z/OS
2.5

TSO/E Programming Services

SA32-0973-50

Note

Before using this information and the product it supports, read the information in “Notices” on page 501.

This edition applies to Version 2 Release 5 of z/OS® (5650-ZOS) and to all subsequent releases and modifications until otherwise indicated in new editions.

Last updated: 2023-03-29

Contents

Figures

Tables

About this document

This document supports z/OS (5650-ZOS).

This book describes the services that TSO/E provides for use in writing system and application programs.

Who should use this document

This book is intended for:

- Application programmers who design and write programs that run under TSO/E.
- System programmers who must modify TSO/E to suit the needs of their installation.

The reader must be familiar with MVS™ programming conventions, the assembler language, and the structure of TSO/E.

Before using this book, read *z/OS TSO/E Programming Guide* which describes how to write a Command Processor and how to compile, assemble, execute and test a program in the TSO/E environment.

How this document is organized

The chapters of this book and their purposes are as follows:

- Chapter 1, “Introduction,” on page 1 gives an overview of the services provided by TSO/E and discusses the types of programs that can be written using TSO/E.
- Chapter 2, “Considerations for using TSO/E services,” on page 9 describes how to determine the version and release of TSO/E installed on your system, and explains programming considerations for MVS and the interface to the TSO/E service routines.
- Chapter 3, “Using the TSO/E Environment Service IKJTSOEV,” on page 17 describes how to use the TSO/E Environment Service to establish a TSO/E environment outside of the TSO/E TMP and Service Routines.
- Chapter 4, “Invoking TSO/E service routines with CALLTSSR,” on page 33 describes how to use the CALLTSSR macro instruction to invoke certain TSO/E service routines.
- Chapter 5, “Verifying subcommand names with IKJSCAN,” on page 35 describes how to validate command and subcommand names.
- Chapter 6, “Verifying command and subcommand operands with parse,” on page 43 describes how to validate command and subcommand operands.
- Chapter 7, “Using the terminal control macro instructions,” on page 145 describes how to control terminal functions and attributes.
- Chapter 8, “Using BSAM or QSAM for terminal I/O,” on page 169 describes how to use the basic sequential and queued sequential access methods in programs that operate under TSO/E.
- Chapter 9, “Using the TSO/E I/O service routines for terminal I/O,” on page 173 describes how to use the STACK, GETLINE, PUTLINE and PUTGET service routines in a Command Processor.
- Chapter 10, “Using the TGET/TPUT/TPG macro instructions for terminal I/O,” on page 257 describes how to use the TGET/TPUT/TPG macro instructions in a program to perform terminal I/O.
- Chapter 11, “Using the TSO/E message handling routine IKJEFF02,” on page 275 describes how to use IKJEFF02 in a Command Processor to issue messages.
- Chapter 12, “Using the STAX service routine to handle attention interrupts,” on page 285 describes how to use the STAX service routine in a program to process attention interruptions.
- Chapter 13, “Using the CLIST attention facility,” on page 293 describes how to use the CLIST attention facility to process a CLIST's attention exit.

- Chapter 14, “Obtaining a List of data set names,” on page 297 describes how a program can use ICQGCL00 to obtain a list of data set names that match specified criteria.
- Chapter 15, “Using the space management CLIST ICQSPC00,” on page 303 describes how a program can use ICQSPC00 to ensure that data sets have adequate free space.
- Chapter 16, “Using IKJADTAB to change alternative library environments,” on page 311 describes how to use IKJADTAB to create and remove alternative library environments and to modify alternative library definitions.
- Chapter 17, “Using the dynamic allocation interface routine DAIR,” on page 321 describes how to use DAIR in a Command Processor to allocate, free, concatenate and deconcatenate data sets during program execution.
- Chapter 18, “Using IKJEHCIR to retrieve system catalog information,” on page 343 describes how to use IKJEHCIR to retrieve information from the system catalog.
- Chapter 19, “Constructing a fully-qualified data set name with IKJEHDEF,” on page 349 describes how a Command Processor can use IKJEHDEF to construct a fully-qualified data set name.
- Chapter 20, “Using the DAIRFAIL routine IKJEFF18,” on page 353 describes how to use the DAIRFAIL routine to analyze return codes from dynamic allocation (SVC 99) or DAIR.
- Chapter 21, “Analyzing error conditions with GNRLFAIL/VSAMFAIL,” on page 357 describes how to use the GNRLFAIL/VSAMFAIL routine to analyze error conditions and issue appropriate error messages.
- Chapter 22, “Using the table look-up service IKJTBLS,” on page 361 describes how to use the table look-up service to search the lists of authorized commands and programs and commands not supported in the background.
- Chapter 23, “Using the TSO/E Service Facility IKJEFTSR,” on page 365 describes how an unauthorized program can use the TSO/E Service Facility to invoke other programs, commands, REXX execs and CLISTs, regardless of whether the invoked function is authorized.
- Chapter 24, “Using the variable access routine IKJCT441,” on page 409 describes how a program can use IKJCT441 to examine and manipulate CLIST and REXX variables.
- Chapter 25, “Accessing the Information Center Facility names directory,” on page 431 describes how to use TSO/E program ICQCAL00 to access the Information Center Facility names directory.
- Chapter 26, “Using the printer support CLISTs,” on page 443 describes how to use the printer support CLISTs to select printers and print data sets on selected printers.
- Chapter 27, “Invoking an Information Center Facility application,” on page 475 describes how to use the application invocation function to invoke an application that is integrated into the Information Center Facility.
- Chapter 28, “Using the GETMSG service,” on page 479 describes how to use the GETMSG service to retrieve system messages issued during a console session.
- Chapter 29, “Using the Unauthorized Resource Processor Service IKJURPS,” on page 485 describes how applications that execute in a TSO/E environment can get control within the TSO/E terminal monitor program (TMP).
- Appendix A, “Limits for TSO/E service routines,” on page 497 describes the limits imposed by TSO/E services.

How to use this document

If you have never used this book, read Chapter 1, “Introduction,” on page 1 to become familiar with the programming services that TSO/E provides. Then read the chapter that discusses the service you want to use.

Where to find more information

Please see *z/OS Information Roadmap* for an overview of the documentation associated with z/OS, including the documentation available for z/OS TSO/E.

