if parameter values must be checked when the command is run, use the Dependent (DEP) statement to define that relationship. Using the DEP statement, you can.

- 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 must be tested if the controlling conditions defined by CTL are true (PARM)
- Specify the number of parameter relationships 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 is to be sent to the display station user if the parameter dependencies have not been satisfied

In the following example, if the display station user specifies the TYPE(LIST) parameter, the display station user must also specify the ELEMLIST parameter.

```
DEP CTL(&TYPE *EQ LIST) PARM(ELEMLIST)
```

In the following example, the parameter &WRITER must never be equal to the parameter &NEWWTR. If this condition is not true, message USR0001 is issued to the display station user.

```
DEP CTL(*ALWAYS) PARM((&WRITER *NE &NEWWTR)) MSGID(USR0001)
```

In the following example, if the display station user specifies the FILE parameter, the display station user must also specify both the VOL and LABEL parameters.

```
DEP CTL(FILE) PARM(VOL LABEL) NBRTRUE(*EQ 2)
```

Possible Choices and Values

The prompter will display possible choices for parameters to the right of the input field on the prompt displays. The text to be displayed can be created automatically, specified in the command definition source, or created dynamically by an exit program. Text describing possible choices can be defined for any PARM, ELEM, or QUAL statement, but because of limitations in the display format, the text will be 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 will be retrieved from the message file specified in the PMTFILE parameter of the CRTCMD command.

You can also specify that an exit program should be 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 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 parameters:

- A 21-byte field containing the following:

Positions	Descriptions
1-10	Command name
11-20	Keyword name
21	The letter C indicates possible choices are to be returned; P indicates a permissible values list is to be returned.

- A 30- or 2000-byte field for returning one of the following:
  - If possible choices text is to be returned (that is, a C is in byte 21 of the first parameter), a 30-byte field into which the program is to place the text.
  - If a permissible values list is to be returned (that is, a P is in byte 21 of the first parameter), a 2000-byte field into which the program is to place the list. The first two bytes of the list must contain the number of entries, in binary, in the list. This value is followed by entries that consist of a two-byte binary length followed by the value, which must be 1 to 32 characters long.

  If a binary zero value is returned in the first two bytes, no permissible values are displayed.

  If a binary negative value is returned in the first two bytes, the list of permissible values is taken from the command.

  If any exception occurs when the program is called, the possible choices text is left blank, and the list of permissible values is taken from the command.

Using Prompt Control

You can control which parameters are displayed for a command during prompting by using prompt control specifications. This control can simplify prompting for a command by displaying only the parameters that you want to see.

You can specify that a parameter be displayed depending on the value specified for other parameters. This specification is useful when a parameter has meaning only when another parameter (called a controlling parameter) has a certain value.

You can also specify that a parameter be selected for prompting only if additional parameters are requested by pressing a function key during prompting. This specification can be used for parameters that are seldom specified by the user, either because the default is normally used or because they control seldom-used functions.

If you want to show all parameters for a command that has prompt control specified, you can request that all parameters be displayed by pressing F9 during prompting.

Conditional Prompting

When prompting the user for a command, a parameter which is conditioned by other parameters will be 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 element or qualifier is compared.

In the following example, parameters OUTFILE and OUTMBR will be selected only if *OUTFILE is specified for the OUTPUT parameter, and parameter OUTQ will be 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(Q2) PMTCTL(PRINT)
OUTFILE: PMTCTL CTL(OUTPUT) COND((*EQ F)) NBRTRUE(*EQ 1)
PRINT:   PMTCTL CTL(OUTPUT) COND((*EQ P)) NBRTRUE(*EQ 1)
```

In the 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 the subsequent PMTCTL statements, the LGLREL 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 will be 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 element 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 will be run during prompting when checking a parameter that is conditioned by the parameter for which the exit program was specified. 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.

If the exit program cannot be found or does not run successfully, a single asterisk is placed in the return value. The exit program should be written to return a value of a single asterisk if it detects an error. You should consider this when coding the PMTCTL statements. For example, you might want to display all parameters conditioned by that parameter if an error occurred.

The exit program must be written to accept three parameters. The parameters are:

- A 20-character field. The prompter will pass 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 will be when passed to the command processing program and should not be changed.
- A 32-character field into which the exit program will place the value to be tested in the PMTCTL statements.

In the following example, OBJ is a qualified name which may be the name of a command, program, or file. The exit program will determine the object type and will return the type in the variable &RTNVAL.

```
PARM OBJ TYPE(Q1) PMTCTLPGM(CNVTYPE)
Q1: QUAL TYPE(*NAME) LEN(10)
    QUAL TYPE(*NAME) LEN(10) SPCVAL(*LIBL) DFT(*LIBL)
PARM CMDPARM TYPE(*CHAR) LEN(10) PMTCTL(CMD)
PARM PGMPARM TYPE(*CHAR) LEN(10) PMTCTL(PGM)
PARM FILEPARM TYPE(*CHAR) LEN(10) PMTCTL(FILE)
CMD: PMTCTL CTL(OBJ) COND((*EQ *CMD) (*EQ *)) NBRTRUE(*EQ 1)
PGM: PMTCTL CTL(OBJ) COND((*EQ *PGM) (*EQ *)) NBRTRUE(*EQ 1)
FILE: PMTCTL CTL(OBJ) COND((*EQ *FILE) (*EQ *)) NBRTRUE(*EQ 1)
```

The source for the exit program is shown here:

```
PGM PARM(&CMD &PARMVAL &RTNVAL)
DCL &CMD *CHAR 20               /* Command and parameter name      */
DCL &PARMVAL *CHAR 20           /* Parameter value                 */
DCL &RTNVAL *CHAR 32            /* Return value                    */
DCL &OBJNAM *CHAR 10            /* Object name                     */
DCL &OBJLIB *CHAR 10            /* Object type                     */
CHGVAR &OBJNAM %SST(&PARMVAL 1 10)
CHGVAR &OBJLIB %SST(&PARMVAL 11 10)
CHGVAR &RTNVAL '*'              /* Initialize return value to error*/
CHKOBJ &OBJLIB/&OBJNAM *CMD     /* See if command exists           */
MONMSG CPF9801 EXEC(GOTO NOTCMD) /* Skip if no command             */
CHGVAR &RTNVAL '*CMD'           /* Indicate object is a command    */
RETURN                          /* Exit                            */
NOTCMD:
CHKOBJ &OBJLIB/&OBJNAM *PGM     /* See if program exists           */
MONMSG CPF9801 EXEC(GOTO NOTPGM) /* Skip if no program             */
CHGVAR &RTNVAL '*PGM'           /* Indicate object is a program    */
RETURN                          /* Exit                            */
NOTPGM:
CHKOBJ &OBJLIB/&OBJNAM *FILE    /* See if file exists              */
MONMSG CPF9801 EXEC(RETURN)     /* Exit if no file                 */
CHGVAR &RTNVAL '*FILE'          /* Indicate object is a file       */
ENDPGM
```

Additional Parameters

You can specify that a parameter which is not frequently used will not be prompted for unless the user requests additional parameters by pressing a function key during prompting. This is done by specifying PMTCTL(*PMTRQS) on the PARM statement for the parameter. When prompting for a command, parameters with PMTCTL(*PMTRQS) coded will not be prompted unless a value was specified for them or the user presses F10 to request the additional parameters.

The prompter displays a separator line before the parameters with PMTCTL(*PMTRQS) to distinguish them from the other parameters. By default, all parameters with PMTCTL(*PMTRQS) are prompted last, even though they are not defined in that order in the command definition source. You can override this by specifying a relative prompt number in the PROMPT keyword. If you do this, however, it can be difficult to see what parameters were added to the prompt when F10 is pressed.

Using Key Parameters and a Prompt Override Program

The prompt override program allows current values rather than defaults to be displayed when a command is prompted.

If a prompt override program is defined for a command, you can see the results of calling the prompt override program in the following two ways:

- Type the name of the command without parameters on any command line and press F4=Prompt. The next screen shows the key parameters for the command. Key parameters are parameters, such as the name of an object, that uniquely identify the object.

  Complete all fields shown and press the Enter key. The next screen shows all command parameters, and the parameter fields that are not key parameter fields contain current values rather than defaults (such as *SAME and *PRV).

  For example, if you type `CHGLIB` on a command line and press F4=Prompt, you see only the Library parameter. If you then type `*CURLIB` and press the Enter key, the current values for your current library are displayed.

- Type the name of the command and the values for all key parameters on any command line. Press F4=Prompt. The next screen shows all command parameters, and the parameter fields that are not key parameter fields will contain current values rather than defaults (such as *SAME and *PRV).

  For example, if you type `CHGLIB LIB(*CURLIB)` on a command line and press F4=Prompt, the current values for your current library are displayed.

When F10=Additional parameters is pressed, any parameters defined with PMTCTL(*PMTRQS) will be displayed with current values. For more information about additional parameters, see "Additional Parameters" on page 9-34.

To exit the command prompt, press F3=Exit.

Procedure for Using Prompt Override Programs

To use a prompt override program, do the following:

1. Specify any parameters that are to be key parameters on the PARM statement in the command definition source. For information about the KEYPARM parameter, see the following section, "Identifying Key Parameters" on page 9-36.
2. Write a prompt override program. For information about creating prompt override programs, see "Writing a Prompt Override Program" on page 9-36.
3. Specify the name of the prompt override program on the PMTOVRPGM parameter when you create or change the command. For information about creating or changing commands that use the prompt override program, see "Specifying the Prompt Override Program When Creating or Changing Commands" on page 9-39.

Identifying Key Parameters

The number of key parameters should be limited to the number of parameters needed to uniquely define the object to be changed.

To ensure a key parameter is coded correctly in the command definition source, do the following:

- Specify KEYPARM(*YES) on the PARM statement in the command definition source.
- Define all parameters that specify KEYPARM(*YES) before all parameters that specify KEYPARM(*NO).

  Note: If a PARM statement specifies KEYPARM(*YES) after a PARM statement that specifies KEYPARM(*NO), the parameter is not treated as a key parameter and a warning message is issued.

- Do not specify a MAX value greater than one in the PARM statement.
- Do not specify a MAX value greater than one for ELEM statements associated with key parameters.
- Do not specify *PMTRQS or a prompt control statement for the PMTCTL keyword on the PARM statement.
- Place key parameters in the command definition source in the same order you want them to appear when prompted.

Writing a Prompt Override Program

A prompt override program needs to be passed certain information to return current values when a command is prompted. You must consider both the passed information and the returned values when you write a prompt override program.

For an example of CL source for a prompt override program, see "CL Sample for Using the Prompt Override Program" on page 9-39.

Information Passed to the Prompt Override Program: The prompt override program is passed the following:

- 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.
- The value for each key parameter, if any. If more than one key parameter is defined, the values are passed in the order that the parameters are defined in the command definition source.
- A 5700-byte space to hold the command string created by the prompt override program. The first two bytes of this field must contain the length of the command string being returned. The actual command string follows the first two bytes.

Information Returned from the Prompt Override Program: Based on the values passed, the prompt override program retrieves the current values for the parameters that are not key parameters. These values are placed into a command string, where the length of the string is determined and returned.

Use the following guidelines to ensure your command string is correctly defined:

- Use the keyword format for the command string just as you would on the command line.
- Do not include the command name and the key parameters in the command string.
- Precede each keyword with a selective prompt character to define how to display the parameter and what value to pass to the CPP. For information about using selective prompt characters, see "Selective Prompting for CL Commands" on page 6-14.

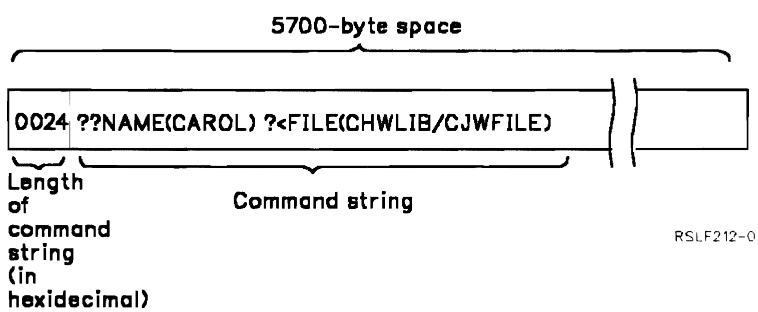
  When using selective prompts, do the following:

  — 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 the length of the command string placed in the first two bytes of the space passed to the program is the length of the actual string and does not include the first two bytes of this space.



- Include only the parameters in the command string whose current values you want displayed when the command is prompted. Parameters not included in the command string have their defaults displayed.
- Use character form for any numbers that appear in the command string. Do not use binary or packed form. Do not include any hexadecimal numbers in the command string.
- Do not put blank spaces between the library and the qualifier or the qualifier and the object. For example:

`??KWD1(library   /object)`	Not valid
`??KWD1(library/  object)`	Not valid
`??KWD1(library/object)`	Valid
`??KWD1(  library/object  )`	Valid

- If you use special values or single values, make sure they are translated into the from-value defined in the command definition source.

For example, a keyword has a special value defined as SPCVAL(*SPECIAL *) in the command definition source. *SPECIAL is the from-value and * is the to-value. When the current value is retrieved for this keyword, * is the value retrieved, but *SPECIAL must appear in the command string returned from the prompt override program. The correct from-value must be placed into the command string since more than one special value or single value can have the same to-value. For example, if `KWD1 SPCVAL((*SPC *) (*SPECIAL *))` is specified, the prompt override program must determine whether * is the to-value for *SPC or *SPECIAL.

- Define the length of fields used to retrieve text as follows:

  ```
  (2*(field length defined in command definition source)) + 2
  ```

  This length allows for the maximum number of quotation marks allowed in the text field. For example, if the TEXT parameter on the CHGxxx command is defined in the command definition source as LEN(50), then the parameter is declared as CHAR(102) in its prompt override program. For an example of how to define the length of fields used to retrieve text, see "CL Sample for Using the Prompt Override Program" on page 9-39.

  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 any embedded single quotation marks are defined as double 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.

Allowing for Errors in a Prompt Override Program: If the prompt override program detects an error, it should do the following:

- Set the command string length to zero so that the defaults rather than current values are displayed when the command is prompted.
- Send a diagnostic message to the previous call.
- 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.

This message can be sent to the calling program of the prompt override program by specifying the following in the prompt override program:

```
SNDPGMMSG    MSGID(USR0012) MSGF(QGPL/ACTMSG) +
             MSGDTA('0000' || &libname) MSGTYPE(*DIAG)
```

After the prompt override program sends all the necessary diagnostic messages, it should then send message CPF0011. To send message CPF0011, use the Send Program Message (SNDPGMMSG) command as follows:

```
SNDPGMMSG    MSGID(CPF0011) MSGF(QCPFMSG) +
             MSGTYPE(*ESCAPE)
```

When message CPF0011 is received, message CPD680A is sent to the calling program and displayed on the prompt screen to indicate that errors have been found. All diagnostic messages are placed in the user's job log.

Specifying the Prompt Override Program When Creating or Changing Commands

To use a prompt override program for a command you want to create, specify the program name when you use the Create Command (CRTCMD) command. You can also specify the program name when you change the command using the Change Command (CHGCMD) command. For both commands, specify the name of the prompt override program on the PMTOVRPGM parameter.

If key parameters are defined in the command definition source but the prompt override program is not specified when the command is created or changed, warning message CPD029B results. The key parameters are ignored, and when the command is prompted, it is displayed using the defaults specified in the command definition source.

Sometimes a prompt override program is specified when a command is created but when no key parameters are defined in the command definition source. In these cases, the prompt override program is called before the command is prompted; informational message CPD029A is sent when the command is created or changed.

CL Sample for Using the Prompt Override Program

The following example shows the command source for a command and the prompt override program. This command allows certain system values to be changed and provides a parameter for date-stamping the change. The prompt override program for this command receives the system value to be changed; retrieves the current value of the system value and the system date; and then places these values into a command string and returns it.

Sample Command Source

```
CHGLIBATR: CMD   PROMPT('Change Library Attributes')
           PARM  KWD(LIB) +
                   TYPE(*CHAR) MIN(1) MAX(1) LEN(10) +
                   KEYPARM(*YES) +
                   PROMPT('Library to be changed')
           PARM  KWD(OWNER) +
                   TYPE(*CHAR) LEN(10) MIN(0) MAX(1) +
                   KEYPARM(*NO) +
                   PROMPT('Library owner')
           PARM  KWD(TEXT) +
                   TYPE(*CHAR) MIN(0) MAX(1) LEN(50) +
                   KEYPARM(*NO) +
                   PROMPT('Text description')
```

Sample Prompt Override Program

```
 PGM PARM(&cmdname &keyparm1 &rtnstring)
/*********************************************************************/
/*                                                                   */
/*  Declarations of parameters passed to the prompt override program */
/*                                                                   */
/*********************************************************************/
 DCL  VAR(&cmdname)   TYPE(*CHAR) LEN(20)
 DCL  VAR(&keyparm1)   TYPE(*CHAR) LEN(10)
 DCL  VAR(&rtnstring) TYPE(*CHAR) LEN(5700)

/******************************************************************/
/*                                                                */
/* Return command string structure declaration                    */
/*                                                                */
/******************************************************************/

                      /* Length of command string generated       */
 DCL  VAR(&stringlen) TYPE(*DEC)  LEN(5 0) VALUE(131)
 DCL  VAR(&binlen)    TYPE(*CHAR) LEN(2)
                      /* OWNER keyword                             */
 DCL  VAR(&ownerkwd)  TYPE(*CHAR) LEN(8)  VALUE('?<OWNER(')
 DCL  VAR(&name)      TYPE(*CHAR) LEN(10)
                      /* TEXT keyword                              */
 DCL  VAR(&textkwd)   TYPE(*CHAR) LEN(8)  VALUE(' ?<TEXT(')
 DCL  VAR(&descript)  TYPE(*CHAR) LEN(102)

/******************************************************************/
/*                                                                */
/* Variables related to command string declarations               */
/*                                                                */
/******************************************************************/
 DCL  VAR(&quote)      TYPE(*CHAR) LEN(1) VALUE('''')
 DCL  VAR(&closparen)  TYPE(*CHAR) LEN(1) VALUE(')')
```

```
/*******************************************************************/
/*                                                                 */
/*                Start of executable 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 single quotation marks in the TEXT description, such as (Carol''s)     */
/*                                                                 */
/*******************************************************************/
    RTVOBJD OBJ(&keyparm1) OBJTYPE(*LIB) OWNER(&name) TEXT(&descript)

/*******************************************************************/
/*                                                                 */
/* Convert the length of the command string from decimal to binary */
/*                                                                 */
/* NOTE:  USERLIB/CVTDECBIN is a user defined command which        */
/*        converts decimal numbers to binary.  See
/*        "CVTBINDEC (Convert from Binary to Decimal) Sample Command" on page 6-36 for an example. */
/*                                                                 */
/*******************************************************************/
    USERLIB/CVTDECBIN FROMDEC(&stringlen) TOBIN(&binlen)

/*******************************************************************/
/*                                                                 */
/* 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:
/*******************************************************************/
/*                                                                 */
/* An error occurred.  Set the command string length to zero.      */
/* Convert the command string length to a binary 0.                */
/*                                                                 */
/* NOTE:  USERLIB/CVTDECBIN is a user defined command which        */
/*        converts decimal numbers to binary.  See Chapter 6,      */
/*        "Advanced Programming Topics" on page 6-1 for an example. */
/*                                                                 */
/*******************************************************************/
    CHGVAR VAR(&stringlen) VALUE(0)
    USERLIB/CVTDECBIN FROMDEC(&stringlen) TOBIN(&binlen)
    CHGVAR VAR(&rtnstring) VALUE(&binlen)

/*******************************************************************/
/*                                                                 */
/* Send error message(s)                                           */
/*                                                                 */
/* NOTE: If you wish to send a diagnostic message as well as CPF0011*/
/*       you will have to enter a valid error message ID in the    */
/*       MSGID parameter and a valid message file in the MSGF      */
/*       parameter for the first SNGPGMMSG command listed below.   */
/*       If you do not wish to send a diagnostic message, do not   */
/*       include the first SNDPGMMSG your program.  However, in    */
/*       error conditions, you must ALWAYS send CPF0011 so the     */
/*       second SNDPGMMSG command must be included in your program. */
/*                                                                 */
/*******************************************************************/
    SNDPGMMSG MSGID(XXXXXXX) MSGF(MSGLIB/MSGFILE) MSGTYPE(*DIAG)
    SNDPGMMSG MSGID(CPF0011) MSGF(QCPFMSG) MSGTYPE(*ESCAPE)

PGMEND:
ENDPGM
```

Creating Commands

After you have defined your command through the command definition statements, you use the Create Command (CRTCMD) command to create the command. Besides specifying the command name, library name, and command processing program name for CL or high-level languages (HLL), or the source member, source file, command environment, and exit program for REXX, you can define the following attributes of the command:

- The validity checking used by the command
- The modes in which the command can be run
  - Production
  - Debug
  - Service

- Where the command can be used
  - Batch job
  - Interactive job
  - CL program in a batch 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 (see Chapter 6, for information about QCMDEXC)
- The maximum number of parameters that can be specified by position
- The message file containing the prompt text
- 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
- The authority given to the public for the command and its description
- Text that briefly describes the command and its function

For commands with REXX CPPs, you can also specify the following:

- The initial command environment to handle commands when the procedure is started
- Exit programs to control running of your procedure

The following example defines a command named ORDENTRY to call an order entry application. The CRTCMD command defines the preceding attributes for ORDENTRY and creates the command using the parameter definitions contained in the member ORDENTRY in the IBM-supplied source file QCMDSRC. ORDENTRY contains the PARM statement used in the example under "Example of Defining a Parameter" on page 9-11.

```
CRTCMD     CMD(DSTPRODLB/ORDENTRY) +
           PGM(*LIBL/ORDENT) +
           TEXT('Calls order entry application')
```

The resulting command is:

```
ORDENTRY  OETYPE(value)
```

where the value can be DAILY, WEEKLY, or MONTHLY.

Once you have created a command, you can:

- Display the attributes of the command by using the Display Command (DSPCMD) command
- Change the attributes of the command by using the Change Command (CHGCMD) command
- Delete the command by using the Delete Command (DLTCMD) command

Command Definition Source Listing

When you create a command, a source list is produced. The following shows a sample source list. The numbers refer to descriptions following the list.

```
      1                                                                                            2            3
5728SS1 R03 M00 861114           Command Definition            DSTPRODLB/ORDENTRY  06/17/88 16:35:46
       Page    1
Command name . . . . . . . . . . . . . . . . :   ORDENTRY
  Library . . . . . . . . . . . . . . . . . . :     DSTPRODLB
Command processing program . . . . . . . . . :   ORDENT     4
  Library . . . . . . . . . . . . . . . . . . :     DSTPRODLB
Source file . . . . . . . . . . . . . . . . . :   QCMDSRC
  Library . . . . . . . . . . . . . . . . . . :     QGPL
Source file member . . . . . . . . . . . . . :   ORDENTRY       06/15/88  16:35:31
Validity checking program . . . . . . . . . . :   *NONE
Mode in which valid . . . . . . . . . . . . . :   *PROD
                                                 *DEBUG
                                                 *SERVICE
Environment allowed . . . . . . . . . . . . . :   *IREXX
                                                 *BREXX
                                                 *BPGM
                                                 *IPGM
                                                 *EXEC
                                                 *INTERACT
                                                 *BATCH
Allow limited user . . . . . . . . . . . . . :   *NO
Max positional parameters . . . . . . . . . . :   *NOMAX
Prompt file . . . . . . . . . . . . . . . . . :   *NONE
Message file . . . . . . . . . . . . . . . . :   QCPFMSG
  Library . . . . . . . . . . . . . . . . . . :     *LIBL
Authority . . . . . . . . . . . . . . . . . . :   *USE
Text  . . . . . . . . . . . . . . . . . . . . :   Calls order entry application
Current library . . . . . . . . . . . . . . . :   *NOCHG
Product library . . . . . . . . . . . . . . . :   *NOCHG
Compiler  . . . . . . . . . . . . . . . . . . :   IBM AS/400 Command Definition Compiler  5
   6                                 Command Definition Source
SEQNBR  *...+... 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ... +... 6 ...+... 7 ...+... 8 ...+... 9 ...+.  DATE  8
   100-        CMD PROMPT('Order Entry Command')          7                                          05/15/88
   200-        PARM   KWD(OETYPE) TYPE(*CHAR) RSTD(*YES) +                                          05/15/88
   300-        VALUES(DAILY WEEKLY MONTHLY) MIN(1) +                                                05/15/88
   400-        PROMPT('Type of order entry:')                                                       05/15/88
* * * * *   E N D   O F   S O U R C E   * * * * *
5728SS1 R08 M00 861114           Command Definition            DSTPRODLB/ORDENTRY     06/15/88 16:35:46        Page 1
                                   Cross Reference
Defined Keywords   9
Keyword              Number          Defined       References
OETYPE                 01              200
                          * * * * *   E N D   O F   C R O S S   R E F E R E N C E   * * * * *
5728SS1 R08 M00 861114           Command Definition            DSTPRODLB/ORDENTRY     06/15/88 16:35:46        Page 2
                                   Final Messages
Message              Sequence
 ID                   Number          Sev        Text
* CPD0205                              30        Command                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, release, modification level, and date of OS/400.

2 The date and time of this run.

3 The page number in the list.

Prologue

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

5 The name of the create command definition compiler.

Source:

6 The sequence number of lines (records) in the source file. A dash following a sequence number indicates that a source statement begins at that sequence number. The absence of a dash indicates that a statement is the continuation of the previous statement. For example, a PARM source statement starts at sequence number 200 and continues at sequence numbers 300 and 400. (Note the continuation character of + on the PARM statement at sequence number 200 and 300.)

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

See the *Database Guide* for information about how sequence numbers are assigned.

7 The source statements.

8 The last date the source statement was changed or added. If the statement has not been changed or added, no date is shown. If the source is not in a database file, the date is omitted.

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

For more information about the command definition errors, see "Errors Encountered when Processing Command Definition Statements" on page 9-46.

Cross-references:

9 The keyword table is a cross-reference list of the keywords validly defined in the command definition. The table lists the keyword, the position of the keyword in the command, the sequence number of the statement where the keyword is defined, and the sequence numbers of statements that refer to the keyword.

If valid labels are defined in the command definition, a cross-reference list of the labels (label table) is provided. The table lists the label, the sequence number of the statement where the label is defined, and the sequence numbers of statements that refer to the label.

Messages:

10 A list of the general error messages not listed in the source section that were encountered during processing of the command definition statements, if any. For each message, this section contains the message identifier, the sequence number of where the error occurred, the severity, and the message.

Message summary:

11 A summary of the number of messages issued during processing of the command definition statements. The total number is given along with totals by severity.

12 A completion message is printed following the message summary.

Errors Encountered when Processing Command Definition Statements

The types of errors that are caught during processing of the command definition statements include syntax errors, references to keywords and labels not defined, and missing statements. The following types of errors detected by the command definition compiler stop the command from being created (severity codes are ignored).

- Value errors
- Syntax errors

Even after an error that stops the command from being created is encountered, the command definition compiler continues to check the source for other errors. Syntax errors and fixed value errors prevent final checking that identifies errors in user names and values or references to keywords or labels. Checking for syntax errors and fixed value errors does continue. This lets you see and correct as many errors as possible before you try to create the command again. To correct errors made in the source statements, see the *Database Guide.*

In the command definition source list, an error condition that relates directly to a specific source statement is listed after that command. See "Command Definition Source Listing" on page 9-44 for an example of these inline messages. Messages that do not relate to a specific source statement but are more general in nature are listed in a messages section of the list, not inline with source statements.

Displaying a Command Definition

You can use the Display Command (DSPCMD) command to display or print the values that were specified as parameters on the CRTCMD command. The DSPCMD command displays the following information for your commands or for IBM-supplied commands:

- Qualified command name. The library name is the name of the library in which the command being displayed is located.
- Qualified name of the command processing program. The library name is the name of the library in which the command processing program resided when the command was created if a library name was specified on the CRTCMD or CHGCMD command. If a library name was not specified, *LIBL is displayed as the library qualifier. If the CPP is a REXX procedure, *REXX is shown.
- Qualified source file name, if the source file was a database file. The library name is the name of the library in which the source file was located when the CRTCMD command was processed. This field is blank if the source file was not a database file.
- Source file member name, if the source file was a database source file.
- If the CPP is a REXX procedure, the following information is shown:
  - REXX procedure member name
  - Qualified REXX source file name where the REXX procedure is located
  - REXX command environment
  - REXX exit programs
- Qualified name of the validity checking. 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.
- Text associated with the command. Blanks are displayed if no text exists for the command.

Effect of Changing the Command Definition of a Command in a Program

When a CL program is created, the command definitions of the commands in the program are used to generate the program. When the CL program is run, the command definitions are also used. If you specify a library name for the command in the CL program, the command must be in the same library at program creation time and at program run time. If you specify *LIBL for the command in the CL program, the command is found, both at program creation and run time, using the library list (*LIBL).

You can make the following changes to the command definition statements for a command without re-creating the 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 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 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 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 program.
- Decrease the number of required list elements 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 proper number and type of parameters.
- Change the name of the validity checking if the new validity checking accepts the proper 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 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 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.
- Add a SNGVAL parameter that was previously a list element.
- Change the name of an optional parameter.
- Remove a value from a list of values.
- Increase the number of required list elements.
- 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.

The following changes can be made to the command definition statements, but may cause the 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.

The following changes to the command definition statements require that the program using the command be re-created.

- Addition of a new required parameter.

- Removal of a required parameter.
- Changing the name of a required parameter.
- Changing a required parameter to a constant.
- Changing the command processing program to or from *REXX

In addition, if you specify *LIBL as the qualifier on the name of the command processing program or the validity checking when the command is created or changed, you can move the command processing program or the validity checking to another library in the library list without changing the command definition statements.

Changing Command Defaults

You can change the default value of a command keyword by using the Change Command Default (CHGCMDDFT) command described in the *CL Reference*. The keyword must have an existing default in order to allow a change to a new default value. The command being changed can be either an IBM-supplied command or a user-written command. The following are recommendations for changing defaults:

1. If the command to be changed is an IBM-supplied command, you should use the Create Duplicate Object (CRTDUPOBJ) command to create a duplicate of the command to be changed 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 and the user library name should be added to the QSYSLIBL system value ahead of QSYS. The changed command will be used system-wide.

2. When a new release of a licensed program is installed, all IBM-supplied commands for the licensed program are replaced on the machine. Therefore, you should use a CL program to make changes to commands so when a new release is installed, 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 used to delete the old version and create the new changed command:

```
PGM
DLTCMD USRQSYS/SIGNOFF
CRTDUPOBJ OBJ(SIGNOFF) FROMLIB(QSYS) OBJTYPE(*CMD) +
          TOLIB(USRQSYS) NEWOBJ(*SAME)
CHGCMDDFT CMD(USRQSYS/SIGNOFF) NEWDFT('LOG(*LIST)')
 .
 .
Repeat the DLTCMD, CRTDUPOBJ and CHGCMDDFT for each
command you want changed
 .
 .
ENDPGM
```

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 to be 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 only parameters, elements, or qualifiers that have existing default values may be changed. If a value is specified for a parameter, element, or qualifier that does not have an existing default value, no default changes will be made.

8. Insert the CHGCMDDFT at the beginning as shown:

   ```
   CHGCMDDFT USRQSYS/CRTCLPGM AUT(*EXCLUDE) +
   TEXT('Isn''t this nice text')
   ```

9. The input for the NEWDFT keyword must be quoted as shown:

   ```
   CHGCMDDFT USRQSYS/CRTCLPGM 'AUT(*EXCLUDE) +
   TEXT('Isn''t this nice text')'
   ```

10. Since there are embedded apostrophes in the NEWDFT value, they must be doubled 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 will be:

    ```
    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.
14. The USRQSYS/CRTCLPGM will have default values of AUT *EXCLUDE and 'Isn''t this nice text' for TEXT.

Example 1

To provide a default value of *NOMAX for the MAXMBRS keyword of command CRTPF, do the following:

```
CRTPF FILE(FILE1) RCDLEN(96) MAXMBRS(1)
  .
  .
CHGCMDDFT CMD(CRTPF) NEWDFT('MAXMBRS(*NOMAX)')
```

Example 2

To provide a default value of 10 for the MAXMBRS keyword of the command CRTPF, do the following:

```
CRTPF FILE(FILE1) RCDLEN(96) MAXMBRS(*NOMAX)
  .
  .
CHGCMDDFT CMD(CRTPF) NEWDFT('MAXMBRS(10)')
```

Example 3

The following allows you to provide a default value of LIB001 for the first qualifier of the SRCFILE keyword and FILE001 for the second qualifier of the SRCFILE keyword for the command CRTCLPGM. The AUT keyword now have a default value of *EXCLUDE.

```
CRTCLPGM PGM(PROGRAM1) SRCFILE(*LIBL/QCMDSRC)
  .
  .
CHGCMDDFT CMD(CRTCLPGM) +
          NEWDFT('SRCFILE(LIB001/FILE001) AUT(*EXCLUDE)')
```

Example 4

The following provides a default value of 'Isn''t this print text' for the PRTTXT keyword of the command CHGJOB. Since the NEWDFT keyword has embedded apostrophes these apostrophes must be doubled to run correctly.

```
CHGJOB PRTTXT('Isn''t this nice text')
  .
  .
CHGCMDDFT CMD(CHGJOB) +
          NEWDFT('PRTTXT(''Isn''''t this print text'')')
```

Example 5

The following provides a default value of QGPL for the first qualifier (library name) of the first list element of the DTAMBRS keyword for the command CRTLF. The new default value for the second list element 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.

Writing a Command Processing Program or Procedure

A command processing program (CPP) can be a CL or HLL program, or a REXX procedure. Programs written in CL or HLL can also be called directly with the CALL CL command. REXX procedures can be called directly using the Start REXX Procedure (STRREXPRC) command. The command processing program does not need to exist when the Create Command (CRTCMD) command is run. If *LIBL is used as the library qualifier, the library list is used to find the command processing program when the created command is run.

Messages issued as a result of running the command processing program can be sent to the job message queue and automatically displayed or printed. You can send displays to the requesting display station.

Notes:

1. The parameters defined on the command are passed individually in the order they were defined (the PARM statement order).
2. Decimal values are passed to HLL and CL programs as packed decimal values of the length specified in the PARM statement.
3. Character, name, and logical values are passed to HLL and CL programs as a character string of the length defined in the PARM statement.

Writing a CL or HLL Command Processing Program

Figure 9-2 on page 9-53 shows the relationship between the Create Command (CRTCMD) command, the command definition statements, and the command processing program.

```
CRTCMD CMD (DSPORD) PGM(DSPORDPGM)
```

Definition for DSPORD Command:

```
CMD     'Display Order'
PARM     KWD (ORDER) +
         TYPE (*DEC) +
         LEN (6 0) +
         RANGE (100000 600000) +
         PROMPT ('Order number :')
```

Command Processing Program for DSPORD Command (DSPORDPGM):

```
PGM (&ORDER)
DCL &ORDER +
      TYPE (*DEC) +
      LEN (6 0)
      RANGE (100000 600000) +
      PROMPT ('Order number :')
•
•
•
```

RSLF213-0

Figure 9-2. Command Relationships for CL and HLL

If the command processing program is a CL program, the variables that receive the parameter values must be declared to correspond to the type and length specified for each PARM statement. The following shows this correspondence. (Note the declare for the parameter ORDER in Figure 9-2.)

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]
*GENERIC	n	*CHAR	≤n[2]
*DATE	7	*CHAR	≤n[2]
*TIME	6	*CHAR	≤n[2]

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 CL program used as a command processing program cannot process binary values (such as *INT2 or *INT4). However, a CL program can receive these values as character fields and then pass the character fields to an HLL program. The HLL program can either process the values or convert them into a format that can be used by a CL program and then return the converted values to the CL program. An example of a command which converts a binary value to a decimal value (CVTBINDEC) is shown in Chapter 6.

For examples of command processing programs, see "Examples of Defining and Creating Commands" on page 9-56.

Writing a REXX Command Processing Procedure

Figure 9-3 shows the relationship between the Create Command (CRTCMD) command, the command definition statements, and the command processing procedure for REXX.

```
CRTCMD CMD(DSPORD) PGM(*REXX) REXSRCMBR(PORDPGM)
```

Definition for DSPORD Command:

```
CMD      'Display Order'
PARM      KWD (ORDER) +
          TYPE (*DEC) +
          LEN (6 0) +
          RANGE (100000 600000) +
          PROMPT ('Order number :')
```

Command Processing Procedure for DSPORD Command (DSPORDPGM):

```
REM Parse arg . 'ORDER('order')'.

  •
  •
  •
```

RSLF211-0

Figure 9-3. Command Relationships for REXX

Writing a Validity Checking Program

If you write a validity checking program for your command, specify the name of the validity checking program on the VLDCKR parameter on the Create Command (CRTCMD) command. The program does not have to exist when the CRTCMD command is run. If *LIBL is used as the library qualifier, the library list is used to find the validity checking program when the created command is run.

The 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. This section describes how to send messages from a validity checking program that is a CL program to the system.

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:

```
PGM PARM(&PARM01)
DCL VAR(&PARM01) TYPE(*CHAR) LEN(10)
IF COND(&PARM01 *EQ 'ERROR') THEN(DO)
SNDPGMMSG MSGID(CPD0006) MSGF(QCPFMSG) +
          MSGDTA('0000 DIAGNOSTIC MESSAGE FROM USER-DEFINED +
          VALIDITY CHECKER INDICATING THAT PARM01 IS IN ERROR.') +
          MSGTYPE(*DIAG)
SNDPGMMSG MSGID(CPF0002) MSGF(QCPFMSG) MSGTYPE(*ESCAPE)
ENDDO
ELSE
.
.
.
ENDPGM
```

Examples of Defining and Creating Commands

This section contains examples of defining and creating commands.

Calling Application Programs

You can create commands to call application programs. If you create a command to call an application program, OS/400 performs validity checking on the parameters passed to the program. However, if you use the CALL command to call an application program, the application program must perform the validity checking.

For example, a label writing program (LBLWRT) writes any number of labels for a specific customer on either 1- or 2-part forms. When the LBLWRT program is run, it requires three parameters: the customer number, the number of labels, and the type of form to be used (ONE or TWO).

If the program were called directly from the display, the second parameter would be in the wrong format for the program. A numeric constant on the CALL command is always 15 digits with 5 decimal positions, and the LBLWRT program expects a 3-digit number with no decimal positions. A command can be created that provides the data in the format required by the program.

The command definition statements for a command to call the LBLWRT program are:

```
CMD PROMPT('Label Writing Program')
PARM  KWD(CUSNBR) TYPE(*CHAR) LEN(5) MIN(1) +
      PROMPT('Customer Number')
PARM  KWD(COUNT) TYPE(*DEC) LEN(3) DFT(20) RANGE(10 150) +
      PROMPT('Number of Labels')
PARM  KWD(FRMTYP) TYPE(*CHAR) LEN(3) DFT('TWO') RSTD(*YES) +
      SPCVAL(('ONE') ('TWO') ('1' 'ONE') ('2' 'TWO')) +
      PROMPT('Form Type')
```

For the second parameter, COUNT, a default value of 20 is specified and the RANGE parameter allows only values from 10 to 150 to be entered for the number of labels.