How to send your comments to IBM

We invite you to submit comments about the z/OS product documentation. Your valuable feedback helps to ensure accurate and high-quality information.

Important: If your comment regards a technical question or problem, see instead "If you have a technical problem" on page xxix.

Submit your feedback by using the appropriate method for your type of comment or question:

Feedback on z/OS function
If your comment or question is about z/OS itself, submit a request through the IBM RFE Community (www.ibm.com/developerworks/rfe/).

Feedback on IBM® Documentation function
If your comment or question is about the IBM Documentation functionality, for example search capabilities or how to arrange the browser view, send a detailed email to IBM Documentation Support at ibmdocs@us.ibm.com.

Feedback on the z/OS product documentation and content
If your comment is about the information that is provided in the z/OS product documentation library, send a detailed email to mhvrcfs@us.ibm.com. We welcome any feedback that you have, including comments on the clarity, accuracy, or completeness of the information.

To help us better process your submission, include the following information:

- Your name, company/university/institution name, and email address
- The following deliverable title and order number: z/OS TSO/E Programming Services, SA32-0973-50
- The section title of the specific information to which your comment relates
- The text of your comment.

When you send comments to IBM, you grant IBM a nonexclusive authority to use or distribute the comments in any way appropriate without incurring any obligation to you.

IBM or any other organizations use the personal information that you supply to contact you only about the issues that you submit.

If you have a technical problem

If you have a technical problem or question, do not use the feedback methods that are provided for sending documentation comments. Instead, take one or more of the following actions:

- Go to the IBM Support Portal (support.ibm.com).
- Contact your IBM service representative.
- Call IBM technical support.

Summary of changes

This information includes terminology, maintenance, and editorial changes. Technical changes or additions to the text and illustrations for the current edition are indicated by a vertical line to the left of the change.

Note: IBM z/OS policy for the integration of service information into the z/OS product documentation library is documented on the z/OS Internet Library under IBM z/OS Product Documentation Update Policy (www-01.ibm.com/servers/resourcelink/svc00100.nsf/pages/ibm-zos-doc-update-policy?OpenDocument).

Summary of changes for z/OS TSO/E Programming Services for Version 2 Release 5 (V2R5)

The following content is new, changed, or no longer included in V2R5.

New

The following content is new.

- None

Changed

The following content is changed.

- None

Deleted

The following content was deleted.

- None

Summary of changes for z/OS Version 2 Release 4 (V2R4)

The following changes are made for z/OS Version 2 Release 4 (V2R4).

New

- Positional operands for IKJPOSIT USERID8 and UID82PWD, are updated to allow special characters within passwords. See “Delimiter-dependent operands” on page 51 and “Acceptance of double-byte character set data” on page 46 for more information.
- In “Return and reason codes from the TSO/E Environment Service” on page 23, new return code, 44, is added. See Table 6 on page 23.

Changed

None.

Deleted

- z/OS support for Server-Requester Programming Interface (SRPI) and TSO/E Enhanced Connectivity Facility are discontinued. The information about Enhanced Connectivity Facility is removed.

Summary of changes for z/OS Version 2 Release 3 (V2R3)

The following changes are made for z/OS Version 2 Release 3 (V2R3).

New

- The limit for TSO/E user IDs is changed to 8 characters. For more information, see:
  - “Delimiter-dependent operands” on page 51
  - “Acceptance of double-byte character set data” on page 46
  - “Delimiter-dependent operands” on page 51
  - “Entering positional operands as lists of ranges” on page 64
  - “Using IKJPOSIT to describe a delimiter-dependent positional operand” on page 68
  - “The parameter control entry built by IKJPOSIT” on page 71
  - New positional operands for IKJPOSIT, USERID8 and UID82PWD, are added to “PDEs created for positional operands described by IKJPOSIT” on page 108

Changed

- The ANY operand is updated. See “Using IKJIDENT to describe a non-delimiter-dependent positional operand” on page 83.

Chapter 1. Introduction

TSO/E provides programming services that you can use in system or application programs. These services consist of programs, macros, and CLISTs.

TSO/E services support a wide range of functions that are useful in writing system programs as well as application programs that exploit the full-screen capabilities of TSO/E.

CLISTs, REXX execs, servers and *command processors* are specific types of programs that you can write to run in the TSO/E environment. The following topic, "Programming using TSO/E" on page 1, gives an overview of these types of programs, and refers you to the appropriate book in the TSO/E library for more information.

"Overview of TSO/E programming services" on page 2 describes the TSO/E services documented in this book.

Programming using TSO/E

You can write programs to run in the TSO/E environment and use the services provided by TSO/E. Specific types of programs that run under TSO/E are CLISTs, REXX execs, servers and command processors. These types of programs are discussed in the following topics.

Writing CLISTs

The CLIST language is a high-level interpretive language that enables you to work more efficiently with TSO/E. You can write programs, called CLISTs (or command procedures), that perform given tasks or groups of tasks. CLISTs can handle any number of tasks, from issuing multiple TSO/E commands to invoking programs written in other languages.

Because the CLIST language is an interpretive language, CLISTs are easy to test and do not require you to compile or link-edit them. To test a CLIST, you simply execute it, correct any errors, and re-execute it.

The CLIST language supports a range of programming functions including:

- CLIST statements that allow you to write structured programs, perform I/O, define and modify variables, and handle errors and attention interruptions.
- Arithmetic and logical operators for processing numeric data.
- String-handling functions for processing character data.

CLISTs can perform a range of tasks. For example,

- CLISTs can perform routine tasks, such as allocating data sets that are required for particular programs.
- The CLIST language enables you to write structured applications by using subprocedures within a CLIST, invoking other CLISTs, defining common data for subprocedures and CLISTs, and passing parameters between CLISTs or subprocedures.

  CLISTs allow you to easily write interactive applications by issuing commands of the Interactive System Productivity Facility (ISPF) to display full-screen panels.
- CLISTs can provide interfaces, which are easy to use, to applications written in other languages. CLISTs can prompt terminal users for information on the tasks they request, set up the environment needed for the application, and then issue the commands needed to invoke the application program.

For information on creating, executing, and testing CLISTs, see *z/OS TSO/E CLISTs*.

Writing REXX Execs

Restructured extended executor (REXX) is a high-level interpretive language that enables you to write programs in a clear and structured way. You can write programs in the REXX language, called execs, that perform given tasks or groups of tasks.