For the third parameter, FRMTYP, the SPCVAL parameter allows the display station user to enter 'ONE', 'TWO', '1', or '2' for this parameter. The program expects the value 'ONE' or 'TWO'; however, if the display station user enters '1' or '2', the command makes the necessary substitution for the FRMTYP parameter.

The command processing program for this command is the application program LBLWRT. If the application program were an RPG/400 program, the following specifications would be made in the program to receive the parameters:

```
*ENTRY    PLIST
          PARM     CUST   5
          PARM     COUNT  30
          PARM     FORM   3
```

The CRTCMD command is:

```
CRTCMD CMD(LBLWRT) PGM(LBLWRT) SRCMBR(LBLWRT)
```

Substituting a Default Value

You can create a command that provides defaults for an IBM-supplied command and reduces the entries that the display station user must make. For example, you could create a Save Library on Tape (SAVLIBTAP) command that initializes a tape and saves a library on the tape device TAPE1. This command provides defaults for the standard Save Library (SAVLIB) command parameters and requires the display station user to specify only the library name.

The command definition statements for the SAVLIBTAP command are:

```
CMD PROMPT('Save Library to Tape')
PARM KWD(LIB) TYPE(*NAME) LEN(10) MIN(1) +
  PROMPT('Library Name')
```

The command processing program is:

```
PGM PARM(&LIB)
DCL &LIB TYPE(*CHAR) LEN(10)
INZTAP DEV(TAPE1) CHECK(*NO)
SAVLIB LIB(&LIB) DEV(TAPE1)
ENDPGM
```

The CRTCMD command is:

```
CRTCMD CMD(SAVLIBTAP) PGM(SAVLIBTAP) SRCMBR(SAVLIBTAP)
```

Displaying an Output Queue

You can create a command to display an output queue that defaults to display the output queue PGMR. The following command, DSPOQ, also allows the display station user to display any queue on the library list and provides a print option.

The command definition statements for the DSPOQ command are:

```
CMD PROMPT('WRKOUTQ.-Default to PGMR')
PARM  KWD(OUTQ) TYPE(*NAME) LEN(10) DFT(PGMR) +
      PROMPT('Output queue:')
PARM  KWD(OUTPUT) TYPE(*CHAR) LEN(6) DFT(*) RSTD(*YES)
      VALUES(* *PRINT) PROMPT ('Output')
```

The RSTD parameter on the second PARM statement specifies that the entry can only be one of the list of values.

The command processing program for the DSPOQ command is:

```
PGM PARM(&OUTQ &OUTPUT)
DCL &OUTQ TYPE(*CHAR) LEN(10)
DCL &OUTPUT TYPE(*CHAR) LEN(6)
WRKOUTQ OUTQ(*LIBL/&OUTQ) OUTPUT(&OUTPUT)
ENDPGM
```

The CRTCMD command is:

```
CRTCMD CMD(DSPOQ) PGM(DSPOQ) SRCMBR(DSPOQ)
```

The following command, DSPOQ1, is a variation of the preceding command. This command allows the work station user to enter a qualified name for the output queue name, and the command defaults to *LIBL for the library name.

The command definition statements for the DSPOQ1 command are:

```
        CMD PROMPT('WRKOUTQ.-Default to PGMR')
        PARM    KWD(OUTQ) TYPE(QUAL1) +
                PROMPT('Output queue:')
        PARM    KWD(OUTPUT) TYPE(*CHAR) LEN(6) RSTD(*YES) +
                VALUES(* *PRINT) DFT(*) +
                PROMPT('Output')
QUAL1:  QUAL TYPE(*NAME) LEN(10) DFT(PGMR)
        QUAL TYPE(*NAME) LEN(10) DFT(*LIBL) +
        SPCVAL(*LIBL)
```

The QUAL statements are used to define the qualified name that the user can enter for the OUTQ parameter. If the user does not enter a name, *LIBL/PGMR is used. The SPCVAL parameter is used because any library name must follow the rules for a valid name (for example, begin with A through Z), and the value *LIBL breaks these rules. The SPCVAL parameter specifies that if *LIBL is entered, OS/400 is to ignore the name validation rules.

The command processing program for the DSPOQ1 command is:

```
PGM PARM(&OUTQ &OUTPUT)
DCL &OUTQ TYPE(*CHAR) LEN(20)
DCL &OBJNAM TYPE(*CHAR) LEN(10)
DCL &LIB TYPE(*CHAR) LEN(10)
DCL &OUTPUT TYPE(*CHAR) LEN(6)
CHGVAR &OBJNAM %SUBSTRING(&OUTQ 1 10)
CHGVAR &LIB %SUBSTRING(&OUTQ 11 10)
WRKOUTQ OUTQ(&LIB/&OBJNAM) OUTPUT(&OUTPUT)
ENDPGM
```

Because a qualified name is passed from a command as a 20-character variable, the substring built-in function (%SUBSTRING or %SST) must be used in this program to put the qualified name in the proper CL syntax.

Displaying Messages from IBM Commands More Than Once

The CLROUTQ command issues the completion message CPF3417, which describes the number of entries deleted, the number not deleted, and the name of the output queue. If the CLROUTQ command is run within a CPP, the message is still issued but it becomes a detailed message because it is not issued directly by the CPP. For example, if a user-defined CLROUTQ command was issued from the Programmer Menu, the message would not be displayed. You can, however, receive an IBM message and reissue it from your CPP.

For example, you create a command named CQ2 to clear the output queue QPRINT2.

The command definition statements for the CQ2 command are:

```
CMD PROMPT ('Clear QPRINT2 output queue')
```

The CRTCMD command is:

```
CRTCMD CMD(CQ2) PGM(CQ2)
```

The CPP, which receives the completion message and displays it, is as follows:

```
PGM /* Clear QPRINT2 output queue CPP */
DCL &MSGID TYPE(*CHAR) LEN(7)
DCL &MSGDTA TYPE(*CHAR) LEN(100)
CLROUTQ QPRINT2
RCVMSG MSGID(&MSGID) MSGDTA(&MSGDTA)
SNDPGMMSG MSGID(&MSGID) MSGF(QCPFMSG) MSGDTA(&MSGDTA) MSGTYPE(*COMP)
ENDPGM
```

The MSGDTA length for message CPF3417 is 28 bytes. However, by defining the variable &MSGDTA as 100 bytes, the same approach can be used on most messages because any unused positions are ignored.

Creating Abbreviated Commands

Example 1

You can create your own commands to simplify or abbreviate IBM-supplied commands. For example, to reduce the entries necessary for the Work with System Status (WRKSYSSTS) command, you could create a command called DX.

The command definition source statement is:

```
CMD /* Shortened WRKSYSSTS command */
```

The command processing program is:

```
PGM
WRKSYSSTS OUTPUT(*)
ENDPGM
```

The CRTCMD command is:

```
CRTCMD CMD(DX) PGM(DX) SRCMBR(DX)
```

Example 2

You could create an abbreviated command called DW1 to start the printer writer W1.

The command definition statement is:

```
CMD /* Start printer writer command */
```

The command processing program is:

```
PGM
STRPRTWTR DEV(QSYSPRT) OUTQ(QPRINT) WTR(W1)
ENDPGM
```

The CRTCMD command is:

```
CRTCMD CMD(DW1) PGM(DW1) SRCMBR(DW1)
```

Adding or Subtracting a Value to a Date

You can create a command that adds or subtracts a user-specified number of days from a date and returns the new date as a variable. The following sample command can be used to establish ending dates for save/restore operations or to schedule various functions.

The Add Date (ADDDAT) sample command supports the following parameters:

DAYS
: This is the number to be added to or subtracted from the DATE parameter. For subtraction, enter a minus number.

TOVAR
: This is the name of the return variable that will contain the new date in system format.

DATE
: This is the date in system format that the number in the DAYS parameter is added to or subtracted from. The default is *TODAY, meaning the current system date.

Perform the following steps to create the ADDDAT sample command:

1. Enter the command definition statements and create the command ADDDAT, specifying the parameters PGM(ADDDATC) and ALLOW(*IPGM *BPGM). The ALLOW parameter is required because of the return variable used.

```
SEQNBR *... ... 1 ... ... 2 ... ... 3 ... ... 4 ... ... 5 ... ... 6 ... ... 7 ..

 01.00             CMD        PROMPT('Add Date')
 02.00                        /* CPP is ADDDATC */
 03.00             PARM       KWD(DAYS) TYPE(*DEC) LENS(5)  RANGE(-35000 +
 04.00                          35000) MIN(1) PROMPT('Number of  days to +
 05.00                          add or sub:')
 06.00             PARM       KWD(TOVAR) TYPE(*CHAR) LEN(6)  RTNVAL(*YES) +
 07.00                          MIN(1) PROMPT('New date variable:')
 08.00             PARM       KWD(DATE) TYPE(*DEC) LEN(6 0) +
 09.00                          RANGE(000000 999999) +
 10.00                         DFT(*TODAY) SPCVAL((*TODAY 0)) +
 11.00                         PROMPT('Date (sys fmt) or *TODAY:')
```

2. Enter the CL program source and create the CL program ADDDATC:

```
SEQNBR *... ... 1 ... ... 2 ... ... 3 ... ... 4 ... ... 5 ... ... 6 ... ... 7 ..

                   PGM        PARM(&DAYS &TOVAR &DATE)
                   DCL        &DAYS *DEC LEN(5 0)
                   DCL        &TOVAR *CHAR LEN(6)
                   DCL        &DATE *DEC LEN(6 0)
                   DCL        &WRKDAT *CHAR LEN(6)
                   DCL        &JULIANA *CHAR LEN(5)
                   DCL        &YRD *DEC LEN(2 0)
                   DCL        &DAYSD *DEC LEN(3 0)
                   DCL        &LEAP *DEC LEN(2 0)
                   DCL        &DAYSINYEAR *DEC LEN(3 0)
                   DCL        &NUM5 *DEC LEN(5)
                   DCL        &NUM2 *DEC LEN(2)
                              /* &DATE=0 is special value *TODAY */
                   IF         (&DATE *EQ 0) RTVJOBA DATE(&WRKDAT)
                   IF         (&DATE *NE 0) CHGVAR &WRKDAT &DATE
                   CVTDAT     DATE(&WRKDAT) TOVAR(&JULIANA) TOFMT(*JUL) +
                                TOSEP(*NONE) /* Convert to Julian */
                  MONMSG     MSGID(CPF0555) EXEC(SNDPGMMSG MSGID(CPF0006) +
                               MSGF(QCPFMSG) MSGDTA('0000' || 'DATE +
                               parameter value cannot be converted') +
                               MSGTYPE(*ESCAPE))
                              /* Substring for year and day */
                   CHGVAR     &YRD %SST(&JULIANA 1 2)
                   CHGVAR     &DAYSD %SST(&JULIANA 3 3)
                   CHGVAR     VAR(&NUM5) VALUE(&DAYSD + &DAYS) /* Add days */
      CHKPLUS:     IF         (&NUM5 *GT 0) GOTO CHKLEAP /* If positive */
                   IF         (&YRD *EQ 00) CHGVAR &YRD 99 /*  Year 2000 */
                   ELSE       CHGVAR &YRD (&YRD -1) /* Decrement year */
      CHKLEAP:     CHGVAR     &NUM2 (&YRD / 4) /* Chk leap year */
                   CHGVAR     &LEAP (&YRD - (&NUM2 * 4))
                   IF         (&LEAP *GT 0) CHGVAR &DAYSINYEAR 365
                   ELSE       CHGVAR &DAYSINYEAR 366 /* Leap year */
                   IF         (&NUM5 *LE 0) DO /* Days are negative */
                   CHGVAR     &NUM5 (&NUM5 + &DAYSINYEAR)
                   GOTO       CHKPLUS /* Check for positive days */
                   ENDDO      /* End negative days */
                   IF         (&NUM5 *GT &DAYSINYEAR) DO /* Ovfl */
                   IF         (&YRD *EQ 99) CHGVAR &YRD -1 /* Year 2000 */
                   CHGVAR     &YRD (&YRD + 1)  /* Bump year */
                   CHGVAR     &NUM5 (&NUM5 - &DAYSINYEAR) /* Subtract */
                   GOTO       CHKLEAP /* Test for next year */
                   ENDDO      /* End days greater than days-in-year */
                   CHGVAR     &DAYSD &NUM5  /* Chg to 3 digits */
                                   /* Substring back into Julian date */
                   CHGVAR     %SST(&JULIANA 1 2) &YRD
                   CHGVAR     %SST(&JULIANA 3 3) &DAYSD
                   CVTDAT     DATE(&JULIANA) TOVAR(&TOVAR) FROMFMT(*JUL) +
                                TOSEP(*NONE) /* Convert to sys fmt */
                   ENDPGM
```

The ADDDAT command must be used in a CL program. The command is intended to provide dates for such things as the ending date (EXPDATE) parameter. In the following program, an ending date of 28 days plus the current date is used as the value for the EXPDATE parameter:

```
PGM
DCL &EXPDATE TYPE(*CHAR) LEN(6)
ADDDAT DAYS(28) TOVAR(&EXPDATE)
SAVOBJ OBJ(FILEX) EXPDATE(&EXPDATE)
ENDPGM
```

Deleting Program Objects

You can create a command to delete HLL programs and their corresponding source members.

The command definition statements for the command named DPS are:

```
CMD PROMPT ('Delete Program and Source')
PARM KWD(PGM) TYPE(*NAME) LEN(10) PROMPT('Program Name')
```

The command processing program is written assuming that the name of the program and the source file member are the same and that the IBM-supplied source files (QCLSRC, QRPGSRC, and QCBLSRC) have been used. The program also assumes that both the program and the source file are on the library list. If the program cannot be deleted, an information message is sent, and the command attempts to remove the source member. If the source member does not exist, an escape message is sent. The command processing program is:

```
PGM PARM(&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
```

Deleting Files and Source Members

You can create a command to delete files and their corresponding source members in QDDSSRC.

The command definition statements for the command named DFS are:

```
CMD PROMPT('Delete File and Source')
PARM KWD(FILE) TYPE(*NAME) LEN(10) PROMPT('File Name')
```

The command processing program is written assuming that the name of the file and the source file member are the same. The program also assumes that both the file and the source file are on the library list. If the program cannot delete the file, an information message is sent and the command attempts to remove the source member. If the source member does not exist, an escape message is sent.

The command processing program is:

```
PGM PARM(&FILE)
DCL &FILE TYPE(*CHAR) LEN(10)
DCL &MSGID TYPE(*CHAR) LEN(7)
DCL &MSGDTA TYPE(*CHAR) LEN(80)
DCL &SRCFILE TYPE(*CHAR) LEN(10)
MONMSG MSGID(CPF0000) EXEC(GOTO ERROR) /* CATCH ALL */
DLTF &FILE
MONMSG MSGID(CPF2105) EXEC(DO) /* NOT FOUND */
RCVMSG MSGTYPE(*EXCP) MSGID(&MSGID) MSGDTA(&MSGDTA)
SNDPGMMSG MSGID(&MSGID) MSGF(QCPFMSG) MSGTYPE(*INFO) +
      MSGDTA(&MSGDTA)
GOTO TRYDDS
ENDDO
RCVMSG MSGTYPE(*COMP) MSGID(&MSGID) MSGDTA(&MSGDTA)
    /* DELETE FILE COMPLETED */
SNDPGMMSG MSGID(&MSGID) MSGF(QCPFMSG) MSGTYPE(*COMP) +
          MSGDTA(&MSGDTA) /* TRY IN QDDSSRC FILE */
TRYDDS:   CHKOBJ QDDSSRC OBJTYPE(*FILE) MBR(&FILE)
          RMVM QDDSSRC MBR(&FILE)
          CHGVAR &SRCFILE 'QDDSSRC'
          GOTO END
END:      RCVMSG MSGTYPE(*COMP) MSGID(&MSGID) MSGDTA(&MSGDTA)
              /* REMOVE MEMBER COMPLETED */
          SNDPGMMSG MSGID(&MSGID) MSGF(QCPFMSG) MSGTYPE(*COMP) +
          MSGDTA(&MSGDTA)
          RETURN
ERROR:    RCVMSG MSGTYPE(*EXCP) MSGID(&MSGID) MSGDTA(&MSGDTA)
              /* ESCAPE MESSAGE */
          SNDPGMMSG MSGID(&MSGID) MSGF(QCPFMSG) MSGTYPE(*ESCAPE) +
               MSGDTA(&MSGDTA)
ENDPGM
```

Chapter 10. User-Defined Menus

The AS/400 system uses two kinds of menus: IBM-supplied (system) menus and user-defined menus. This chapter is primarily concerned with user-defined menus: what they are, how to create them, and how to use them with system menus.

- IBM-Supplied Menus

  IBM-supplied menus consist of those menus created by IBM and shipped with the system. These menus consist of *MENU object types only.

  Some menu options associated with IBM-supplied menus are not allowed or not active for some users of the system. Generally, these options are not displayed to those users.

  Only the following changes can be made by the user to IBM-supplied menus:

  - The current library (CURLIB parameter)
  - The product library (PRDLIB parameter)
  - The text description of the menu (TEXT parameter)

  See the *New User's Guide* for more information about AS/400 displays.

- User-Defined Menus

  Two types of user-defined menus can be created using the information in this chapter. Both types consist of *MENU object types and must be created using the Create Menu (CRTMNU) command.

  - *Display file menus* use a display defined by data description specifications (DDS) to present a user-defined menu. The menu functions are controlled by a message file containing the commands used to run each of the menu options.

  - *Program menus* use a high-level language program to display the menu and provide the functions necessary to run the menu options.

  The screen design aid (SDA) utility can also be used to create display file menus like those described in this chapter. Program menus to be used for the AS/400 environment cannot be created using SDA. See the *SDA User's Guide/Reference* for more information.

Running Menus and Displays

The Go to Menu (GO) command is used to run menus and displays stored as *MENU object types. The GO command allows the user to specify either a particular menu or a generic menu name.

Note: If a generic menu name is specified, the user is shown the Work with Menus display, showing those menus included by the generic specification. From this list the user may choose the menu to run.

The user may optionally specify whether or not to return to the menu from which the command is entered after running the menu specified by the GO command.

You must have *USE authority for the menu and its display and message files or program (whichever applies) to use this command. You must also have *USE authority for the library where the menu is located. See *Security Concepts and Planning* for a detailed discussion of object authority.

Return Points

The optional RTNPNT parameter on the GO command specifies whether or not the menu or display from which the command is entered is returned to by pressing the Exit key. If RTNPNT is not specified, RTNPNT(*YES) is used as the default value and the display from which the command is entered is used as a return point.

IBM-supplied menus with options to select other menus also use the GO command to run the selected menu option. The specification of the RTNPNT parameter as *YES or *NO for these menu options determines whether or not the menu is returned to when the user presses the Exit key.

Specifying the GO command with RTNPNT(*YES) can therefore be done in two ways:

1. By typing the command on the command line or
2. By selecting a menu option which implicitly uses a GO command with RTNPNT(*YES).

In either case, *the menu from which the command was entered* (or implied) becomes a return point which can be returned to by pressing the Exit key.

The Menu Stack

A menu is placed on an internal menu stack before it is run. If a stack is not available for the menu, one is created. When the Cancel key is pressed for a menu, the previous menu in the stack is shown. Each menu stack is 10 elements (menus) deep. When the 11th menu is placed on the menu stack, the first, or oldest, menu is removed from the stack. This menu cannot be returned to by using the Cancel key.

Using the Cancel and Exit Keys

In the following example, a series of menus is presented using both the menu options and the GO command.