REXX execs have many characteristics that are similar to CLISTs. For example, using either the REXX or CLIST language, you can:

- Perform numerous tasks, including issuing multiple TSO/E commands and invoking programs written in other languages.
- Write structured programs, perform I/O and process arithmetic and character data.
- Write interactive applications by issuing commands of ISPF to display full-screen panels.
- Provide easy-to-use interfaces to applications written in other languages. Execs can prompt the terminal user for information on the tasks the user requests, set up the environment needed for the application, and then issue the commands needed to invoke the application program.

However, a significant difference between execs and CLISTs is that you can execute CLISTs only in a TSO/E environment. REXX execs do not require a TSO/E environment, and can execute in any MVS address space. In addition, you can use the Systems Application Architecture® (SAA) Procedures Language to write execs that are system independent. The SAA Procedures Language, which is a subset of the REXX language, enables you to write execs to run in multiple host environments.

TSO/E Release 3 extended the REXX language to provide host environments that support the common programming interface (CPI) and LU 6.2-based APPC/MVS callable services. Using these host environments, you can write a REXX exec to act as an APPC/MVS transaction program or communicate with other APPC/MVS transaction programs. Because these transaction programs can be used across different machines, greater system connectivity is possible.

For information on writing and executing execs, see *z/OS TSO/E User's Guide* and *z/OS TSO/E REXX Reference*.

Writing command processors

TSO/E provides commands that you can use to perform a wide variety of tasks. For example, you can use TSO/E commands to define and maintain data sets, and write and test programs.

You can write *command processors* to replace or add to this set of commands. By writing your own command processors, your installation can add to or modify TSO/E to better suit the needs of its users.

A Command Processor is a program that receives control when a user at a terminal enters a command name. It is given control by the terminal monitor program (TMP), a program that provides an interface between terminal users and command processors, and has access to many system services.

The main difference between command processors and other programs is that when a Command Processor is invoked, it is passed a Command Processor parameter list (CPPL) that gives the program access to information about the caller and to system services.

Command processors must be able to communicate with the user at the terminal, as well as respond to abnormal terminations and attention interruptions. Command processors can recognize subcommand names entered by the terminal user and then load and pass control to the appropriate subCommand Processor.

You can use many of the services documented in this book to write a Command Processor. For guidelines on how to write a Command Processor, what TSO/E services to use, and how to test and install the Command Processor, see *z/OS TSO/E Programming Guide*.

Overview of TSO/E programming services

TSO/E provides various services that your programs can use to perform the tasks described below. Table 1 on page 3 summarizes the TSO/E services that are described in this book. In addition to these

services, TSO/E also provides REXX programming and customization services you can use for REXX processing. These REXX services are explained in *z/OS TSO/E REXX Reference*.

Table 1. Summary of TSO/E services

Task	Service	Reference
Establishing a TSO/E environment outside of the TSO/E TMP and Service Routines	TSO/E Environment Service	Chapter 3, "Using the TSO/E Environment Service IKJTSOEV," on page 17
Invoking TSO/E service routines	CALLTSSR macro instruction	Chapter 4, "Invoking TSO/E service routines with CALLTSSR," on page 33
Checking the syntax of subcommand names	Command Scan Service Routine	Chapter 5, "Verifying subcommand names with IKJSCAN," on page 35
Checking the syntax of command and subcommand operands	Parse Service Routine	Chapter 6, "Verifying command and subcommand operands with parse," on page 43
Controlling terminal functions and attributes	Terminal control macro instructions	Chapter 7, "Using the terminal control macro instructions," on page 145
Processing terminal I/O	BSAM and QSAM	Chapter 8, "Using BSAM or QSAM for terminal I/O," on page 169
	TSO/E I/O service routines	Chapter 9, "Using the TSO/E I/O service routines for terminal I/O," on page 173
	TGET/TPUT/TPG macros	Chapter 10, "Using the TGET/TPUT/TPG macro instructions for terminal I/O," on page 257
	TSO/E Message Handling Routine	Chapter 11, "Using the TSO/E message handling routine IKJEFF02," on page 275
Handling attention interruptions	STAX service routine	Chapter 12, "Using the STAX service routine to handle attention interrupts," on page 285
	CLIST attention facility	Chapter 13, "Using the CLIST attention facility," on page 293
Obtaining a list of data set names	ICQGCL00	Chapter 14, "Obtaining a List of data set names," on page 297
Ensuring that data sets contain enough space	Space management	Chapter 15, "Using the space management CLIST ICQSPC00," on page 303
Changing alternative library environments	Alternative library interface routine	Chapter 16, "Using IKJADTAB to change alternative library environments," on page 311
Allocating, concatenating and freeing data sets	Dynamic allocation interface routine	Chapter 17, "Using the dynamic allocation interface routine DAIR," on page 321
Retrieving information from the system catalog	Catalog information routine	Chapter 18, "Using IKJEHCIR to retrieve system catalog information," on page 343

Table 1. Summary of TSO/E services (continued)

Task	Service	Reference
Constructing a fully-qualified data set name	Default service routine	Chapter 19, "Constructing a fully-qualified data set name with IKJEHDEF," on page 349
Analyzing return codes	DAIRFAIL	Chapter 20, "Using the DAIRFAIL routine IKJEFF18," on page 353
	GNRLFAIL/VSAMFAIL	Chapter 21, "Analyzing error conditions with GNRLFAIL/VSAMFAIL," on page 357
Searching lists of authorized commands and programs as well as commands not supported in the background	Table look-up service	Chapter 22, "Using the table look-up service IKJTBLS," on page 361
Invoking commands, CLISTs, REXX execs and programs	TSO/E Service Facility	Chapter 23, "Using the TSO/E Service Facility IKJEFTSR," on page 365
Accessing CLIST and REXX variables	Variable access routine	Chapter 24, "Using the variable access routine IKJCT441," on page 409
Retrieving information from the names directory	ICQCAL00	Chapter 25, "Accessing the Information Center Facility names directory," on page 431
Displaying printers	Printer support CLISTs	Chapter 26, "Using the printer support CLISTs," on page 443
Invoking Information Center Facility applications	Application invocation function	Chapter 27, "Invoking an Information Center Facility application," on page 475
Retrieving system messages issued during a console session	GETMSG service	Chapter 28, "Using the GETMSG service," on page 479
Using the Unauthorized Resource Processor	IKJURPS service	Chapter 29, "Using the Unauthorized Resource Processor Service IKJURPS," on page 485

Invoking TSO/E service routines

To pass control to certain TSO/E service routines, use the CALLTSSR macro instruction. See Chapter 4, "Invoking TSO/E service routines with CALLTSSR," on page 33.

Establishing a TSO/E environment outside of the TSO/E TMP and service routines

You can establish a TSO/E environment outside of the TSO/E TMP and Service Routines using the TSO/E Environment Service (IKJTSOEV). See Chapter 3, "Using the TSO/E Environment Service IKJTSOEV," on page 17.

Checking the syntax of subcommand names

Use the Command Scan Service Routine in your command processors to validate a subcommand name. See Chapter 5, "Verifying subcommand names with IKJSCAN," on page 35.

Checking the syntax of command and subcommand operands

Use the Parse Service Routine to validate command or subcommand operands. See Chapter 6, “Verifying command and subcommand operands with parse,” on page 43.

Communicating with the terminal user

TSO/E provides several services to aid you in communicating with the terminal user.

Controlling terminal functions and attributes

Use the terminal control macro instructions to control terminal functions and attributes, such as full-screen mode and terminal line size. See Chapter 7, “Using the terminal control macro instructions,” on page 145 for a description of each of these macro macro instructions.

Processing terminal I/O

TSO/E offers several services for use in processing terminal I/O and issuing messages.

- Your programs can use the basic sequential access method (BSAM) and the queued sequential access method (QSAM) to provide terminal I/O support. See Chapter 8, “Using BSAM or QSAM for terminal I/O,” on page 169.
- You can use the TSO/E I/O service routines (STACK, GETLINE, PUTLINE and PUTGET) in a Command Processor to control the source of input, and write a line of output or obtain a line of input from the terminal. The I/O service routines can be used to issue messages to the terminal user. See Chapter 9, “Using the TSO/E I/O service routines for terminal I/O,” on page 173.
- You can use the TGET/TPUT/TPG macro instructions to process terminal I/O in any programs you write that run under TSO/E. See Chapter 10, “Using the TGET/TPUT/TPG macro instructions for terminal I/O,” on page 257.
- Your command processors can use the TSO/E message issuer routine (IKJEFF02) to issue messages to the terminal. See Chapter 11, “Using the TSO/E message handling routine IKJEFF02,” on page 275.

Handling attention interruptions

Use the STAX service routine in a program to cause the system to recognize and schedule an attention exit that receives control when an attention interruption occurs. See Chapter 12, “Using the STAX service routine to handle attention interrupts,” on page 285.

Use the CLIST attention facility in a program that processes a CLIST with a CLIST attention exit. This facility allows a program to process the CLIST's attention exit when an attention interruption occurs. See Chapter 13, “Using the CLIST attention facility,” on page 293.

Processing data sets

TSO/E provides several services that your programs can use to process data sets.

Obtaining a list of data set names

Use the TSO/E program (ICQGCL00) to obtain a list of data set names that match specified criteria. See Chapter 14, “Obtaining a List of data set names,” on page 297.

Ensuring that data sets contain sufficient space

Use the space management CLIST (ICQSPC00) in your programs to ensure that a specified data set has adequate free space for additional data. See Chapter 15, “Using the space management CLIST ICQSPC00,” on page 303.

Allocating, concatenating and freeing data sets

TSO/E provides the dynamic allocation interface routine (DAIR) to allocate, free, concatenate and deconcatenate data sets during program execution. *However, because of the reduced function and additional system overhead associated with DAIR, your programs should access dynamic allocation directly*. This book documents DAIR to provide compatibility for existing programs that use it. For a complete discussion of dynamic allocation, see *z/OS MVS Programming: Authorized Assembler Services Guide*. DAIR is discussed in Chapter 17, "Using the dynamic allocation interface routine DAIR," on page 321.

Retrieving information from the system catalog

Use the catalog information routine (IKJEHCIR) to retrieve information from the system catalog, such as data set name, index name, control volume address or volume serial number. See Chapter 18, "Using IKJEHCIR to retrieve system catalog information," on page 343.

Constructing a fully-qualified data set name

Use the default service routine (IKJEHDEF) in your Command Processor to construct a fully-qualified data set name when a partially-qualified data set name is entered by a terminal user. See Chapter 19, "Constructing a fully-qualified data set name with IKJEHDEF," on page 349.

Changing alternative library environments

Use the alternative library interface routine (IKJADTAB) in an application program to create and remove alternative library environments and to modify alternative library definitions for CLIST and REXX libraries. See Chapter 16, "Using IKJADTAB to change alternative library environments," on page 311.

Analyzing return codes

Use the DAIRFAIL routine (IKJEFF18) to analyze return codes from dynamic allocation or DAIR and issue appropriate error messages. See Chapter 20, "Using the DAIRFAIL routine IKJEFF18," on page 353.

Use the GNRLFAIL/VSAMFAIL routine (IKJEFF19) to analyze VSAM macro instruction failures, subsystem request failures, Parse Service Routine or PUTLINE failures, and ABEND codes, and issue an appropriate error message. See Chapter 21, "Analyzing error conditions with GNRLFAIL/VSAMFAIL," on page 357.

Searching system lists

Use the table look-up service (IKJTBLS) to determine if the name of a command or program is present in one of the following lists:

- Names of authorized command processors that the terminal monitor program executes.
- Names of authorized programs that the CALL command executes.
- Names of authorized programs that can be invoked by the TSO/E Service Facility (IKJEFTSR).
- Names of commands not supported in the background.

See Chapter 22, "Using the table look-up service IKJTBLS," on page 361.

Invoking commands, CLISTs, REXX execs and programs

Use the TSO/E Service Facility routine, IKJEFTSR, to invoke programs, commands, CLISTs, and REXX execs. The TSO/E Service Facility allows an unauthorized program to invoke functions that are authorized. Use the TSO/E Service Facility initialization routine (IKJEFTSI) to create a command invocation platform environment for certain unauthorized commands. TSO/E Release 3 offered extended platform support, in which you can invoke authorized commands and authorized and unauthorized programs on a command/program invocation platform. Use the TSO/E Service Facility termination routine, IKJEFTST, to clean up resources allocated to the command or command/program invocation platform environment. See Chapter 23, "Using the TSO/E Service Facility IKJEFTSR," on page 365.