1. From the AS/400 Main Menu, `GO PROGRAM RTNPNT(*YES)` is typed on the command line. Specifying RTNPNT(*YES) here means that the Main Menu will be used as a return point.

```
MAIN                              Main Menu
                                                          System    XXXXXXXX
Select one of the following:

       1. User tasks
       2. Office tasks
       3. General system tasks
       4. Files, libraries, and folders
       5. Programming
       6. Communications
       7. Define or change the system
       8. Problem handling
       9. Display a menu

      90. Sign off




Selection or Command
===> GO PROGRAM RTNPNT(*YES)

F3=Exit   F4=Prompt   F9=Retrieve   F12=Cancel   F13=User support
F23=Set initial menu
```

2. Next, by typing `GO PROBLEM RTNPNT(*NO)` on the command line of the Programming menu and pressing the Enter key, the Problem Handling menu is shown. The Programming menu is *not* set as a return point.

```
PROGRAM                          Programming
                                                          System    XXXXXXXX
Select one of the following:

       1. Programmer's menu
       2. Programming Development Manager
       3. Utilities
       4. Programming language debug
       5. SQL pre-compiler
       6. Question and answer
       7. IBM product information
       8. Copy screen image

      50. System/36 programming

      70. Related commands


Selection or Command
===> GO PROBLEM RTNPNT(*NO)

F3=Exit   F4=Prompt   F9=Retrieve   F12=Cancel   F13=User support
F16=System main menu
```

```
PROBLEM                        Problem Handling
                                                              System    XXXXXXXX
Select one of the following:

     1. Question and answer
     2. Work with problems
     3. Network problem handling
     4. Display system operator messages
     5. Display the history log
     6. System service tools

    60. More problem handling options

    70. Related commands




Selection or Command
===>

F3=Exit   F4=Prompt   F9=Retrieve   F12=Cancel   F13=User support
F16=System main menu
```

3. In this example, if the Cancel key is pressed while viewing the Problem Handling menu, the user is returned to the Programming menu because the Programming menu is the preceding display on the menu stack. If F3 (Exit) is pressed while viewing the Problem Handling menu, the user is returned to the Main Menu, because the Main Menu is the most recent display from which a GO command was entered with RTNPNT(*YES).

Note that since the default value for the RTNPNT parameter is *YES, `GO PROBLEM` will have the same effect as `GO PROBLEM RTNPNT(*YES)`.

Pressing either F3 (Exit) or F12 (Cancel) while viewing the help for a menu returns the user to the menu itself.

Initial Menus

F23 can be used to set the initial menu attribute in the user's user profile. Use of this key is allowed only for those users who do not have a required menu specified in their user profile by the limit capabilities keyword (LMTCPB(*YES)).

Pressing F23 sets the system or user menu being shown at the time the key is pressed as the user's initial menu. Unless the user's initial program requests some other menu, the menu selected by pressing F23 is shown as the initial menu the next time the user signs on.

See the *New User's Guide* for a detailed discussion of function keys and the *Security Concepts and Planning* for information about initial menus and limited users.

Display File Menus

Display file menus consist of two principal parts:

- A DDS-defined display file to define the way the menu looks at the display station
- A message file to define what action is taken when any of the menu options is selected

Help for the menu options may also be provided using DDS. See "Help for Menus" on page 10-6 for more about help information.

DDS and Display File Considerations

Several restrictions exist for the DDS used in defining a menu:

- The menu format and the associated help record formats for the menu in the display file must have a display size of 24 rows by 80 columns, DSPSIZ(*DS3) in DDS.
- The menu record format name must be the same as the display file name.
- Help format names must follow the convention #Hxxyy, described below. The help record format is limited to 150 items.
- A separate indicator area (INDARA keyword) must be declared in the DDS for the display.
- Subfiles cannot be used.
- Only the first 21 lines of the display should be described. Only the first 21 lines of the help display should be described. Lines 22 and 23 are used with long command lines; system messages are shown on line 24. (Line 22 is used with short command lines and lines 23 and 24 are used for messages).

  The following lines can be used when a function key legend is requested:

  - for a long command line with no function key legend, lines 1 through 21 can be used
  - for a long command line with a function key legend, lines 1 through 19 can be used
  - for a short command line with no function key legend, lines 1 through 21 can be used
  - for a short command line with a function key legend, lines 1 through 20 can be used
- The LOCK record level keyword should be used to prevent the keyboard from unlocking before the display is shown.
- The Allow Roll (ALWROL) record level keyword cannot be specified.
- DFRWRT(*NO) is *required* on the CRTDSPF command when creating a display file menu. DFRWRT(*NO) ensures the menu format is displayed when the menu is run.
- The RSTDSP(*NO) is optional on the CRTDSPF command, but may significantly improve the menu's performance if specified.
- An alias can be used for the name of the menu and for the name of the work station (for S/36EE). When issuing the CRTMNU command, you can specify a different name than the name used in the DDS. This way, the name of the

work station identifier will be displayed with the menu. By giving the menu a different name, you can use the GO command to find that menu.

See the *DDS Reference* for information about DDS coding and keywords.

Message Files

Message files are used to describe what action is taken when a menu option is selected. Commands controlling the action to be taken are usually placed in the message.

In some cases, the command string may be too long to fit in the message. In these cases, the message text contains the message identifier for the message; the command should be placed in the message text.

Messages used in display file menus must be named using the message prefix USR. The remaining four digits of the message identifier correspond to the menu option number.

For example, if option 3 on a given display file menu is:

```
3. Personnel Menu
```

then the message describing the action taken when 3 is entered on the command line would be identifier USR0003. The message could contain any command, such as `'GO PERSMENU RTNPNT(*NO)'`.

See Chapter 7 and Chapter 8 for more information about messages.

Help for Menus

Help formats for display file menus are named using the form:

```
#Hxxyy
```

where:

`xx` is the number of the first menu option to which the help information applies
`yy` is the number of the last menu option to which the help information applies

For example, `#H0103` could be used to name help that applies to menu options 1 through 3.

`#H0000` names the general help for the menu.

The help formats may be described in any order in the DDS. The system sorts the help formats in ascending order when the menu is run.

If two or more help format names apply to the same option only the first help format (as sorted by the system) will be shown when Help is requested for that option. For example, if the following help formats are given:

```
#H0000
#H0101
#H0205
#H0306
#H0707
```

Typing 3 on the command line (to select option 3) and pressing the Help key will show the help designated #H0205. In this example, #H0306 can be viewed using the Page Down key, but will be shown *directly* only by requesting help for option 6.

Similarly, by moving the cursor to the command line and pressing the Help key with no option typed on the command line, the user is shown general help for the menu (#H0000). The cursor movement keys can be used to look through the other help formats for the menu options.

Example of Help for a Display File Menu

The following example shows a menu with five options. The names of the help formats and the menu options to which they apply are shown following the example.

```
PERSMENU                    Personnel Menu

Select one of the following:

     1. Departments menu
     2. Education menu
     3. Benefits menu
     4. Job openings
     5. Job applicants









Selection or Command
===>
```

Help Name	**Contents**
#H0000	General help for the Personnel Menu (PERSMENU)
#H0101	Help for option 1, Departments menu
#H0202	Help for option 2, Education menu
#H0303	Help for option 3, Benefits menu
#H0405	Help for options 4 and 5, Job openings and Job applicants

The help information for the menu options can be shown by typing the option number on the command line and pressing the Help key, or by pressing Help (general help for the menu) then using the Page Down key to page through the help for the options.

Help

Help is available for the following:

- Commands. Command help is shown by typing the command on the command line, then pressing the Help key.
- Function Keys. Help for function keys is shown by moving the cursor to the function area and pressing the Help key.

- Messages. Help for messages can be shown by moving the cursor to a message and pressing the Help key.

Creating a Display File Menu

The following example shows how a display file (*DSPF) menu is created. The following steps will be shown:

1. Describing the menu and menu help information (DDS)
2. Creating the display file (CRTDSPF command)
3. Creating and adding messages to the message file (CRTMSGF and ADDMSGD commands)
4. Creating the menu (CRTMNU command)
5. Running the menu

A display file menu is designed using DDS to describe the menu's appearance and provide help for the menu.

The following DDS is for the sample display called PERSMENU. The first record format (PERSMENU), is used to describe the appearance of the menu when displayed. Five menu options are listed, starting in position 7 of lines 5 through 9.

Five more record formats are shown: #H0000, #H0101, #H0202, #H0303, and #H0405. Each of these is used to provide help information for the menu and its options. Following the system convention described above (see "Help for Menus" on page 10-6), #H0000 is used to provide general help for the menu. The other record formats provide help for the menu options.

See the *DDS Reference* for more information about DDS coding.

IBM International Business Machines Corporation **AS/400™ DATA DESCRIPTION SPECIFICATIONS**

Printed in U.S.A.

File	Keying Instruction	Graphic	Description	Page 1 of 3
Programmer / Date		Key		

A

```
     A* PERSONNEL MENU (PERSMENU) SPECIFICATION
     A*
     A                                      PRINT DSPSIZ(*DS3)
     A                                      INDARA
     A*
     A          R PERSMENU                  LOCK
     A                                  1 02'PERSMENU'
     A                                  1 28'PERSONNEL MENU'
     A                                  3 02'SELECT ONE OF THE FOLLOWING:'
     A                                  5 07'1. DEPARTMENTS MENU'
     A                                  6 07'2. EDUCATION MENU'
     A                                  7 07'3. BENEFITS MENU'
     A                                  8 07'4. JOB OPENINGS'
     A                                  9 07'5. JOB APPLICANTS'
     A                                 21 02'SELECTION OR COMMAND'
     A*
     A          R #H0000                    LOCK
     A                                  1 02'GENERAL'
     A                                  1 26'HELP FOR THE PERSONNEL MENU
     A                                  3 02'THIS IS THE GENERAL HELP'
     A
     A
     A
```

*Number of sheets per pad may vary slightly.

```
     A                                  3 27'FOR THE PERSONNEL MENU.'
     A*
     A          R #H0101                    LOCK
     A                                  1 02'HELP1'
     A                                  1 26'HELP FOR OPTION 1'
     A                                  3 02'THIS IS THE HELP'
     A                                  3 19'FOR OPTION 1.'
     A*
     A          R #H0202                    LOCK
     A                                  1 02'HELP2'
     A                                  1 26'HELP FOR OPTION 2'
     A                                  3 02'THIS IS THE HELP'
     A                                  3 19'FOR OPTION 2.'
     A*
     A          R #H0303                    LOCK
     A                                  1 02'HELP3'
     A                                  1 26'HELP FOR OPTION 3'
     A                                  3 02'THIS IS THE HELP'
     A                                  3 19'FOR OPTION 3.'
     A*
```

```
     A          R #H0405                    LOCK
     A                                  1 02'HELP45'
     A                                  1 26'HELP FOR OPTIONS 4 AND 5'
     A                                  3 02'THIS IS THE HELP'
     A                                  3 19'FOR OPTIONS 4 AND 5.'
     A*
```

RSLF170-1

Building the Display File Menu

The following are the steps in building a display file menu:

1. The DDS source member, PERSMENU, in the source file SFPERS in library PERLIB, is used to create the display file for the PERSMENU menu with the Create Display File (CRTDSPF) command:

   ```
   CRTDSPF FILE(PERLIB/PERSMENU) +
           SRCFILE(USERLIB/SFPERS) +
           DFRWRT(*NO) +
           RSTDSP(*NO)
   ```

   The following is the compiler list created by the CRTDSPF command:

```
5728SS1 R01 M00 861114             Data Description               PERLIB/PERSMENU        06/14/88 15:01:31        Page
File name . . . . . . . . . . . . . . . . . . . . . . :   PERSMENU
  Library name  . . . . . . . . . . . . . . . . . . . :   PERLIB
File attribute  . . . . . . . . . . . . . . . . . . . :   Display
Source file containing DDS  . . . . . . . . . . . . . :   SFPERS
  Library name  . . . . . . . . . . . . . . . . . . . :   USERLIB
Source member containing DDS  . . . . . . . . . . . . :   PERSMENU
Source member last changed  . . . . . . . . . . . . . :   06/14/88  15:01:14
Source listing options  . . . . . . . . . . . . . . . :   *SOURCE    *LIST
DDS generation severity level . . . . . . . . . . . . :   20
Authority . . . . . . . . . . . . . . . . . . . . . . :   *CHANGE
Text  . . . . . . . . . . . . . . . . . . . . . . . . :
Compiler  . . . . . . . . . . . . . . . . . . . . . . :   IBM AS/400 Data Description Processor
```

```
                                  Data Description Source
SEQNBR  *...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8  Date
  100        ***   START OF SPECIFICATIONS   ***********************************       06/14/88
  110         *                                                                        06/14/88
  120         *                                                                        06/14/88
  130        A                                    PRINT DSPSIZ(*DS3)                   06/14/88
  140        A                                    INDARA                               06/14/88
  150        *******************************************************************       06/14/88
  160        A          R PERSMENU                LOCK                                 06/14/88
                   CPD5235-**********
  170        A                                1  2'PERSMENU'                           06/14/88
  180        A                                1 29'Personnel Menu'                     06/14/88
  190        A                                3 02'Select one of the following:'       06/14/88
  200        A                                5 07'1. Departments menu'                06/14/88
  210        A                                6 07'2. Education menu'                  06/14/88
  220        A                                7 07'3. Benefits menu'                   06/14/88
  230        A                                8 07'4. Job openings'                    06/14/88
  240        A                                9 07'5. Job applicants'                  06/14/88
  250        A                               21 02'Selection or Command'               06/14/88
  260        *******************************************************************       06/14/88
  270        A          R #H0000                  LOCK                                 06/14/88
  280        A                                1  2'GENERAL'                            06/14/88
  290        A                                1 26'Help for the Personnel menu'        06/14/88
  300        A                                3  2'This is the general help'           06/14/88
  310        A                                3 27'for the Personnel Menu.'            06/14/88
  320        *******************************************************************       06/14/88
  330        A          R #H0101                  LOCK                                 06/14/88
  340        A                                1  2'HELP1'                              06/14/88
  350        A                                1 26'Help for option 1'                  06/14/88
  360        A                                3  2'This is the help'                   06/14/88
  370        A                                3 19'for option 1.'                      06/14/88
  380        *******************************************************************       06/14/88
  390        A          R #H0202                  LOCK                                 06/14/88
  400        A                                1  2'HELP2'                              06/14/88
  410        A                                1 26'Help for option 2'                  06/14/88
  420        A                                3  2'This is the help'                   06/14/88
  430        A                                3 19'for option 2.'                      06/14/88
  440        *******************************************************************       06/14/88
  450        A          R #H0303                  LOCK                                 06/14/88
  460        A                                1  2'HELP3'                              06/14/88
  470        A                                1 26'Help for option 3'
  480        A                                3  2'This is the help'                   06/14/88
  490        A                                3 19'for option 3.'
  500        *******************************************************************       06/14/88
  510        A          R #H0405                  LOCK                                 06/14/88
  520        A                                1  2'HELP45'                             06/14/88
5728SS1 R01 M00 861114            Data Description             PERLIB/PERSMENU          06/14/88 15:01:31          Page    2
                                  Data Description Source
SEQNBR  *...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8  Date
  530        A                                1 26'Help for options 4 and 5'           06/14/88
  540        A                                3  2'This is the help'                   06/14/88
  550        A                                3 19'for options 4 and 5.'               06/14/88
                    * * * * *   E N D   O F   S O U R C E   * * * * *
```

```
5728SS1 R01 M00 861114              Data Description              PERLIB/PERSMENU        06/14/88 15:01:31
                                     Expanded Source
                                                                                    Field       Buffer position
SEQNBR  *...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8 length        Out     In
   130                                          PRINT DSPSIZ(*DS3) INDARA
   160               R PERSMENU                 LOCK
   170                                      1  2'PERSMENU'                              8
   180                                      1 29'Personnel Menu'                       14
   190                                      3  2'Select one of the following:'         28
   200                                      5  7'1. Departments menu'                  19
   210                                      6  7'2. Education menu'                    17
   220                                      7  7'3. Benefits menu'                     16
   230                                      8  7'4. Job openings'                      15
   240                                      9  7'5. Job applicants'                    17
   250                                     21  2'Selection or Command'                 20
   270               R #H0000                   LOCK
   280                                      1  2'GENERAL'                               7
   290                                      1 26'Help for the Personnel menu'          27
   300                                      3  2'This is the general help'             24
   310                                      3 27'for the Personnel Menu.'              23
   330               R #H0101                   LOCK
   340                                      1  2'HELP1'                                 5
   350                                      1 28'Help for option 1'                    17
   360                                      3  2'This is the help'                     16
   370                                      3 19'for option 1.'                        13
   390               R #H0202                   LOCK
   400                                      1  2'HELP2'                                 5
   410                                      1 26'Help for option 2'                    17
   420                                      3  2'This is the help'                     16
   430                                      3 19'for option 2.'                        13
   450               R #H0303                   LOCK
   460                                      1  2'HELP3'                                 5
   470                                      1 26'Help for option 3'                    17
   480                                      3  2'This is the help'                     16
   490                                      3 19'for option 3.'                        13
   510               R #H0405                   LOCK
   520                                      1  2'HELP45'                                6
   530                                      1 28'Help for options 4 and 5'             24
   540                                      3  2'This is the help'                     16
   550                                      3 19'for options 4 and 5.'                 20
                * * * * *   E N D   O F   E X P A N D E D   S O U R C E   * * * * *
5728SS1 R01 M00 861114              Data Description              PERLIB/PERSMENU        06/14/88 15:01:31
                                         Messages
   ID       Severity  Number  Text
 * CPD5235     10        1    Record name is the same as name of file being created
                                      Message Summary
   Total          Informational        Warning       Error       Severe
                     (0-9)             (10-19)      (20-29)     (30-99)
      1                0                  1            0           0
 * CPC7301  00   File PERSMENU created in PERLIB.
                        * * * * *   E N D   O F   C O M P I L A T I O N   * * * * *
```

Note: The message CPD5235 is an information message only. The record format name *must* be the same as the display file name.

2. The following Create Message File (CRTMSGF) and Add Message Description (ADDMSGD) commands are entered to build the message file.

   The CRTMSGF command creates a message file (also called PERSMENU) which will be used to contain messages describing the actions taken when the various menu options are selected.

   ```
   CRTMSGF MSGF(PERLIB/PERSMENU) +
           TEXT('Message file of commands for menu PERSMENU.')
   ```

   The MSGID parameter of each ADDMSGD command is in the form USRxxxx, where xxxx is the menu option number. The MSG parameter contains the command to run the menu option.

   The MSG parameters of the first three messages shown below contain commands to run menus (using the GO command); the last two messages contain commands which will call programs (using the CALL) command when either of those menu options are selected from the menu.

```
ADDMSGD MSGID(USR0001) MSGF(PERLIB/PERSMENU) +
        MSG('GO DEPTMENU RTNPNT(*NO)')
ADDMSGD MSGID(USR0002) MSGF(PERLIB/PERSMENU) +
        MSG('GO EDUCMENU RTNPNT(*NO)')
ADDMSGD MSGID(USR0003) MSGF(PERLIB/PERSMENU) +
        MSG('GO BENEMENU RTNPNT(*NO)')
ADDMSGD MSGID(USR0004) MSGF(PERLIB/PERSMENU) +
        MSG('CALL JOBOPEN')
ADDMSGD MSGID(USR0005) MSGF(PERLIB/PERSMENU) +
        MSG('CALL JOBAPPS')
```

3. The menu object is created using the Create Menu (CRTMNU) command:

```
CRTMNU  MENU(PERLIB/PERSMENU) TYPE(*DSPF) DSPF(*MENU) +
        MSGF(*MENU) DSPKEY(*NO) CMDLIN(*LONG) +
        TEXT('Personnel menu')
```

The DSPKEY parameter specifies whether the function key legend is shown at the bottom of the menu when the menu is displayed. *NO specifies that the function key legend is not shown at the bottom of the menu. *YES specifies that the function key is shown at the bottom of the menu.