Accessing CLIST and REXX variables

Use the variable access routine (IKJCT441) in your programs to update, create, and return the values of CLIST and REXX variables when running in a TSO/E environment. TSO/E also provides the REXX variable access routine (IRXEXCOM) that lets unauthorized programs and commands access REXX variables from a REXX Language Processor Environment running in any MVS address space. For more information about using IKJCT441, see Chapter 24, “Using the variable access routine IKJCT441,” on page 409. For more information about using IRXEXCOM, see *z/OS TSO/E REXX Reference*.

Retrieving information from the names directory

Use the TSO/E program (ICQCAL00) to search the Information Center Facility's name directory and retrieve information such as phone numbers, user IDs and addresses for specified names. See Chapter 25, “Accessing the Information Center Facility names directory,” on page 431.

Displaying printers for selection by the user

Use the TSO/E printer support CLISTs to display lists of printers for users to select and to print data sets on selected printers. See Chapter 26, “Using the printer support CLISTs,” on page 443.

Invoking an Information Center Facility application

Use the application invocation function, ICQAMLI0, to invoke an application that is defined to the Information Center Facility. See Chapter 27, “Invoking an Information Center Facility application,” on page 475.

Retrieving system messages issued during a console session

Use the GETMSG service to retrieve solicited messages (responses to system commands) and unsolicited messages issued during a console session. See Chapter 28, “Using the GETMSG service,” on page 479.

Coding the macro instructions

The following paragraphs describe the notation used to define the macro syntax in this publication.

1. The set of symbols listed below are used to define macro instructions, but should never be written in the actual macro instruction:

   hyphen
   -

   underscore
   _

   braces
   { }

   brackets
   [ ]

   ellipsis
   . . .

   The special uses of these symbols are explained in paragraphs 4-8.

2. Uppercase letters and words, numbers, and the set of symbols listed below should be written in macro instructions exactly as shown in the definition:

   apostrophe
   '

   asterisk
   *

comma
,

equal sign
=

parentheses
( )

period
.

3. Lowercase letters, words, and symbols appearing in a macro instruction definition represent variables for which specific information should be substituted in the actual macro instruction.

   Example: If **name** appears in a macro instruction definition, a specific value (for example, ALPHA) should be substituted for the variable in the actual macro instruction.

4. Hyphens join lowercase letters, words, and symbols to form a single variable.

   Example: If member-name appears in a macro instruction definition, a specific value (for example, BETA) should be substituted for the variable in the actual macro instruction.

5. An underscore indicates a default option. If an underscored alternative is selected, it need not be written in the actual macro instruction.

   Example: The representation

   ```
   A     {A}
   B  or {B}
   C     {C}
   ```

   indicates that either A or B or C should be selected; however, if B is selected, it need not be written because it is the default option.

6. Braces group related items, such as alternatives.

   Example: The representation

   ```
          {A}
   ALPHA=({B},D)
          {C}
   ```

   indicates that a choice should be made among the items enclosed within the braces. If A is selected, the result is ALPHA=(A,D). If B is selected, the result can be either ALPHA=(,D) or ALPHA=(B,D).

7. Brackets also group related items; however, everything within the brackets is optional and may be omitted.

   Example: The representation

   ```
          [A]
   ALPHA=([B],D)
          [C]
   ```

   indicates that a choice can be made among the items enclosed within the brackets or that the items within the brackets can be omitted. If B is selected, the result is: ALPHA=(B,D). If no choice is made, the result is: ALPHA=(,D).

8. An ellipsis indicates that the preceding item or group of items can be repeated more than once in succession.

   Example: The representation

   ```
   ALPHA[,BETA]…
   ```

   indicates that ALPHA can appear alone or can be followed by ,BETA any number of times in succession.

Note: To designate register 0 and register 1 on a macro invocation, use (0) and (1), respectively. You cannot use a symbolic variable to designate these registers.

Chapter 2. Considerations for using TSO/E services

This topic discusses considerations for MVS/ESA SP that you should be aware of when writing a command processor or using the services documented in this information. You should be familiar with the publications that describe comprehensive programming considerations for z/OS as well as with those that describe the routines and macros discussed in this manual. Interfaces for service routines and macro instructions mentioned in this topic are covered in more detail in the chapters of this manual describing the individual service routines and macro instructions.

Determining the version and release of TSO/E installed

Sometimes you need to know which version and release of TSO/E is installed to determine if a particular function is present on your system. By knowing the version and release of TSO/E, you can decide whether to use the available functions or services in your application programs.

An indication of the version and release of TSO/E installed is stored in a field (TSVTTSOL) in the TSO/E vector table (TSVT). The TSVT is a control block pointed to by the communications vector table (CVT). TSVTTSOL is a four-byte EBCDIC field that contains the TSO/E version and release information in the following format:

Offset dec(Hex)	Number of bytes	Contents or meaning
0(0)	1	Version level
1(1)	2	Release number
3(3)	1	Modification level

If you are using a CLIST application, the CLIST control variable &SYSTSOE contains the TSO/E version and release information.

In a REXX exec, use the TSO/E SYSVAR external function with the variable SYSTSOE to obtain the TSO/E version and release information.

General interface considerations

You need to consider addressing modes, address space control (ASC) modes, and program residency when determining linkage conventions. See “Interface considerations for the TSO/E service routines” on page 10 for brief descriptions of those considerations for the service routines and macro instructions described in this manual.

When making linkage decisions, you should consider:

- Who passes control to whom
- Whether return is desired
- AMODE and RMODE attributes
- Address space control mode attributes.

The following discussion provides a general description of ASC mode, AMODE and RMODE attributes. For a detailed description of ASC mode considerations, see *z/OS MVS Programming: Extended Addressability Guide*. For a detailed description of 31-bit addressing, see *z/OS MVS Programming: Assembler Services Guide*.

AR mode

Access register (AR) mode is the address space control (ASC) mode in which a general register and the corresponding access register (AR) are used together to locate an address in an address/data space.

Specifically, the general register is used as a base register for data reference and the corresponding AR contains a value that identifies the address/data space that contains the data.