The CMDLIN parameter specifies the length of the command line to be displayed. *LONG specifies a 153-byte long command line. *SHORT specifies a 73-byte long command line. *NONE specifies a 4-byte option line in place of a command line.

Note: Note that for both the DSPF and MSGF parameters, the value *MENU was specified. *MENU specifies that the name of the display or message file is the same as the name of the menu. Display and message file names *do not* have to have the same name as the menu.

4. The new display file menu can now be run using the GO command:

```
GO PERSMENU
```

```
PERSMENU                    Personnel Menu

Select one of the following:

     1. Departments menu
     2. Education menu
     3. Benefits menu
     4. Job openings
     5. Job applicants










Selection or Command
===>____________________________________________________________
________________________________________________________________
```

See the *CL Reference* for detailed descriptions of the commands used in this example.

Program Menus

Program menus provide an alternative to display file and system menus by using the GO command to call a program. The user has complete control of the display and function of the program menu through the program.

Parameters Passed for Program Menus

Three parameters are passed to the program when the menu is run:

- The 10-character name of the menu object
- The 10-character name of the library containing the menu object
- A 2-character binary return code

All three parameters must be declared as variables in the program. The return code determines how the program menu is exited, simulating the function keys found on IBM-supplied menus. The four possible values are:

Binary Return Code	Hex Equivalent	Description
0	0000	Call the program (display the menu) again
-1	FFFF	Exit function requested
-2	FFFE	Previous function requested
-4	FFFC	Home function requested (display the initial menu

Note: The hexadecimal values shown above are used for those high-level languages (such as CL) that do not support binary numbers. CL programs should use a 2-byte character variable. See the example shown under "Converting from and to Binary Data" on page 6-36.

Creating a Program Menu

This example shows how a program (*PGM) menu is created. The following steps will be shown:

1. Describing the menu (DDS)
2. Creating the display file (CRTDSPF command)
3. Entering source and creating a CL program (CRTCLPGM command) to control the menu function
4. Creating the menu (CRTMNU command)
5. Running the menu

The program menu is designed using data description specifications (DDS) to describe the menu's appearance. The following DDS is for the sample display called PGMMENU. See the *DDS Reference* for more information about DDS coding.

```
      * DDS SPECIFICATIONS FOR PROGRAM MENU EXAMPLE
      *
                                            DSPSIZ(*DS3)
                                            PRINT
                                            CA03(01)
                                            CA12(02)
                                            HOME(03)
                                            INDARA
                R MENUFMT                   BLINK OVERLAY
                  DSPMNUN       10A  O  1  2
                                        1 72TIME
                                        1 31'EXAMPLE PROGRAM MENU'
                                            DSPATR(HI)
                                        2  2'SELECT ONE OF THE FOLLOWING:'
                                        3  5'1.' DSPATR(HI)
                                        3 10'DISPLAY LIBRARY LIST  (DSPLIBL)'
                                        4  5'2.' DSPATR(HI)
                                        4 10'WORK WITH ACTIVE JOBS (WRKACTJOB)'
                                        5  5'3.' DSPATR(HI)
                                        5 10'WORK WITH YOUR JOB    (WRKJOB)'
                                        6  5'90.' DSPATR(HI)
                                        6 10'RETURN'
                                       23  2'OPTION:'
                  OPTION         3   I 23 12DSPATR(PC)
```

RSLF171-3

Building the Program Menu

The following are the steps in building a program menu:

1. The DDS source file, called SOURCE1, is used to create the display file for the PGMMENU menu with the Create Display File (CRTDSPF) command:

   ```
   CRTDSPF FILE(QGPL/PGMDDS) SRCFILE(USERLIB/SOURCE1)
   ```

2. The following CL program is used to control the menu function. (See Chapter 6 for more information about entering CL source.) Note that any high-level language supported by the system could be used to control a menu.

```
PGM        PARM(&MENUNAME &MENULIB &ACTION) /* Begin the program and +
             indicate that 3 parameters will be passed to it when +
             called.  The parameters include 1) The menu name, 2) +
             The menu library name 3) The action desired by this +
             program on return. */
DCL        VAR(&MENUNAME) TYPE(*CHAR) LEN(10) /* Menu name */
DCL        VAR(&MENULIB) TYPE(*CHAR) LEN(10) /* Menu library name */
DCL        VAR(&ACTION) TYPE(*CHAR) LEN(2) /* Action variable */
DCLF       FILE(QGPL/PGMDDS) RCDFMT(MENUFMT) /* Display file */
CHGVAR     VAR(&DSPMNUN) VALUE(&MENUNAME) /* Set the menu name on the +
             menu                   */
SNDRCVF    DEV(*FILE) RCDFMT(MENUFMT) /* Display the menu at the +
             display station */
CHGVAR     VAR(&ACTION)  VALUE(X'0000') /* Indicate the menu should +
             be displayed again */
/*****************************************************************/
/* Handle function keys                                          */
/*****************************************************************/
IF         COND(&IN01 *EQ '1') THEN(CHGVAR VAR(&ACTION) +
            VALUE(X'FFFF')) /* If F3 was pressed, set action +
            to EXIT */
IF         COND(&IN02 *EQ '1') THEN(CHGVAR VAR(&ACTION) +
            VALUE(X'FFFE')) /* If F12 was pressed, set action +
            to PREVIOUS */
IF         COND(&IN03 *EQ '1') THEN(CHGVAR VAR(&ACTION) +
            VALUE(X'FFFC')) /* If HOME was pressed, set action +
            to HOME */
/*****************************************************************/
/* Handle menu options                                           */
/*****************************************************************/
IF         COND(&OPTION *EQ '1') THEN(DSPLIBL) /* If the menu user +
             has selected option 1, then display the library list */
IF         COND(&OPTION *EQ '2') THEN(WRKACTJOB) /* If the menu user +
             has selected option 2, then display the active jobs */
IF         COND(&OPTION *EQ '3') THEN(WRKJOB) /* If the menu user +
             has selected option 3, then display the current job. */
IF         COND(&OPTION *EQ '90') THEN(CHGVAR VAR(&ACTION) +
             VALUE(X'FFFE')) /* If the menu user has selected option +
             90, then set the action to previous. */
ENDPGM     /* End the program */
```

3. The CL program is created using the Create CL Program (CRTCLPGM) command:

```
CRTCLPGM PGM(QGPL/PGMCL) SRCFILE(USERLIB/SOURCE1)
```

4. The program menu is created using the Create Menu (CRTMNU) command:

```
CRTMNU MENU(QGPL/PGMMENU) TYPE(*PGM) PGM(QGPL/PGMCL)
```

5. The new display file menu can now be run using the GO command:

```
GO PGMMENU
```

```
                              EXAMPLE PROGRAM MENU (PGMMENU)             09:02:51
SELECT ONE OF THE FOLLOWING:
   1.   DISPLAY LIBRARY LIST  (DSPLIBL)
   2.   WORK WITH ACTIVE JOBS (WRKACTJOB)
   3.   WORK WITH YOUR JOB    (WRKJOB)
   90.  RETURN









OPTION:  __
```

For detailed descriptions of the CL commands used in this example, see the *CL Reference*.

Exiting from a Program Menu without Returning to the Previous Menu

When you code a TYPE(*PGM) menu to go from one program menu to another and use the RTNPNT(*NO) parameter, you need to tell the menu driver when to exit the original menu.

The program menu needs to communicate with the menu driver that F3 was pressed. If the program menu is coded to go to program MENU2, with a RTNPNT(*NO), it has to be able to convey information between programs and to convey information to the menu driver to tell it what should be done. You can use data queues to communicate between the programs. For more information about data queues, see “Using Data Queues to Communicate between Programs” on page 3-13. The following example shows you how to communicate between program menus if you want to use the RTNPNT(*NO) parameter to go from one menu to another:

Program 1

```
         PGM
         DCL  VAR(&FLDLEN) TYPE(*DEC) LEN(5 0) VALUE (10)
         DCL  VAR(&FIELD) TYPE(*CHAR) LEN(10)
         DCL  VAR(&WAIT) TYPE(*DEC) LEN(5 0) VALUE(0)  /* Don't wait */
         .
         .
         /*    Handle the function keys on the menu                  */
         IF    COND(&IN03 *EQ '1')     +    /* F3=Exit pressed       */
                 THEN(DO)
                      /*   Delete the Data Queue                     */
                      DLTDTAQ  DTAQ(QGPL/DTAQ1)
                      /*   Create the Data Queue                     */
                      CRTDTAQ  DTAQ(QGPL/DTAQ1) MAXLEN(10)          +
                               TEXT('Test Data Queue')
                      GOTO SNDDQ           /* Send info to data queue*/
                      ENDDO
         .
         /*    Handle the options on the menu                        */
         IF    COND(&OPTION *EQ 1)                                    +
                THEN(GO MENU(*LIBL/MENU2) RTNPNT(*NO)
         .
         .
         /*    Returned from menu 2 after F3 or F12 pressed          */
         CALL  PGM(QRCVDTAQ)  PARM(QGPL DTAQ1 &FLDLEN &FIELD &WAIT)
         IF    COND((&FLDLEN *NE 0) *AND (&FIELD *EQ 'EXIT      '))
                 THEN(GOTO SNDDQ)
         ELSE  GOTO END
         .
         .
   SNDDQ:DO
           /*  Change the variable so the menu driver will know to   */
           /*  exit and send information to the data queue to        */
           /*  communicate to next program what happened             */
           CHGVAR VAR(&ACTION) VALUE(X'FFFF')
           CHGVAR VAR(&FLDLEN) VALUE(4)
           CHGVAR VAR(&FIELD) VALUE('EXIT      ')
           CALL PGM(QSNDDTAQ) PARM(QGPL DTAQ1 *FLDLEN &FIELD)
           GOTO END
         ENDDO
   END:  ENDPGM
```

Program 2

```
         PGM
         DCL  VAR(&FLDLEN) TYPE(*DEC) LEN(5 0) VALUE (10)
         DCL  VAR(&FIELD) TYPE(*CHAR) LEN(10)
         DCL  VAR(&WAIT) TYPE(*DEC) LEN(5 0) VALUE(0)  /* Don't wait */
         .
         .
         /*    Handle the function keys on the menu                */
         IF    COND(&IN03 *EQ '1')     +    /* F3=Exit pressed      */
                 THEN(DO)
                      /*   Delete the Data Queue                   */
                      DLTDTAQ  DTAQ(QGPL/DTAQ1)
                      /*   Create the Data Queue                   */
                      CRTDTAQ  DTAQ(QGPL/DTAQ1) MAXLEN(10)         +
                                TEXT('Test Data Queue')
                      GOTO SNDDQ           /* Send info to data queue*/
                      ENDDO
         .
         .
         /*    Handle the options on the menu                      */
         IF    COND(&OPTION *EQ 1)                                  +
                THEN(GO MENU(*LIBL/MENU2) RTNPNT(*NO)
         .
         .
         /*    Returned from menu 2 after F3 or F12 pressed        */
         CALL  PGM(QRCVDTAQ)  PARM(QGPL DTAQ1 &FLDLEN &FIELD &WAIT)
         IF    COND((&FLDLEN *NE 0) *AND (&FIELD *EQ 'EXIT      '))
                 THEN(GOTO SNDDQ)
         ELSE  GOTO END
         .
         .
SNDDQ:DO
           /*  Change the variable so the menu driver will know to   */
           /*  exit and send information to the data queue to        */
           /*  communicate to next program what happened             */
           CHGVAR VAR(&ACTION) VALUE(X'FFFF')
           CHGVAR VAR(&FLDLEN) VALUE(4)
           CHGVAR VAR(&FIELD) VALUE('EXIT      ')
           CALL PGM(QSNDDTAQ) PARM(QGPL DTAQ1 *FLDLEN &FIELD)
           GOTO END
         ENDDO
END:     ENDPGM
```

Naming Considerations for User-Defined Menus

Because high-level system menus are named by full words, there is some potential for conflict with user-defined menu names. For example, if you use the CRTMNU command to create a menu called MAIN, there may be a conflict with the system-supplied menu, MAIN. If you use the GO command to call that menu (GO MAIN), the system menu MAIN will probably be shown, because it resides in the QSYS library. A conflict can also occur if a future release contains a new system menu that has the same name as a menu you created during the previous release.

To avoid naming conflicts with system command menus, you should avoid menu names that start with CMD. The system command menu names use the format CMDxxxxxxx, where xxxxxxx is any subject or verb used in CL command names.

Reducing Menu Name Conflicts

All branches to system menus from other system menus use the library list search order. Use of the library list search order allows multilingual support for the system menus. It also allows you to override a system menu by placing your own version of it in a library higher on the search order than library QSYS.

There are several other techniques you can use, both in programming and in interactive work, to avoid menu name conflicts.

- You can specify the library containing the menu you want to run. For example, you can use `GO MENU(*USRLIBL/menu-name)` to call user-defined menus, and use `GO MENU(*LIBL/menu-name)` to call system menus.
- When working interactively, you can use a generic menu specification on the GO command. In an example given earlier (see "Naming Considerations for User-Defined Menus" on page 10-19), a potential conflict existed between system and user-defined menus called MAIN. By specifying GO MAIN*, the system will display a list of all menu names that start with `MAIN` in your library list and allow you to select which menu you want.
- You can use the Create Duplicate Object (CRTDUPOBJ) command to create a duplicate of the GO command for calling user-defined menus. The following example creates a duplicate of the GO command called GOUSR:

  ```
  CRTDUPOBJ OBJ(GO) FROMLIB(QSYS) OBJTYPE(*CMD) NEWOBJ(GOUSR) TOLIB(QGPL)
  ```

  Next, use the Change Command Default (CHGCMDDFT) command to change the command default for the MENU parameter to use library *USRLIBL rather than *LIBL:

  ```
  CHGCMDDFT CMD(QGPL/GOUSR) NEWDFT('MENU(*USRLIBL/*N)')
  ```

  With the MENU parameter default changed to *USRLIBL, the GOUSR command will bypass system menus and find only those menus in the user portion of your library list.

Other Commands Used with Menus

In addition to the GO and CRTMNU commands, several other commands are available for working with menu objects:

- DSPMNUA

  The Display Menu Attributes (DSPMNUA) command can be used to show the attributes of a menu object. These include:

  - The menu type (IBM-supplied, display file, or program menu)
  - The display and message files used by display file (*DSPF) menus
  - The programs used by program (*PGM) menus
  - The current (CURLIB) and product (PRDLIB) libraries associated with the menu
  - The text describing the menu object

- CHGMNU

  The Change Menu (CHGMNU) command can be used to change the attributes of a menu object without having to re-create the object. The attributes which can be changed are:

- The current (CURLIB) and product (PRDLIB) libraries for all menu types (including IBM-supplied menus)
- The descriptive text for all menu types
- The display and message files used by display file (*DSPF) menus
- The programs used by program (*PGM) menus

- DLTMNU

  The Delete Menu (DLTMNU) command can be used to delete menu objects from a library. The DLTMNU command can be used to delete only the menu object. Referenced display and message files (used by display file menus) and programs (used by program menus) can also be deleted.

  DLTMNU is a *generic* command. For example, all menus in a library called OELIB could be deleted using the command, `DLTMNU OELIB/*ALL`.

  As another example, if only those menus in OELIB whose names started with ACC were to be deleted, the command, `DLTMNU OELIB/ACC*` could be used.

See the *CL Reference* for detailed descriptions of these commands.

Chapter 11. Testing Functions

Testing functions are designed to help you write and maintain your applications. It lets you run your programs in a special testing environment while closely observing and controlling the processing of these programs in the testing environment. You can interact with your programs using the testing functions described in this chapter. These functions are available through a set of commands that can be used interactively or in a batch job. The functions allow you to:

- Trace a program's processing sequence and show the statements processed and the values of program variables at each point in the sequence.
- Stop at any statement in a program (called a breakpoint) and receive control to perform a function such as displaying or changing a variable value or calling another user-defined program.

No special commands specifically for testing are contained in the program being tested. The same program being tested can be run normally without changes. All test commands are specified within the job the program is in, not as a permanent part of the program being tested. With the testing commands, you interact with the programs symbolically in the same terms as the high-level language (HLL) program was written in. You refer to variables by their names and statements by their numbers. (These are the numbers used in the program's source list.) In addition, the test functions are only applicable to the job they are set up in. The same program can be used at the same time in another job without being affected by the testing functions set up.

Debug Mode

To begin testing, your program must be put in debug mode. Debug mode is a special environment in which the testing functions can be used in addition to the normal system functions. Testing functions cannot be used outside debug mode. To start debug mode, you must use the Start Debug (STRDBG) command. In addition to placing your program in debug mode, the STRDBG command lets you specify certain testing information such as the programs that are being debugged. Your program remains in debug mode until an End Debug (ENDDBG) or Remove Program (RMVPGM) command is encountered or your current routing step ends.

The following STRDBG command places the job in debug mode and adds program CUS310 as the program to be debugged.

```
STRDBG  PGM(CUS310)
```

Adding Programs to Debug Mode

Any program can be run in debug mode, but before you can debug it, you must put it in debug mode. You can place a program in debug mode by specifying it in the PGM parameter on the STRDBG command or by adding it to the debugging session with an Add Program (ADDPGM) command. You can specify as many as 10 programs to be debugged simultaneously in a job. You must have *CHANGE authority to add a program to debug mode.

If you specified 10 programs for debug mode (using either the STRDBG or ADDPGM command or both commands) and you want to add more programs to the debug job, you must remove some of the previously specified programs. Use the Remove Program (RMVPGM) command. When debug mode ends, all programs are automatically removed from debug mode.

When you start debug mode, you can specify that a program be a default program. By specifying a default program, you can use any debug command that has the PGM parameter without having to specify a program name each time a command is used. This is helpful if you are only debugging one program. For example, in the Add Breakpoint (ADDBKP) command, you would not specify a program name on the PGM parameter because the default program is assumed to be the program the breakpoint is being added to. The default program name must be specified in the list of programs to be debugged (PGM parameter). If more than one program is listed to be debugged, you can specify the default program on the DFTPGM parameter. If you do not, the first program in the list on the PGM parameter on the STRDBG command is assumed to be the default program.

The default program can be changed any time during testing by using either the Change Debug (CHGDBG) or the ADDPGM command.

Note: If a program that is in debug mode is deleted, re-created, or saved with storage freed, references made to that program (except a RMVPGM command) may result in a function check. You must either remove the program using a RMVPGM command or end debug mode using an ENDDBG command. If you want to change the program and then debug it, you must remove it from debug mode and after it is re-created, add it to debug mode (ADDPGM command).

Preventing Updates to Database Files in Production Libraries

You can use files in production libraries while you are in debug mode. To prevent database files in production libraries from being unintentionally changed, you can specify UPDPROD(*NO) or default to *NO on the STRDBG command. Then, only files in test libraries can be opened for updating or adding new records. If you want to open database files in production libraries for updating or adding new records or if you want to delete members from production physical files, you can specify UPDPROD(*YES).

You can use this function with the library list. In the library list for your debug job, you can place a test library before a production library. You should have copies of the production files that might be updated by the program being debugged in the test library. Then, when the program runs, it uses the files in the test library. Therefore, production files cannot be unintentionally updated.

The Program Stack

You can use the Display Debug (DSPDBG) command to display the program stack, which indicates:

- Which programs are currently being debugged
- The instruction number of the calling instruction or the instruction number of each breakpoint at which program processing is stopped

- The program recursion level
- The names of the programs that are in debug mode but have not been called

A call of a program is the allocation of *automatic* storage for the program and the transfer of machine processing to the program. A series of calls is placed in a program stack; when a program finishes processing or transfers control, it is removed from the program stack. See Chapter 3 for more information about the program stack.

A program may be called a number of times while the first call is still in the program stack. This is called nesting. 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 program stack. A procedure's recursion level is referred to explicitly as the procedure recursion level.)
3. All CL commands and displays make use of only the program qualified name recursion level.

Program Activations

An activation of a program is the allocation of static storage for the program. An activation is always ended when one of the following happens:

- The current routing step ends.
- The request that activated the program is canceled.
- The Reclaim Resources (RCLRSC) command is run such that the last (or only) call of a program is ended.

In addition, an activation can be destroyed by actions taken during a program call. These actions are dependent on the language (HLL or CL) in which the program is written.

When a program is deactivated, static storage is returned to the system. The language (HLL or CL) in which the program is written determines when the program is normally deactivated. A CL program is always deactivated when the program ends.

An RPG/400 program is deactivated when the last record indicator (LR) is set on before the program ends. If there is a return operation and LR is off, the program is not deactivated.

Handling Unmonitored Messages

Normally, if a program receives an unmonitored escape message, the system sends the function check message (CPF9999) to the program's program message queue and the program stops processing. However, HLL program compilers may insert monitors for the function check message or for messages that may occur in the program. (An inquiry message will be sent to the program messages display.) This allows you to end the program the way you want. In an interactive debug job, when a function check occurs, the system provides default handling and gives you control instead of stopping the program. The system displays the following on the unmonitored message display:

- The message
- The MI instruction number and HLL statement identifier, if available, to which the message was sent
- The name and recursion level of the program to which the message was sent

The following is an example of an unmonitored message breakpoint display:

```
                    Display Unmonitored Message Breakpoint

 Statement/Instruction . . . . . . . :   440 /0077
 Program . . . . . . . . . . . . . . :   TETEST
 Recursion level . . . . . . . . . . :   1

Errors occurred on command.







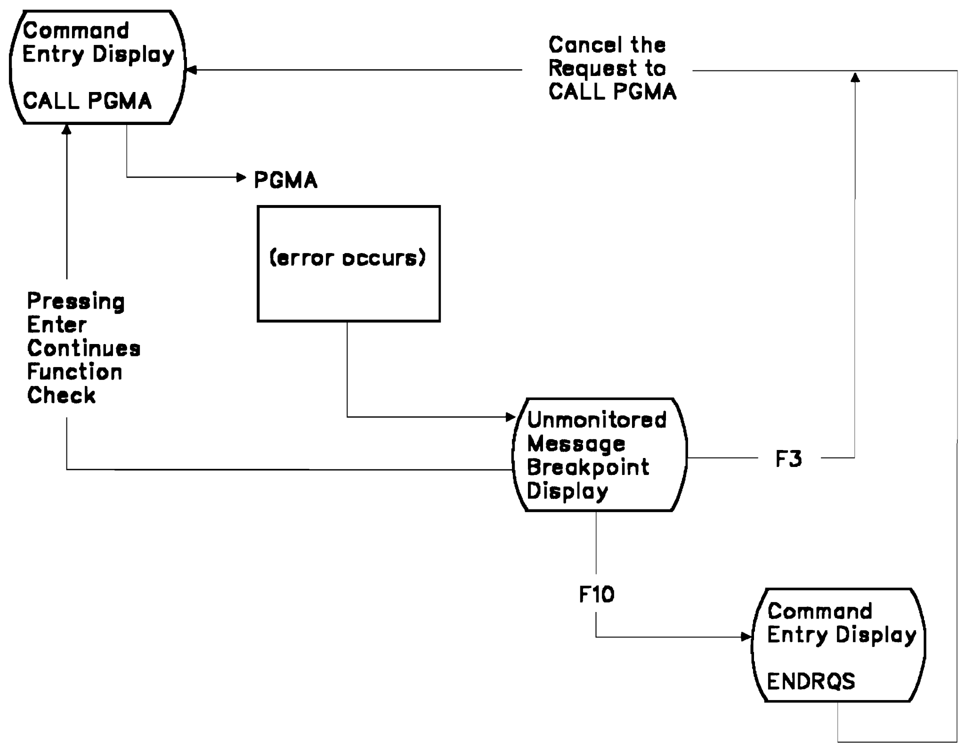


Press Enter to continue.

F3=Exit program   F10=Command Entry
```

You can try to isolate the source of the error by using the testing functions. However, the original request in error is still stopped at the point where the error occurred. To remove the request in error from the program stack, you must use the End Request (ENDRQS) command or press F3 when the unmonitored message breakpoint display is shown. You can let the usual function check processing continue by pressing the Enter key when the unmonitored message breakpoint display is shown. If you press F10 to call the command entry display, you must press F3 to return to the unmonitored message breakpoint display.

The following shows how a ENDRQS command works:



RSLF169-0

Program calls are destroyed when a ENDRQS command is entered. (In the previous diagram, the program call of PGMA is destroyed.)

Breakpoints

A breakpoint is a place in a program at which the system stops program processing and gives control to you at a display station (interactive mode) or to a program specified on the BKPPGM parameter in the Add Breakpoint (ADDBKP) command (batch mode).

Adding Breakpoints to Programs

Use the ADDBKP command to add breakpoints to the program you want debugged. You can specify up to 10 statement identifiers on the one ADDBKP command. The program variables specified on an ADDBKP command apply only to the breakpoints specified on that command. Up to 10 variables can be specified in one ADDBKP command.

You can also specify the name of the program to which the breakpoint is to be added. If you do not specify the name of the program that you want the breakpoint added to, the breakpoint is added to the default program specified on the STRDBG, CHGDBG, or ADDPGM command.

For more information about breakpoint commands, see the *CL Reference*.

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')
```

The source for CUS310 looks like this:

```
5728PW1 R01M00  880101                 SEU SOURCE LISTING

SOURCE FILE . . . . . . .  QGPL/QCLSRC
MEMBER  . . . . . . . . .  CUS310

SEQNBR*...+... 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8 ...+... 9 ...+... 0
  100 PGM   PARM(&NBRITEMS &ITEMPRC &PARBAL &PTOTBAL)
  200   DCL VAR(&PARBAL)   TYPE(*DEC) LEN(15 5) /* INPUT AREA INV BALANCE */
  300   DCL VAR(&PTOTBAL)  TYPE(*DEC) LEN(15 5) /* INPUT TOTAL INV BALANCE*/
  400   DCL VAR(&NBRITEMS) TYPE(*DEC) LEN(15 5) /* NUMBER OF ITEMS        */
  500   DCL VAR(&ITEMPRC)  TYPE(*DEC) LEN(15 5) /* PRICE OF THE ITEM      */
  600   DCL VAR(&ARBAL)    TYPE(*DEC) LEN(5 2)  /* AREA INVENTORY BALANCE */
  700   DCL VAR(&TOTBAL)   TYPE(*DEC) LEN(5 2)  /* TOTAL INVENTORY BALANCE*/
  800   DCL VAR(&TOTITEM)  TYPE(*DEC) LEN(5 2)  /* TOTAL PRICE OF ITEMS   */
  900   CHGVAR     VAR(&ARBAL) VALUE(&PARBAL)
 1000   CHGVAR     VAR(&TOTBAL) VALUE(&PTOTBAL)
 1100   IF COND(&NBRITEMS *EQ 0) THEN(DO)
 1200      SNDPGMMSG  MSG('The number of items is zero.  This item +
 1300                    should be ordered.') TOMSGQ(INVLIB/INVQUEUE)
 1400      GOTO       CMDLBL(EXIT)
 1500   ENDDO
 1600   CHGVAR     VAR(&TOTITEM) VALUE(&NBRITEMS * &ITEMPRC)
 1700   IF COND(&NBRITEMS *GT 50) THEN(DO)
 1800      SNDPGMMSG  MSG('Too much inventory for this item.') +
 1900                 TOMSGQ(INVLIB/INVQUEUE)
 2000   ENDDO
 2100   CHGVAR     VAR(&ARBAL)   VALUE(&ARBAL + &TOTITEM)
 2200   IF COND(&ARBAL *GT 1000) THEN(DO)
 2300      SNDPGMMSG  MSG('The area has too much money in +
 2400                    inventory.') TOMSGQ(INVLIB/INVQUEUE)
 2500   ENDDO
 2600   CHGVAR     VAR(&TOTBAL)  VALUE(&TOTBAL + &TOTITEM)
 2700   EXIT:      ENDPGM
```

The following is displayed as a result of reaching the first breakpoint:

```
                              Display Breakpoint

Statement/Instruction . . . . . . . . . :   900 /0009
Program . . . . . . . . . . . . . . . . :   CUS310
Recursion level . . . . . . . . . . . . :   1











Press Enter to continue.

F3=Exit program   F10=Command entry
```

The following is displayed as a result of reaching the second breakpoint:

```
                              Display Breakpoint

Statement/Instruction . . . . . . . . . :   2200 /0022
Program . . . . . . . . . . . . . . . . :   CUS310
Recursion level . . . . . . . . . . . . :   1
Start position  . . . . . . . . . . . . :   1
Format  . . . . . . . . . . . . . . . . :   *CHAR
Length  . . . . . . . . . . . . . . . . :   *DCL

Variable  . . . . . . . . . . . . . . . :   &ARBAL
  Type  . . . . . . . . . . . . . . . . :     PACKED
  Length  . . . . . . . . . . . . . . . :     5 2
 '610.00'





Press Enter to continue.

F3=Exit program   F10=Command entry
```

The variable &ARBAL is shown. (Note that the value of &ARBAL will vary depending on the parameter values passed to the program.) You can press F10 to display the command entry display so that you could change the value of the variable &ARBAL to alter your program's processing. You use the Change Program Variable (CHGPGMVAR) command to change the value of a variable.

Conditional Breakpoints

You may add a conditional breakpoint to a program that is being debugged. Use the Add Breakpoint (ADDBKP) command to specify the statement and condition. If the condition is met, the system stops the program processing at the specified statement.

You may specify a skip value on the ADDBKP command. A **skip value** is a number that indicates how many times a statement should be processed before the system stops the program. For example, to stop a program at statement 1200 after the statement has been processed 100 times, enter the following command:

```
ADDBKP STMT(1200) SKIP(100)
```

If you specify multiple statements when the SKIP parameter is specified, each statement has a separate count. The following command causes your program to stop on statement 150 or 200, but only after the statement has processed 400 times:

```
ADDBKP STMT(150 200) SKIP(400)
```

If statement 150 has processed 400 times but statement 200 has processed only 300 times, then the program does not stop on statement 200.

If a statement has not processed as many times as was specified on the SKIP parameter, the Display Breakpoint (DSPBKP) command can be used to show how many times the statement was processed. To reset the SKIP count for a statement to zero, enter the breakpoint again for that statement.

You can specify a more general breakpoint condition on the ADDBKP command. This expression uses a program variable, an operator, and another variable or constant as the operands. For example, to stop a program at statement 1500 when variable &X is greater than 1000, enter the following command:

```
ADDBKP STMT(1500) PGMVAR('&X') BKPCOND(*PGMVAR1 *GT 1000)
```

The BKPCOND parameter requires three values:

- In the example, the first value specifies the first variable specified on the PGMVAR parameter. (To specify the third variable, you would use *PGMVAR3.)
- The second value must be an operator. For a list of all valid operators, see the *CL Reference*.
- The third value may be a constant or another variable. A constant may be a number, character string, or bit string, and must be the same type as the program variable specified in the first value.

The SKIP and BKPCOND parameters can be used together to specify a complex breakpoint condition. For example, to stop a program on statement 1000 after the statement has been processed 50 times **and** only when the character string &STR is `TRUE`, enter the following command:

```
ADDBKP STMT(1000) PGMVAR('&STR') SKIP(50)
     BKPCOND(*PGMVAR1 *EQ 'TRUE ')
```

Removing Breakpoints from Programs

To remove breakpoints from a program, use the Remove Breakpoint (RMVBKP) command. To remove a breakpoint you must specify the statement number of the statement for which the breakpoint has been defined.

Traces

A trace is the process of recording the sequence in which the statements in a program are processed. A trace differs from a breakpoint in that you are not given control during the trace. The system records the traced statements that were processed. However, the trace information is not automatically displayed when the program completes processing. You must request the display of trace information using the Display Trace Data (DSPTRCDTA) command. The display shows the sequence in which the statements were processed and, if requested, the values of the variables specified on the Add Trace (ADDTRC) command.

Adding Traces to Programs

Adding a trace consists of specifying what statements are to be traced and, if you want, the names of program variables. Before a traced statement processes, the value of the variable is recorded. Also, you can specify that the values of the variables are to be recorded only if they have changed from the last time a traced statement was processed. These variables can be displayed in character format or hexadecimal format.

To specify which statements are to be traced, you can specify:

- The statement identifier at which the trace is to start and the statement identifier at which the trace is to stop
- That all statements in the program are to be traced
- A single statement identifier of a statement to be traced

On the STRDBG or CHGDBG command, you can specify how many statement traces can be recorded for a job and what action the system should take when the maximum is reached. When the maximum is reached, the system performs one of the following actions (depending on what you specify):

- For an interactive job, either of the following can be done:
  - Stop the trace (*STOPTRC). Control is given to you (a breakpoint occurs), and you can remove some of the trace definitions (RMVTRC command), clear the trace data (CLRTRCDTA command), or change the maximum (MAXTRC parameter on the CHGDBG command).
  - Continue the trace (*WRAP). Previously recorded trace data is overlaid with trace data recorded after this point.
- For a batch job, either of the following can be done:
  - Stop the trace (*STOPTRC). The trace definitions are removed and the program continues processing.
  - Continue the trace (*WRAP). Previously recorded trace data is overlaid with trace data recorded after this point.

You can change the maximum and the default action any time during the debug job using the Change Debug (CHGDBG) command. However, the change does not affect traces that have already been recorded.

You can only specify a total of five statement ranges for a single program at any one time, which is a total taken from all the Add Trace (ADDTRC) commands for the program. In addition, only 10 variables can be specified for each statement range.

In high-level language programs, different statements and labels may be mapped to the same internal instruction. This happens when there are several inoperable statements (such as DO, END) following one another in a program. You can use the IRP list to determine which statements or labels are mapped to the same instruction.

When you specify CL variables, you must enclose the & and the variable name in single apostrophes. For example:

```
ADDTRC PGMVAR('&IN01')
```

When you specify a statement range, the source statement number for the stop statement is ordinarily larger than the number for the start statement. Tracing, however, is performed with machine interface (MI) instructions, and some compilers (notably RPG/400) generate programs in which the order of MI instructions is not the same as the order of the source statements. Therefore, in some cases, the MI number of the stop statement may not be larger than the MI number of the start statement, and you will receive message CPF1982.

When you receive this message, you should do one of the following:

- Trace all statements in the program.
- Restrict a statement range to one specification.
- Use MI instruction numbers gotten from an intermediate representation of a program (IRP) list of the program. (See “Debugging at the Machine Interface Level” on page 11-19.)

The following Add Trace (ADDTRC) command adds a trace to the program CUS310. CUS310 is the default program, so it does not have to be specified. The value of the variable &TOTBAL is recorded only if its value changes between the times each traced statement is processed.

```
ADDTRC STMT((900 2700))  PGMVAR('&TOTBAL')  OUTVAR(*CHG)
```

The following displays result from this trace and are displayed using the Display Trace Data (DSPTRCDTA) command. Note that column headers are not supplied for all displays.

```
                              Display Trace Data

                 Statement/
Program          Instruction          Recursion level      Sequence number
CUS310           900                        1                           1

 Start position . . . . . . . . . . . . :   1
 Length . . . . . . . . . . . . . . . . :   *DCL
 Format . . . . . . . . . . . . . . . . :   *CHAR

 Variable . . . . . . . . . . . . . . . :   &TOTBAL
   Type . . . . . . . . . . . . . . . . :     PACKED
   Length . . . . . . . . . . . . . . . :     5 2
  '    .00'

                 Statement/
Program          Instruction          Recursion level      Sequence number
CUS310           1000                       1                           2
CUS310           1100                       1                           3  +

Press Enter to continue.

F3=Exit   F12=Cancel
```

```
                              Display Trace Data


 Start position . . . . . . . . . . . . :   1
 Length . . . . . . . . . . . . . . . . :   *DCL
 Format . . . . . . . . . . . . . . . . :   *CHAR

*Variable . . . . . . . . . . . . . . . :   &TOTBAL
   Type . . . . . . . . . . . . . . . . :     PACKED
   Length . . . . . . . . . . . . . . . :     5 2
  '   1.00'

                 Statement/
Program          Instruction          Recursion level      Sequence number
CUS310           1600                       1                           4
CUS310           1700                       1                           5
CUS310           2100                       1                           6
CUS310           2200                       1                           7
CUS310           2600                       1                           8  +

Press Enter to continue.

F3=Exit   F12=Cancel
```

```
                           Display Trace Data

CUS310        2700                       1                      9

 Start position . . . . . . . . . . . :  1
 Length . . . . . . . . . . . . . . . :  *DCL
 Format . . . . . . . . . . . . . . . :  *CHAR

*Variable . . . . . . . . . . . . . . :  &TOTBAL
   Type . . . . . . . . . . . . . . . :    PACKED
   Length . . . . . . . . . . . . . . :    5 2
  '    2.00'








Press Enter to continue.

F3=Exit   F12=Cancel
```

Instruction Stepping

You can step through the instructions of a program by using the STRDBG or CHGDBG commands and setting the MAXTRC parameter to 1 and the TRCFULL parameter to *STOPTRC. When you specify a trace range (ADDTRC command) and the program processes an instruction within that range, a breakpoint display with an error message appears. If you press Enter, another breakpoint display with the same error message appears for the next instruction processed in the trace range. When tracing is completed, the trace data contains a list of the instructions traced. You can display this data by entering the Display Trace Data (DSPTRCDTA) command.

Using Breakpoints within Traces

Breakpoints can be used within a trace range. At a breakpoint within a trace, you can display the trace data (DSPTRCDTA command) to determine if you need to take some action. The trace data is recorded before the breakpoint occurs. The trace information contains the value of any variables *before* the statement was processed.

Removing Trace Information from the System

On the DSPTRCDTA command, you can specify whether the trace information is removed from the system or left on the system after the information is displayed. If you leave the trace information on the system, any other traces are added to it. The information remains on the system (unless removed) until the debug job has completed processing. You can also use the Clear Trace Data (CLRTRCDTA) command to remove trace information from the system.

Removing Traces from Programs

The Remove Trace (RMVTRC) command removes all or some of the ranges specified in one or more Add Trace (ADDTRC) commands. Removing a trace range consists of specifying the statement identifiers used on the RMVTRC command, or specifying that all ranges be removed.

You can use the STMT parameter on the ADDTRC command to specify:

- All HLL statements and/or machine instructions in the specified program are not to be traced regardless of how the trace was defined by the ADDTRC command.
- The start and stop trace location of the HLL statements and/or system instructions to be removed.

The RMVPGM and ENDDBG commands also remove traces, but they also remove the program from debug mode.

Display Functions

In debug mode, you can display testing information that lets you review how you have set up your debug job. You can display what programs are in debug mode and what breakpoints and traces have been defined for those programs. In addition, you can display the status of the programs in debug mode.

You can use the following commands to display testing information:

- Display Debug (DSPDBG), which displays the current program stack and the names of the programs that are in debug mode and indicates the following:
  - Which are stopped at a breakpoint
  - Which are currently called
  - The request level of those that are called
  - Debug options selected for the debug job
- Display Breakpoint (DSPBKP), which displays the locations of breakpoints that are currently defined in a program.
- Display Trace (DSPTRC), which displays the statements or statement ranges that are currently defined in a program.

Displaying the Values of Variables

When you are at a breakpoint, you can display the values of program variables. You can have this done automatically on the breakpoint display by specifying the variable names on the ADDBKP command, or you can enter the Display Program Variable (DSPPGMVAR) command at the breakpoint by pressing F10 to show the command entry display. Only 10 variables can be specified on one DSPPGMVAR command. For character and bit variables, you can tell the system to begin displaying the value of the variable starting at a certain position and for a specified length. Variables can be displayed in either character or hexadecimal format.

Notes:

1. If you specify an array variable, you can do one of the following:
   a. Specify the subscript values of the array element you want to display. The subscript values can either be integer values or the names of numeric variables in the program.
   b. Display the entire array by not entering any subscripts.
   c. Display a single-dimension cross-section of the array by specifying values for all subscripts except one, and an asterisk for that one subscript value.

2. Variable names can be specified as simple or qualified names, but must be placed between apostrophes. A qualified name can be specified in either of two ways:

   a. Variable names alternating with the special separator words OF or IN, ordered from lowest to highest qualification level. A blank must separate the variable name and the special separator word.
   b. Variable names separated by periods, ordered from highest to lowest qualification level.

The following DSPPGMVAR command displays the variable ARBAL used in the program CUS310. CUS310 is the default program, so it does not have to be specified. The entire value is to be displayed in character format.