Primary mode

Primary mode is the address space control (ASC) mode in which only a general register is used to locate an address in an address space. In primary mode, the contents of the access registers (ARs) are ignored.

AMODE=24, RMODE=24

Programs with these attributes must receive control in 24-bit addressing mode, and are loaded below 16 MB in virtual storage.

If you do not assign AMODE and RMODE attributes to a program, the attributes default to AMODE=24 and RMODE=24. Most IBM-supplied command processors have these attributes and are loaded below 16 MB in virtual storage.

AMODE=ANY, RMODE=24

AMODE=ANY indicates that a program must receive control in the addressing mode of the program that invoked it. Note that a program with the AMODE=ANY attribute might have to switch addressing modes for certain processing. However, such a program must switch back to the addressing mode in which it received control before returning to the caller.

AMODE=ANY programs must be given the RMODE=24 attribute.

AMODE=ANY does not indicate whether the program should be passed input that resides below 16 MB in virtual storage; the particular interfaces should be analyzed to determine where input can reside. However, a program should meet certain criteria to be assigned the AMODE=ANY attribute. For a description of the criteria, see *z/OS MVS Programming: Assembler Services Guide*.

AMODE=31

AMODE=31 indicates that a program must receive control in 31-bit addressing mode. Such a program can have the RMODE=24 or RMODE=ANY attribute, depending on its residency requirements. Regardless of the program's RMODE attribute, the residency of its input depends on the program's requirements. The program might require that some of its input reside below 16 MB in virtual storage, while other input might reside anywhere.

A program that runs exclusively in 31-bit addressing mode (AMODE=31) can do so provided it complies with the restrictions for invoking, and being invoked by, programs that run in 24-bit addressing mode (AMODE=24 or AMODE=ANY).

For more information on the AMODE=31 attribute, see *z/OS MVS Programming: Assembler Services Guide*.

Interface considerations for the TSO/E service routines

All TSO/E service routines documented in this book must receive control in primary address space control mode. These service routines return control in primary mode.

User-written Command Processors can execute in either 24-bit or 31-bit addressing mode provided they follow the restrictions involved in invoking programs that have 24-bit dependencies. When assigned the AMODE=31 attribute, they can be loaded above 16 MB in virtual storage (RMODE=ANY), and passed input that resides above 16 MB.

The Command Processor parameter list (CPPL), which contains certain addresses required as input to the TSO/E service routines, resides below 16 MB in virtual storage. Refer to “Interfacing with the TSO/E service routines” on page 13 for more information on the CPPL.

Table 2 on page 11 shows the interface considerations for the TSO/E service routines.

Table 2. Interface considerations for TSO/E service routines

Service routine	Entry point name	Interface considerations
Catalog information routine Default service routine	IKJEHCIR IKJEHDEF	These routines can be invoked in either 24- or 31-bit addressing mode, but all input passed to these routines must reside below 16 MB in virtual storage. These routines return control in the same addressing mode in which they are invoked.
Dynamic allocation interface routine DAIRFAIL GNRLFAIL/VSAMFAIL TSO/E service facility routine	IKJDAIR IKJEFF18 IKJEFF19 IKJEFTSR	These service routines can be invoked in either 24- or 31-bit addressing mode. When invoked in 31-bit addressing mode, these routines can be passed input that resides above 16 MB in virtual storage. These routines return control in the same addressing mode in which they are invoked.
TSO/E message issuer routine GETLINE service routine Parse service routine PUTGET service routine PUTLINE service routine Command scan service routine STACK service routine Variable access routine Table look-up service	IKJEFF02 IKJGETL IKJPARS IKJPTGT IKJPUTL IKJSCAN IKJSTCK IKJCT441 IKJTBLS	These service routines can be invoked in either 24-bit or 31-bit addressing mode. They can accept input above or below 16 MB in virtual storage. These routines return control in the same addressing mode in which they are invoked.
Alternative library interface routine GETMSG service routine TSO/E service facility routines TSO/E environment service routine	IKJADTAB GETMSG IKJEFTSI IKJEFTST IKJTSOEV	These service routines must be invoked in 31-bit addressing mode, and can accept input above or below 16 MB in virtual storage. These routines return control in 31-bit addressing mode.

Invoking the TSO/E service routines

You can use either the LINK or the LOAD macro instructions to pass control to the TSO/E service routines.

The LINK macro instruction loads the routine into storage based on the routine's RMODE attribute. The LINK macro instruction passes control to the routine in the addressing mode specified or allowed by its AMODE attribute.

The LOAD macro instruction loads the routine into storage based on the routine's RMODE attribute. Because the LOAD macro instruction loads a program but does not invoke it, you must do branches to the loaded routine. LOAD returns the address of the loaded program where the high-order bit of this address reflects the AMODE attribute of the loaded program. If the loaded program should not be invoked in the current addressing mode, you can use the BASSM, BSM, SAM24 or SAM31 instruction to set the appropriate addressing mode. If you use BASSM or BSM, ensure that the invoked program can return successfully.

Summary of macro interfaces

Table 3 on page 12 shows the MVS programming rules for using the macros described in this manual.

In Table 3 on page 12, a dash (-) indicates that the category does not apply to the macro because the macro does not generate executable code. The addressing mode of the program that accesses the data generated by the macro must agree with the residence of the data.

Table 3. MVS interface rules for using macro interfaces

Macro	(X) May be issued in		(P) May be issued by a program (I) Input may be	
	24-bit mode	31-bit mode	Below 16MB	Above 16MB
CALLTSSR	X	X	P	P
GETLINE	X	X	I,P	I,P
GTSIZE	X	X	P	P
GTTERM	X		P	
IKJENDP	-	-	P	P
IKJIDENT	-	-	P	P
IKJKEYWD	-	-	P	P
IKJNAME	-	-	P	P
IKJOPER	-	-	P	P
IKJPARM	-	-	P	P
IKJPOSIT	-	-	P	P
IKJRLSA	X	X	P	P
IKJRSVWD	-	-	P	P
IKJSUBF	-	-	P	P
IKJTERM	-	-	P	P
IKJUNFLD	-	-	P	P
IKJTSMSG	-	-	P	P
PUTGET	X	X	I,P	I,P
PUTLINE	X	X	I,P	I,P
RTAUTOPT	X	X	P	P
SPAUTOPT	X	X	P	P
STACK	X	X	I,P	I,P
STATTN	X		I,P	
STAUTOCP	X	X	P	P
STAUTOLN	X		I,P	
STAX	X	X	I,P	
STBREAK	X		I,P	
STCC	X		I,P	
STCLEAR	X		I,P	
STCOM	X		I,P	
STFSMODE	X		I,P	
STLINENO	X		I,P	