```
DSPPGMVAR  PGMVAR('&ARBAL')
```

The resulting display looks like this:

```
                         Display Program Variables

 Program . . . . . . . . . . . . . . . . . :   CUS310
 Recursion level . . . . . . . . . . . . . :   1
 Start position  . . . . . . . . . . . . . :   1
 Format  . . . . . . . . . . . . . . . . . :   *CHAR
 Length  . . . . . . . . . . . . . . . . . :   *DCL

 Variable  . . . . . . . . . . . . . . . . :   &ARBAL
   Type  . . . . . . . . . . . . . . . . . :   PACKED
   Length  . . . . . . . . . . . . . . . . :   5 2
  '610.00'








 Press Enter to continue.

 F3=Exit   F12=Cancel
```

Some HLLs allow variables to be based on a user-specified pointer variable (HLL pointer). If you do not specify an explicit pointer for a based variable, the pointer specified in the HLL declaration (if any) is used. You must specify an explicit basing pointer if one was not specified in the HLL declaration for the based variable. The PGMVAR parameter allows you to specify up to five explicit basing pointers when referring to a based variable. When multiple basing pointers are specified, the first basing pointer specified is used to locate the second basing pointer, the second one is then used to locate the third, and so forth. The last pointer in the list of basing pointers is used to locate the primary variable.

Changing the Values of Variables

To change the value of a program variable, use the Change Program Variable (CHGPGMVAR), the Change HLL Pointer (CHGHLLPTR), or the Change Pointer (CHGPTR) command. Changing the value of a program variable consists of specifying the variable name and a value that is compatible with the data type of the variable. For example, if the variable is character type, you must enter a character value.

When changing the value of variables, you should be aware of whether the variable is an automatic variable or a static variable. The difference between the two is in the storage for the variables. For automatic variables, the storage is associated with the call of the program. Every time a program is called, a new copy of the variable is placed in automatic storage. A change to an automatic variable remains in effect only for the program call the change was made in.

Note: In some languages, the definition of an call is made at the procedure level and not just at the program level. For these languages, storage for automatic variables is associated with the call of the procedure. Every time a procedure is called, a new copy of the variable is gotten. A change to an automatic variable remains in effect only while that procedure is called. Only the automatic variables in the most recent procedure call can be changed. The RCRLVL (recursion level) parameter on the commands applies only on a program basis and not on a procedure basis.

For static variables, the storage is associated with the activation. Only one copy of a static variable exists in storage no matter how many times a program is called. A change to a static variable remains in effect for the duration of the activation.

To determine if a program variable is a static or an automatic variable, request an intermediate representation of a program (IRP) list (*LIST and *XREF on the GENOPT parameter) when the program containing the variables is created.

When changing a variable that is an array, you must specify one element of the array. Consequently, you must specify the subscript values for the array element you want to change.

Using a Job to Debug Another Job

You may want to use a separate job to debug programs running in another job for one of the following reasons:

- Batch jobs can be debugged by an interactive job.
- An interactive job can be debugged from another interactive job. This allows one display to show debug information without interrupting the application program display.
- An interactive or batch job that is looping can be interrupted and put into debug mode.

Debugging Batch Jobs Submitted to a Job Queue

Using a separate job to debug another batch job submitted to the job queue allows you to put the batch job into debug mode and to set breakpoints and traces before the job starts to process. Use the following steps to debug batch jobs to be submitted to a job queue:

1. Hold the job queue that you want to submit the job to. To hold the queue QBATCH, enter the following:

   ```
   HLDJOBQ QBATCH
   ```

2. Submit the batch job using the Submit Job (SBMJOB) command or a program that automatically submits the job.

3. Determine the qualified job name (number/user/name) that is assigned to the job using the Work with Submitted Jobs (WRKSBMJOB) command or the Work with Job Queues (WRKJOBQ) command. The SBMJOB command also displays the name in a completion message when the command finishes processing.

   The WRKJOBQ (Work With Job Queue) command displays all the jobs waiting to start in a particular job queue. You can show the job name from this display by selecting option 5 for the job.

4. 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)
   ```

5. 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.

6. 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.

7. Use the Command Entry display to enter any debug commands, such as the Add Breakpoint (ADDBKP) or Add Trace (ADDTRC) commands.

8. Press F3 to leave the Command Entry display, and then press Enter to start the batch job.

9. 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.

10. If you wish to debug another batch job, first end debugging using the End Debug (ENDDBG) command and then end servicing the job using the End Servicing Job (ENDSRVJOB) command.

Debugging Batch Jobs Not Started from Job Queues

Some jobs started on the system are not submitted to a job queue. These jobs cannot be stopped before they start running but they can usually be debugged. To debug jobs not started from a job queue, do the following:

1. Rename the program that is called when the job starts. For example, if the job runs program CUST310, you can rename this program to CUST310DBG.

2. Create a small CL program with the same name as the original program (before the program was renamed). In the small CL program, use the Delay Job (DLYJOB) command to delay for one minute and then use the CALL command to call the renamed program.

3. Allow the batch job to start to force the CL program to be delayed for one minute.

4. Use the Work with Active Jobs (WRKACTJOB) command to find the batch job that is running. When the display appears, enter option 5 next to the job to obtain the qualified job name.

5. Enter the Start Service Job (STRSRVJOB) command as follows:

   ```
   STRSRVJOB JOB(qualified-job-name)
   ```

6. Enter STRDBG and any other debug commands, such as the Add Breakpoint (ADDBKP) or Add Trace (ADDTRC) command. Proceed with debugging as usual.

Debugging a Running Job

You can debug a job that is already running if you know what statements the job will run. For example, you may want to debug a running program if the job is looping or the job has not yet run a program that is to be debugged. The following steps allow you to debug a running job:

1. Use the Work with Active Jobs (WRKACTJOB) command to find the job that is running. When the display appears, enter option 5 next to the job to obtain the qualified job name.
2. Enter the Start Service Job (STRSRVJOB) command as follows:

   ```
   STRSRVJOB JOB(qualified-job-name)
   ```
3. Enter the Start Debug (STRDBG) command. (Entering the command does not stop the job from running.)

   Note: You can use the Display Debug (DSPDBG) command to show the program stack; however, unless the program is stopped for some reason, the stack is correct only for an instant and the program continues to run.
4. If you know a statement to be run, enter the Add Breakpoint (ADDBKP) command to stop the job at the statement.

   If you do not know what statements are being run, do the following:

   a. Enter the Add Trace (ADDTRC) command.

   b. After a short time, enter the Remove Trace (RMVTRC) command to stop tracing the program.

   c. Enter the Display Trace Data (DSPTRCDTA) command to show what statements have processed. Use the trace data to determine which data statements to process next (for example, statements inside a program loop).

   d. Enter the Add Breakpoint (ADDBKP) command to stop the job at the statement.
5. Enter the desired debug commands when the program is stopped at a breakpoint.

Debugging Another Interactive Job

You can debug a job from another display, whether the job is running or waiting at a menu or command entry display. To debug another interactive job, do the following:

1. Determine the qualified job name of the job to be debugged. To determine the name, either enter the Display Job (DSPJOB) command from the display of the job to be debugged, or use the Work with Active Jobs (WRKACTJOB) command.
2. Enter the Start Service Job (STRSRVJOB) command using the qualified job name.

3. Enter the Start Debug (STRDBG) command and any other debug commands desired. If the job is already running, you may need to enter the Display Debug (DSPDBG) command to determine what statement in the program is processing.

When the job being debugged is stopped at a breakpoint, the display station is locked.

Considerations When Debugging One Job from Another Job

Although most jobs can be debugged from another job, you must take the following into consideration:

- A job being debugged cannot be held or suspended (for example, when running another group job or a secondary job).
- When servicing another job with the Start Service Job (STRSRVJOB) command, you cannot also debug the job doing the servicing. All debug commands apply only to the job being serviced. To debug the job doing the servicing, you must either end the servicing of the other job, or have another job service and debug it.
- Debug commands operate on another job, even if that job is not stopped at a breakpoint. For example, if you are debugging a running job and you enter the Display Program Variable (DSPPGMVAR) command, the variable you specify is shown. Since the job continues to run, the value of the variable may change soon after the command is entered.
- A job being debugged must have enough priority to respond to debug commands. If you are debugging a batch job with a low priority and that job gets no processing time, then any debug command you issue waits for a response from the job. If the job does not respond, the command ends and an error message is displayed.
- You cannot service and debug a job that is debugging itself. However, you can service and debug a job that is servicing and debugging another job.

Debugging at the Machine Interface Level

To debug your programs at the machine interface (MI) level, you can specify an MI object definition vector (ODV) number for the PGMVAR parameter of a command and MI instruction numbers for the STMT parameter of a command. For a breakpoint, the system stops at the MI instruction number just as it would at an HLL statement number. You must always precede the ODV or MI instruction number with a slash (/) and enclose it in apostrophes (for example, '/1A') to signal to the system that you are debugging at the MI level.

The ODV and MI instruction numbers can be obtained from the IRP listing produced by most high-level language compilers. Use the *LIST value of the GENOPT parameter to produce the IRP listing at program creation time.

Note: When you debug at the machine interface level, only the characteristics that are defined at the machine interface level are available; the HLL characteristics that are normally passed to the test environment are not available. These HLL characteristics may include: the variable type, number of fractional digits, length, and array information. For example, a numeric variable in your HLL program may be displayed without the correct decimal alignment or possibly as a character string.

Security Considerations

To debug a program, you must have *CHANGE authority to that program. The *CHANGE authority available by adopting another user's profile is not considered when determining whether a user has authority to debug a program. This prevents users from accessing program data in debug mode by adopting another user's profile.

Additionally, when you are at a user-defined breakpoint of a program that you are debugging with adopted user authority, you have only the authority of your user profile and not the adopted profile authority. You do not have authorities adopted by prior program calls for all breakpoints whether they are added by the Add Breakpoint (ADDBKP) command or are caused by an unmonitored escape message.

Glossary

This glossary includes terms and definitions from the *ISO Vocabulary—Information Processing* and the *ISO Vocabulary—Office Machines*, developed by the International Organization for Standardization, Technical Committee 97, Subcommittee 1. Definitions of published segments of the vocabularies are identified by the symbol (I) after the definition; definitions from draft international standards, draft proposals, and working papers in development by the ISO/TC97/SC1 vocabulary subcommittee are identified by the symbol (T) after the definition, indicating final agreement has not yet been reached among participating members.

add authority. A data authority that allows the user to add entries to an object; for example, add job entries to a job queue or add records to a file. Contrast with *delete authority*. See also *read authority*.

address. The location in the storage of a computer where particular data is stored. Also, the numbers that identify such a location.

adopted authority. Authority given to the user by the program for the duration of the job that uses that program. The program must be created with owner authority.

alert. In SNA, a record sent to a focal point to identify a problem or an impending problem.

all authority. An object authority that allows the user to perform all operations on the object except those limited to the owner or controlled by authorization list management authority. The user can control the object's existence, specify the security for the object, and change the object.

American National Standard Code for Information Interchange (ASCII). The code developed by the American National Standards Institute for information exchange among data processing systems, data communications systems, and associated equipment. The ASCII character set consists of 7-bit control characters and symbolic characters, plus one parity-check bit.

argument. In REXX, a string passed between a calling routine and a called routine or a parameter provided to a program.

argument list. In REXX, a complete set of arguments, separated by commas, that is passed between a calling routine and a called routine.

array. In RPG, a series of elements with like characteristics. An array can be searched for a uniquely identified element; or elements in an array can be accessed by their position relative to other elements.

array element. In RPG, BASIC, and PL/I, one of the data items in an array.

ASCII. See *American National Standard Code for Information Interchange (ASCII)*.

ASP. See *auxiliary storage pool (ASP)*.

asynchronous. (1) Not occurring in a regular or predictable pattern. (2) Without a regular time relationship.

authorization list. A list of two or more user IDs and their authorities for system resources. The system-recognized identifier for the object type is *AUTL.

auxiliary storage. All addressable disk storage other than main storage.

auxiliary storage pool (ASP). A group of units defined from the disk units that make up auxiliary storage. ASPs provide a means of isolating certain objects on specific disk units to prevent the loss of data due to disk media failures on other disk units. See also *unit*, *system ASP*, and *user ASP*.

batch. Pertaining to a group of jobs to be run on a computer sequentially with the same program with little or no operator action. Contrast with *interactive*.

batch job. A predefined group of processing actions submitted to the system to be performed with little or no interaction between the user and the system. Contrast with *interactive job*.

binary. (1) Pertaining to a selection, choice, or condition that has two possible values. (I) (2) A numbering system with a base of two (0 and 1). (3) A data type indicating that the data is a binary number with a precision of 15 (halfword) or 31 (fullword) bits.

bit. Either of the binary digits, 0 or 1. Compare with *byte*.

break delivery. The method of delivering messages to a message queue in which the job associated with that message queue is interrupted as soon as the message arrives.

breakpoint. A place in a program (specified by a command or a condition) where the system stops the processing of that program and gives control to the display station user or to a specified program.

breakpoint program. For a batch job, a user program that can be called when a breakpoint is specified.

buffer. (1) A routine or an area of storage that corrects for the different speeds of data flow or timings of events, when transferring data from one device to another. (2) A portion of storage used to hold input or output data temporarily.

built-in function. (1) In PL/I and CL, a predefined function, such as a commonly used arithmetic function or a function necessary to high-level language compilers (for example, a function for manipulating character strings or converting data). It is automatically called by a built-in function reference. (2) In REXX, a function that is supplied by a language. These functions, defined as part of the REXX language, include character manipulation, conversion, and information functions.

byte. A group of 8 adjacent bits. In the EBCDIC coding system, 1 byte can represent a character. In the double-byte coding system, 2 bytes represent a character.

calling program. In the RM/COBOL-85 for the AS/400 licensed program, a program that starts a CALL to another program.

character field. An area that is reserved for information that can contain any of the characters in the character set. Contrast with *numeric field*.

character set. A group of characters used for a specific reason; for example, the set of characters the computer can display, the set of characters a printer can print, or a particular set of graphic characters in a code page; for example, the 256 EBCDIC characters.

character string. A sequence of consecutive characters that are used as a value.

character variable. Character data whose value is assigned and/or changed while the program is running.

checksum protection. (1) A function that protects data stored in the system auxiliary storage pool from being lost because of the failure of a single disk. When checksum protection is in effect and a disk failure occurs, the system automatically reconstructs the data when the system program is loaded after the device is repaired. (2) In TCP/IP, the sum of a group of data associated with the group and used for error checking purposes.

CL. See *control language (CL)*.

command definition. An object that contains the definition of a command (including the command name, parameter descriptions, and validity checking information) and identifies the program that performs the function requested by the command. The system-recognized identifier for the object type is *CMD.

command definition statement. A source statement that defines keywords and parameter values, qualified names, elements in a list, parameter requirements and interrelationships, and prompt text for a command. Command definition statements are used to create a CL command.

command length. In Systems Application Architecture* Query, one of the arguments passed to the language-specific interface programs that specifies the length of the query command to be run.

command level. Pertaining to an operation that is performed for a specific command in a program. For example, a Monitor Message (MONMSG) command that immediately follows a specific command in a CL program is a command-level MONMSG command. Contrast with *program level*.

command name. In Systems Application Architecture Query, the verb in a query command that specifies the action to be performed.

command processing program (CPP). A program that processes a command. This program performs some validity checking and processes the command so that the requested function is performed.

command string. In Systems Application Architecture Query, a character string that contains a query command.

comment. (1) Source program information that is not translated by the compiler. A comment consists of the characters (on one or more lines) delimited by /* and */. (2) In the C and REXX languages, a token consisting of characters (on one or more lines) delimited by /* and */. Comments can be written anywhere in the program.

compilation. Translation of a source program (such as RPG/400 or COBOL specifications) into a program in machine language.

compile. To translate a program written in a high-level programming language into a machine-language program.

compile time. The time during which a source program is translated by a compiler into a machine-language program.

compiler. A program that translates programming language into machine language for use by the computer.

compiler listing. A printout that is produced by compiling a program or creating a file and that optionally includes, for example, a line-by-line list of the high-level language source, a cross-reference list, diagnostic information; and for programs, the description of the externally described files. See also *source listing*.

completion message. A message that tells the operator when work is successfully ended.

concatenate. (1) To link together. (2) To join two character strings.

conditional expression. A statement that compares the relationship (such as greater than or equal) of two items.

conditional prompting. Pertaining to prompting that is provided by the system depending on the values selected by the user for other parameters. Contrast with *selective prompting*.

configuration. The physical and logical arrangement of devices and programs that make up a data processing system.

console. A display station from which an operator can control and observe the system operation. For example, an operator can install the operating system, do an attended IPL, or sign on the system after using the End System (ENDSYS) command.

constant. (1) Data that has an unchanging, predefined value to be used in processing. (2) In RPG, data that has an unchanging, predefined value to be used in processing. A constant does not change during the running of a program, but the contents of a field or variable can. See also *literal*.

control language (CL). The set of all commands with which a user requests system functions.

control language (CL) program. A program that is created from source statements consisting entirely of control language commands.

controlling subsystem. The interactive subsystem that is automatically started first when the system is started and through which the system operator controls the system.

CPP. See *command processing program (CPP)*.

creation date. The system date when an object is created. See also *job date*, and *system date*.

current library. The library that is specified to be the first user library searched for objects requested by a user. The name for the current library can be specified on the Sign-On display or in a user profile. When you specify an object name (such as the name of a file or program) on a command, but do not specify a library name, the system searches the libraries in the system part of the library list, then searches the current library before searching the user part of the library list. The current library is also the library that the system uses when you create a new object, if you do not specify a library name.

current release. The latest available release of the system that replaced the licensed internal code and/or the operating system.

data area. A storage area used to communicate data such as CL variable values between the programs within a job and between jobs. The system-recognized identifier for the data area is *DTAARA.

data authority. A specific authority to read, add, update, or delete data. See also *add authority*, *delete authority*, and *read authority*.

data description specifications (DDS). A description of the user's database or device files that is entered into the system in a fixed form. The description is then used to create files.

data queue. An object that is used to communicate and store data used by several programs in a job or between jobs. The system-recognized identifier is *DTAQ.

database file. One of several types of the system object type *FILE kept in the system that contains descriptions of how input data is to be presented to a program from internal storage and how output data is to be presented to internal storage from a program. See also *physical file* and *logical file*.

DBCS. See *double-byte character set (DBCS)*.

DBCS conversion. A function of the operating system that allows a DBCS display station user to enter alphanumeric data and request that the alphanumeric data be converted to double-byte data.