Table 3. MVS interface rules for using macro interfaces (continued)

Macro	(X) May be issued in		(P) May be issued by a program (I) Input may be	
	24-bit mode	31-bit mode	Below 16MB	Above 16MB
STSIZE	X		I,P	
STTIMEOU	X		I,P	
STTMPMD	X		I,P	
STTRAN	X		I,P	
TCLEARQ	X		I,P	
TGET	X	X	I,P	
TPG	X	X	I,P	
TPUT	X	X	I,P	

Notes:

CALLTSSR
The CALLTSSR macro instruction can be issued in either 24-bit or 31-bit addressing mode. See Chapter 4, "Invoking TSO/E service routines with CALLTSSR," on page 33 for more information on issuing the CALLTSSR macro.

GETLINE, PUTGET, PUTLINE, STACK
The GETLINE, PUTGET, PUTLINE, and STACK macros can be issued in either 24-bit or 31-bit addressing mode. These routines return control in the same addressing mode in which they are invoked. Input passed to these routines can reside above or below 16 MB in virtual storage. However, if you use the STACK macro, the list source descriptor (LSD) must reside below 16 MB.

IKJTSMSG
The IKJTSMSG macro can be issued by a program loaded below or above 16 MB in virtual storage. Refer to Chapter 11, "Using the TSO/E message handling routine IKJEFF02," on page 275 for a description of the standard and extended formats of the input parameter list for IKJEFF02.

If the Parse Service Routine is invoked in 31-bit addressing mode, the parse parameter list, mapped by IKJPPL, can reside above 16 MB in virtual storage and the parse macro instructions can be issued by a program loaded above 16 MB. See above for a list of the parse macros and their linkage requirements. The IKJRLSA parse macro can be issued in either 24- or 31-bit addressing mode.

STAX
A program can issue the STAX macro in either 24- or 31-bit addressing mode. Refer to Chapter 12, "Using the STAX service routine to handle attention interrupts," on page 285 for specific restrictions.

TGET, TPUT, TPG
The TGET, TPUT, and TPG macros can be issued in either 24- or 31-bit addressing mode. All input passed to them must reside below 16 MB in virtual storage.

Terminal Control Macros
With a few exceptions, terminal control macros must be issued in 24-bit addressing mode. The exceptions are the GTSIZE, RTAUTOPT, SPAUTOPT, and STAUTOCP terminal control macros, which can be issued in 31-bit addressing mode. See above for a list of the terminal control macros and their linkage requirements.

Interfacing with the TSO/E service routines

When you invoke the TSO/E service routines from a program running in a TSO/E environment, your program must pass to the service certain addresses contained in the Command Processor parameter list (CPPL).

The Command Processor parameter list

The command processor (CPPL) is a 4-word parameter list. When the TSO/E TMP attaches a command processor, it creates a CPPL in subpool 1 and passes the address of the CPPL to the command processor in register 1. The TSO/E TMP shares subpool 78 with the Command Processor, but it does not share subpool 0. In turn, the command processor or program must share subpool 78 with any lower-level tasks.

Notes:

1. Programs that use the TSO/E Environment Service to establish a TSO/E environment should share subpool 78 with any commands or programs that they invoke.

2. The TSO/E Environment Service returns the address of a Command Processor parameter list to programs that specify the CPPL address parameter.
3. A program running under the TSO/E TMP and Service Routines cannot invoke the TSO/E Environment Service.

The interface between the TMP and an attached Command Processor is shown in Figure 1 on page 14.

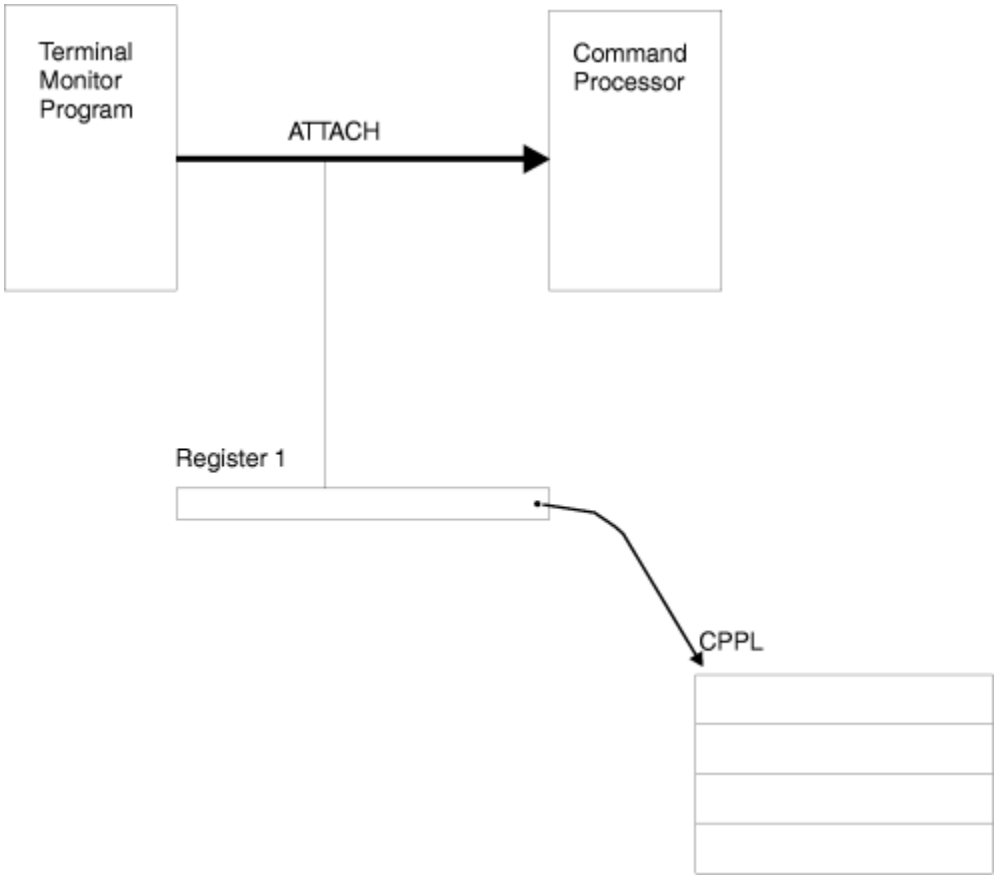


Figure 1. Interface between the TMP and a Command Processor

The interface between the TSO/E Environment Service and a calling program is shown in Figure 2 on page 15.