DDS. See *data description specifications (DDS)*.

deallocate. To release a resource that is assigned to a specific task.

debug mode. An environment in which programs can be tested.

decimal position. (1) The location of the decimal point in a series of numbers. (2) Numbers to the right of the decimal point. For example, 4.009 has three decimal positions.

default. A value that is automatically supplied or assumed by the system or program when no value is specified by the user.

default program. A user-specified program that is assumed when no other program is specifically named on a debug command, or a user-defined program for handling error messages.

default reply. A system-assigned reply to an inquiry or notify message, which is used when the message queue at which the message arrives is in default delivery mode.

delete authority. A data authority that allows the user to remove entries from an object; for example, delete messages from a message queue or delete records from a file. Contrast with *add authority*. See also *read authority*.

diagnostic. Pertaining to the detection and isolation of an error.

diagnostic message. A message that contains information about errors or possible errors. This message is generally followed by an escape message.

display file. A device file created by the user to support a display station.

do group. (1) A set of commands in a control language program defined by a DO command and an ENDDO command that is conditionally processed as a group. (2) In RPG, a group of calculations done one or more times based on the results of comparing factor 1 and factor 2 of certain calculation operations (for example, DOUXX). A DO operation and an END operation are the delimiters for a do group.

double-byte character set (DBCS). A set of characters in which each character is represented by 2 bytes. Languages such as Japanese, Chinese, and Korean, which contain more symbols than can be represented by 256 code points, require double-byte character sets. Because each character requires 2 bytes, the typing, displaying, and printing of DBCS characters requires hardware and programs that support DBCS. Four double-byte character sets are supported by the system: Japanese, Korean, Simplified Chinese, and Traditional Chinese. Contrast with *single-byte character set*.

dump. (1) In problem analysis and resolution, to write, at a particular instant, all or part of the contents of main or auxiliary storage onto another data medium for the purpose of protecting the data or collecting error information. (2) To copy data from main or auxiliary storage onto an external medium, such as tape, diskette, or printer. (3) Data copied in a readable format from main or auxiliary storage to an external medium such as tape, diskette, or printer.

EBCDIC. See *extended binary-coded decimal interchange code (EBCDIC)*.

EBCDIC character. Any one of the symbols included in the 8-bit EBCDIC set.

error log. A record of machine checks, device errors, and media statistics.

escape message. A message that reports a condition that caused the program to end before the requested function was complete.

expression. In REXX, a general process for combining one or more pieces of data in various ways to produce a result, usually different from the original data. Expressions consist of terms (literal strings, function calls, and symbols), and zero or more operators.

general-purpose library. The library shipped with the system that contains IBM-provided objects required for many system functions and user-created objects that are not explicitly placed in a different library when they are created. Named QGPL.

generate. In CSP/AD, to produce a representation of an application program, map group, or table that can be interpreted by Cross System Product/Application Execution.

generic name. The characters common to object names that can be used to identify a group of objects. A generic name ends with an asterisk (*). For example, ORD* identifies all objects whose names begin with the characters ORD.

group data area. A data area that is automatically created when an interactive job becomes a group job. This data area is shared by all jobs in the group but cannot be used by jobs outside the group.

group job. One of up to sixteen interactive jobs that are associated in a group with the same work station device and user.

hex. See *hexadecimal*.

hexadecimal. Pertaining to a numbering system with a base of 16.

high-level language (HLL). A programming language, such as RPG, BASIC, PL/I, Pascal, COBOL, and C used to write computer programs.

high-level language (HLL) pointer. A source pointer that the programmer declares in the user program.

history log. A summary of the system activities, such as system and job information, device status, system operator messages, and a record of program temporary fix (PTF) activity on the system. The history log is identified by the name QHST, and the system-recognized identifier for the object type is *MSGQ.

HLL. See *high-level language (HLL)*.

hold delivery. The method of delivering messages to a message queue that holds the messages until the user requests them. The user is not notified when a message arrives.

immediate message. A message that is created when it is sent. Contrast with *predefined message*.

informational message. A message that provides information to the user about the system as compared with a completion message, which indicates success, and escape or diagnostic messages, which indicate failure.

initial program. A user-profile program that runs when the user signs on and after the command processor program QCMD is started. QCMD calls the first program.

initial program load (IPL). The process that loads the system programs from the system auxiliary storage, checks the system hardware, and prepares the system for user operations.

initialize. To set the addresses, switches, or the contents of storage to zero, or to the starting value set by the manufacturer.

inquiry message. A message that gives information and requests a reply.

interactive. Pertaining to the exchange of information between people and a computer. Contrast with *batch*.

interactive job. A job started for a person who signs on to a work station. Contrast with *batch job*.

interactive mode. In Systems Application Architecture Query, the query mode associated with a query instance that allows users to interact with the query commands while a procedure is running.

IPL. See *initial program load (IPL)*.

job description. A system object that defines how a job is to be processed. The object name is *JOBD.

job log. A record of requests submitted to the system by a job, the messages related to the requests, and the actions performed by the system on the job. The job log is maintained by the system program.

job message queue. A message queue that is created for each job. A job message queue receives requests to be processed (such as commands) and sends messages that result from processing the requests. A job message queue consists of an external message queue and a set of program message queues. See also *external message queue* and *program message queue*.

job queue. An object that contains a list of batch jobs waiting to be processed by the system. The system-recognized identifier for the object type is *JOBQ.

keyword. (1) A name that identifies a parameter in a command. (2) In REXX, a symbol reserved for use by the language processor in a certain context. Keywords include the names of the instructions and ELSE, END, OTHERWISE, THEN, and WHEN. (3) In RPG, a word that is essential to the meaning and structure of a statement in a programming language.

label. (1) An identifier of a command or program statement generally used for branching. (2) In REXX, a clause that consists of a single symbol followed by a colon. (3) In RPG, a symbolic name that represents a specific location in a program. A label can serve as the destination point for one or more branching operations.

leading zeros. In OfficeVision/400, zeros that are place holders to the left of numbers that are aligned to the right and have fewer positions than the specified field length.

library. An object on disk that serves as a directory to other objects. A library groups related objects, and allows the user to find objects by name. The system-recognized identifier for the object type is *LIB.

library list. A list that indicates which libraries are to be searched and the order in which they are to be searched. The system-recognized identifier is *LIBL.

licensed program. A separately orderable program, supplied by IBM, that performs functions related to processing user data. Examples of licensed programs are AS/400 PC Support, AS/400 BASIC, AS/400 Application Development Tools, OfficeVision/400, and so on.

local data area. A 1024-byte data area that can be used to pass information between programs in a job. A separate local data area is automatically created for each job.

lock state. A condition defined for an object that determines how it is locked, how it is used (read or write), and whether the object can be shared (used by more than one job).

logical expression. An expression consisting of logical operators and/or relational operators that can be evaluated to a value of either true or false.

logical operator. (1) A word or symbol that defines the logical connection between conditions or that makes opposite a condition. (2) In REXX, an operator that performs logical operations on one or two terms. The logical operators include: AND, inclusive OR, exclusive OR, or logical NOT. A value of 1 is returned if the expression is true and 0 if the expression is false.

member. Different sets of data within one database file. See also *source member*.

message file. An object that contains message descriptions. The system-recognized identifier for the object type is *MSGF.

message identifier. A seven-character code that identifies a predefined message, and is used to get the message description from a message file. See *predefined message*.

message queue. A list on which messages are placed when they are sent to a person or program. The system-recognized identifier for the object type is *MSGQ.

message reference key. A key assigned to every message on a message waiting line. This key is used to remove a message from a message waiting line, to receive a message, and to reply to a message.

message subfile. A subfile where the records are messages from a program message queue.

mirrored protection. A function that protects data by duplicating all disk data in an auxiliary storage pool (ASP) to a mirrored unit in the same ASP on another disk unit. If a disk failure occurs, the system will keep running, using the mirrored unit of the mirrored pair until the disk unit is repaired or replaced.

mixed list. A list of unlike values for a parameter that accepts a set of separately defined values. Contrast with *simple list*.

multinational character set (MNCS). An expanded set of 188 characters available to countries with supported language groups.

national language support (NLS). The modification or conversion of a United States English product to conform to the requirements of another language or country. This can include the enabling or retrofitting of a product and the translation of nomenclature, online information, or documentation of a product.

nest. To incorporate a structure or structures into a structure of the same kind; for example, one call instruction (nested call) within another call instruction (nesting call) or one subroutine (nested subroutine) within another subroutine (nesting subroutine).

network. A collection of data processing products connected by communications lines for exchanging information between stations.

notify delivery. The method of delivering messages to a message queue in which the work station user is notified that a message arrived. The signal is a light or an audible alarm.

notify message. A message that describes a condition for which a program requires a reply from the calling program, or for which a reply is automatically sent to the program.

null value. A parameter position for which no value is specified.

numeric constant. The actual numeric value to be used in processing, instead of the name of a field containing the data. A numeric constant can contain any of the numeric digits 0 through 9, a sign (plus or minus), and a decimal point.

numeric field. An area that is reserved for a particular unit of information and that can contain only the digits 0 through 9. Contrast with *character field*.

numeric variable. The name of a numeric data item whose value is assigned or changed during program processing.

object. A named storage space that consists of a set of characteristics that describe itself and, in some cases, data. An object is anything that exists in and occupies space in storage and on which operations can be performed. Some examples of objects are programs, files, libraries, and folders.

object authority. A specific authority that controls what a system user can do with an entire object. For example, object authority includes deleting, moving, or renaming an object. There are three types of object authorities: object operational, object management, and object existence.

object existence authority. An object authority that allows the user to delete the object, free storage of the object, save and restore the object, transfer ownership of the object, and create an object that was named by an authority holder.

object management authority. An object authority that allows the user to specify the authority for the object, move or rename the object, and add members to database files.

object operational authority. An object authority that allows the user to look at the description of an object and use the object as determined by the user's data authorities to the object. See also *all authority* and *use authority*.

object type. In Systems Application Architecture Query, the substring following the query command name that specifies the type of query object to be processed.

operator. In the C, COBOL, and REXX languages, a token that specifies the type of action to be done on one or more terms. The four types of operators are: concatenation, arithmetic, comparison, and logical.

output queue. An object that contains a list of spooled files to be written to an output device, such as a printer or a diskette. The system-recognized identifier for the object type is *OUTQ.

override. (1) To specify attributes at run time that change the attributes specified in the file description or in the program. (2) The attributes specified at run time that change the attributes specified in the file description or in the program.

owner. The user who creates an object (or is named the owner of an object).

packed field. A field that contains data in the packed decimal format.

parameter. (1) A value supplied to a command or program that is used either as input or to control the actions of the command or program. (2) In REXX, a variable that is given a constant value for a specified application and that can indicate the application.

parameter list. A list of values that provide a means of associating addressability of data defined in a called program with data in the calling program. It contains parameter names and the order in which they are to be associated in the calling and called program.

parse. In REXX, to split a sequence of characters with a template and to assign the resultant values to variables.

pool. A division of main or auxiliary storage. See also *base pool*, *storage pool*.

predefined message. A message whose description is created and stored in a message file before it is sent by the program. Contrast with *immediate message*.

previous release. The last required release of the system (such as Release 1.0) prior to the current release (such as Release 2.0), including any modification levels (such as Release 1.0 Modification Level 1 or Modification Level 2) that were not required.

problem management. The part of the AS/400 Systems Management Utilities that allows the system to record, analyze, and manage problems and to report problems and/or alerts to the provider of service.

program level. Pertaining to an operation that is performed for an entire program. For example, a Monitor Message (MONMSG) command that immediately follows the last declare command in a CL program is a program-level MONMSG command. Contrast with *command level*.

program message queue. An object used to hold messages that are sent between program calls of a routing step. The program message queue is part of the job message queue.

program object. One of two machine object classifications. It includes those objects used in programs that get their definition from an object definition table. Program objects are used as the parameter or values of machine instructions. Contrast with *system object*.

program stack. A list of programs linked together as a result of programs calling other programs with the CALL instruction, or implicitly from some other event, within the same job.

program variable. A named changeable value that can exist only within programs. Its value cannot be obtained or used when the program that contains it is no longer running.

programming development manager (PDM). A part of the AS/400 Application Development Tools licensed program that allows users to perform several operations (such as copy, delete, and rename) from lists of libraries, objects, and members. PDM also allows users to create user-defined options to perform operations.

QCMD. The IBM-supplied control language processor that interprets and processes CL commands for the system.

QGPL. See *general-purpose library*.

qualified job name. A job name and its associated user name and a system-assigned job number.

qualified name. The name of the library containing the object and the name of the object.

qualifier. In data processing, all names in a qualified name other than the far right, which is called the simple name.

read authority. A data authority that allows the user to look at the contents of an entry in an object or to run a program. See also *add authority* and *delete authority*.

recursion. A programming technique in which a program or routine calls itself to perform each successive step in the solution of a problem and uses the output returned from this call in completing the current step.

recursion level. The position of a program in a program stack. The first occurrence of a program in a job has a recursion level of 1, the second occurrence of the same program has a recursion level of 2, and so on.

relational expression. A logical statement that describes the relationship (such as greater than or equal to) of two arithmetic expressions or data items.

request message. A message that requests a function from the receiving program.

REstructured eXtended eXecutor (REXX) language. A general-purpose programming language, particularly suitable for CL commands, or programs for personal computing. Procedures and programs written in this language can be interpreted by the 400/REXX interpreter.

return code. For printer files, display files, and ICF files, a value sent by the system to a program to indicate the results of an operation by that program.

REXX. See *REstructured eXtended eXecutor (REXX) language*.

SBCS. See *single-byte character set (SBCS)*.

screen design aid (SDA). A function of the AS/400 Application Development Tools licensed program that helps the user design, create, and maintain displays and menus.

SDA. See *screen design aid (SDA)*.

security officer. A person assigned to control all of the security authorizations provided with the system. A security officer can, for example, remove password or resource security; or add, change, or remove security information about any system user.

selective prompting. A function of the operating system that allows the user to tailor command prompts at a parameter level. Contrast with *conditional prompting*.

separator. In COBOL, a punctuation character used to set apart character-strings.

SEU. See *source entry utility (SEU)*.

severity code. A number that indicates how important a message is. The higher the number, the more serious the condition.

shared-for-read lock state. The lock state for a file in which the file can be shared with another program if the program does not request exclusive use of the file. The predefined value for this lock state is *SHRRD.

shared-for-update lock state. The lock state for a file in which the file can be shared either for update or for read operations with another program. The predefined value for this lock state is *SHRUPD.

simple list. A list of like values; for example, a list of user names. Contrast with *mixed list*.

single-byte character set (SBCS). A character set in which each character is represented by a one-byte code. Contrast with *double-byte character set*.

source entry utility (SEU). A function of the AS/400 Application Development Tools licensed program that is used to create and change source members.

source file. A file created by the specification of FILETYPE(*SRC) on the Create command. A source file can contain source statements for such items as high-level language programs and data description specifications.

source listing. A portion of a compiler listing that contains source statements and, optionally, test results. See also *compiler listing*.

source member. A member of a database source file that contains source statements such as RPG/400, COBOL, BASIC, PL/I, or DDS statements. See also *member*.

source program. A set of instructions that are written in a programming language and must be translated to machine language before the program can be run.

source statement. A statement written in symbols of a programming language. For example, RPG/400, COBOL, BASIC, PL/I, and DDS statements are source statements.

storage pool. A logical division of storage reserved for processing a job or group of jobs.

subfile. A group of records of the same record format that can be displayed at the same time at a display station. The system sends the entire group of records to the display in a single operation and receives the group from the display in another operation.

subsystem. An operating environment, defined by a subsystem description, where the system coordinates processing and resources.

syntax checking. A function of the system, a compiler, the BASIC interpreter, or SEU that checks individual statements for errors in the structure of the statement.

system ASP. The auxiliary storage pool where system programs and data reside. It is the storage pool used if a storage pool is not defined by the user. See also *auxiliary storage pool* and *user ASP*.

system library. The library shipped with the system that contains objects, such as authorization lists and device descriptions created by a user; and the licensed programs, system commands, and any other system objects shipped with the system. The system identifier is QSYS.

system object. One of two machine object classifications. Any of the machine objects shipped with the system or any of the operating system objects created by the system.

system value. Control information for the operation of certain parts of the system. A user can change the system value to define his working environment. System date and library list are examples of system values.

table. See *translation table*.

time stamp. (1) To apply the current system time. (2) The value on an object that indicates the system time at some critical point in the object's history. (3) In AS/400 Query, the identification of the day and time a query report is created, which Query automatically provides on each report.

toggle. To switch between two modes; for example, on a personal computer connected to a network, to switch between the data entry and command entry modes or between stand-alone operation and device emulation.

trace. In REXX, a means of tracking the interpretation of a program. Tracing is primarily used for debugging.

translation table. An object that contains a set of hexadecimal characters used to translate one or more characters of data. The table can be used for translation of data being moved between the system and a device. For example, data stored in one national language character set may need to be displayed or entered on display devices that support a different national language character set. The table can also be used to specify an alternative collating sequence or field translation functions. The system-recognized identifier for the object type is *TBL. See also *table*.

unit. The defined space within disk units that is referred to by the system.

use authority. An object authority that allows the user to run a program or to display the contents of a file. Use authority combines object operational authority and read authority.

user ASP. One or more auxiliary storage pools used to isolate journals, journal receivers, and save files from the other system objects stored in the system ASP. See also *auxiliary storage pool (ASP)* and *system ASP*.

user message queue. A user-created object used to receive messages sent from the system, other users, and application programs.

validity checking. To verify the contents of a field.

Index

U

V

W

Special Characters

Reader's Comments

Application System/400™
Programming:
Control Language Programmer's Guide

Publication No. SC21-8077-2

Use this form to tell us what you think about this manual. If you have found errors in it, or if you want to express your opinion about it (such as organization, subject matter, appearance) or make suggestions for improvement, this is the form to use.

To request additional publications, or to ask questions or make comments about the functions of IBM products or systems, you should talk to your IBM representative or to your IBM authorized remarketer. This form is provided for comments about the information in this manual and the way it is presented.

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

Be sure to print your name and address below if you would like a reply.

Name

Address

Company or Organization

Phone No.

Reader's Comments
SC21-8077-2

IBM®

Fold and Tape **Please do not staple** Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

International Business Machines Corporation
Information Development
Department 245
3605 North Hwy 52
ROCHESTER MN 55901-9986

Fold and Tape **Please do not staple** Fold and Tape

Reader's Comments

Application System/400™
Programming:
Control Language Programmer's Guide

Publication No. SC21-8077-2

Use this form to tell us what you think about this manual. If you have found errors in it, or if you want to express your opinion about it (such as organization, subject matter, appearance) or make suggestions for improvement, this is the form to use.

To request additional publications, or to ask questions or make comments about the functions of IBM products or systems, you should talk to your IBM representative or to your IBM authorized remarketer. This form is provided for comments about the information in this manual and the way it is presented.

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

Be sure to print your name and address below if you would like a reply.

Name

Address

Company or Organization

Phone No.

Reader's Comments
SC21-8077-2

Fold and Tape | **Please do not staple** | Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

International Business Machines Corporation
Information Development
Department 245
3605 North Hwy 52
ROCHESTER MN 55901-9986

Fold and Tape | **Please do not staple** | Fold and Tape

Reader's Comments

Application System/400™
Programming:
Control Language Programmer's Guide

Publication No. SC21-8077-2

Use this form to tell us what you think about this manual. If you have found errors in it, or if you want to express your opinion about it (such as organization, subject matter, appearance) or make suggestions for improvement, this is the form to use.

To request additional publications, or to ask questions or make comments about the functions of IBM products or systems, you should talk to your IBM representative or to your IBM authorized remarketer. This form is provided for comments about the information in this manual and the way it is presented.

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

Be sure to print your name and address below if you would like a reply.

Name

Address

Company or Organization

Phone No.

Reader's Comments
SC21-8077-2

Fold and Tape | **Please do not staple** | Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

International Business Machines Corporation
Information Development
Department 245
3605 North Hwy 52
ROCHESTER MN 55901-9986

Fold and Tape | **Please do not staple** | Fold and Tape

Program Number
5728-SS1

21F2720

SC21-8077-2

Printed in U.S.A.