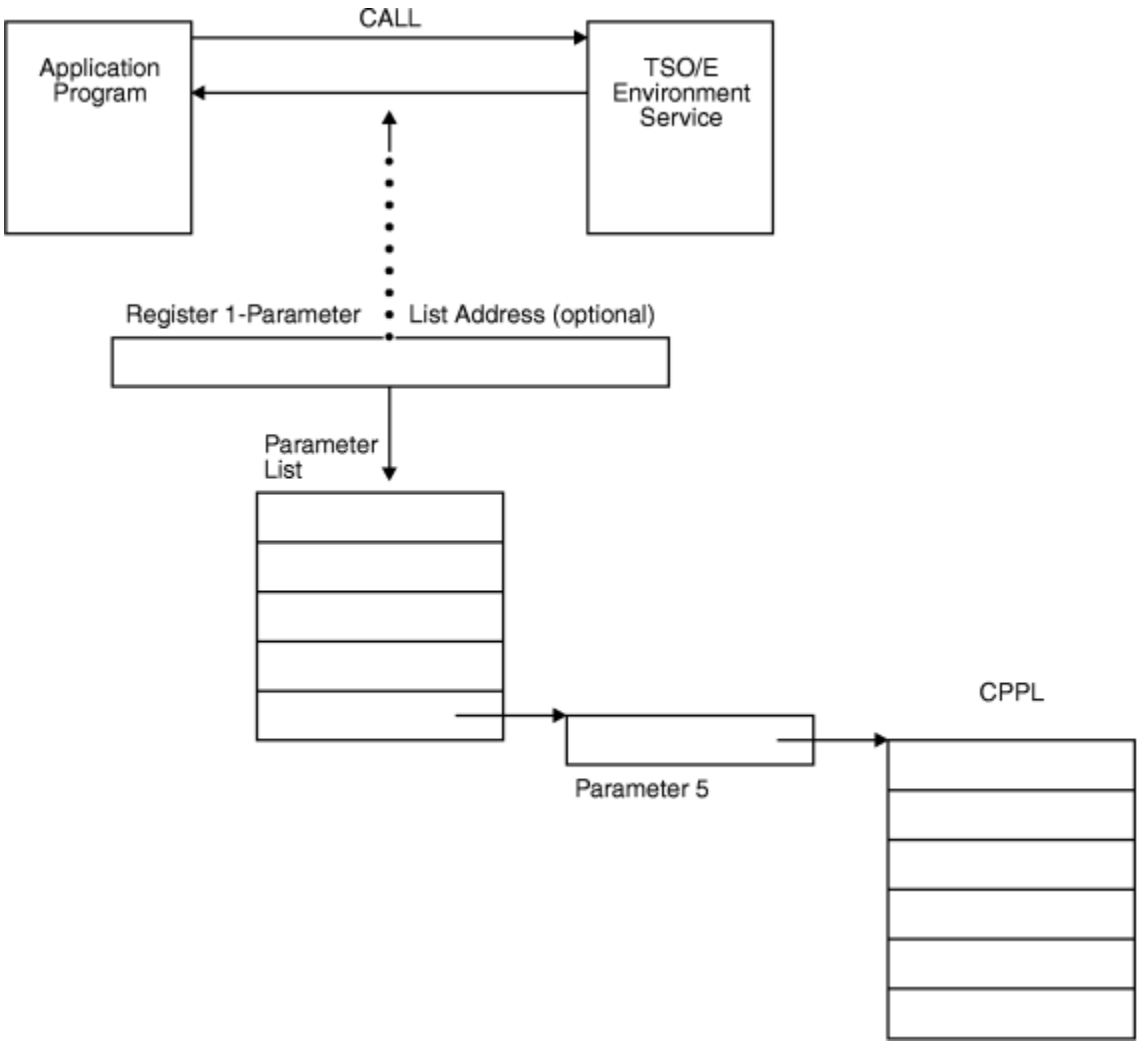


Figure 2. Control block interface between the TSO/E Environment Service and a calling program

You can use the IKJCPPL DSECT, which is provided in SYS1.MACLIB, to map the fields in the CPPL. Use the address contained in register 1 as the starting address for the DSECT, and then reference the symbolic field names within the IKJCPPL DSECT to access the fields in the CPPL. Use the DSECT because it protects the Command Processor from any changes to the CPPL. Table 4 on page 15 describes the contents of the CPPL.

Table 4. The Command Processor parameter list (CPPL)

Number of bytes	Field name	Contents or meaning
4	CPPLCBUF	The address of the command buffer for the currently attached Command Processor.
4	CPPLUPT	The address of the user profile table (UPT). Use the IKJUPT mapping macro, which is provided in SYS1.MACLIB, to map the fields in the UPT.
4	CPPLPSCB	The address of the protected step control block (PSCB). Use the IKJPSCB mapping macro, which is provided in SYS1.MACLIB, to map the fields in the PSCB.
4	CPPLECT	The address of the environment control table (ECT). Use the IKJECT mapping macro, which is provided in SYS1.MACLIB, to map the fields in the ECT.

Services that access data in the CPPL

When you invoke any of the following TSO/E service routines from your program, you must pass certain addresses contained in the CPPL as input:

IKJDAIR
Dynamic allocation interface routine

IKJEFF02
TSO/E message issuer routine

IKJEFF18
DAIRFAIL

IKJEFF19
GNRLFAIL/VSAMFAIL

IKJGETL
GETLINE service routine

IKJEHDEF
Default service routine

IKJPARS
Parse Service Routine

IKJPTGT
PUTGET service routine

IKJPUTL
PUTLINE service routine

IKJSCAN
Command Scan Service Routine

IKJSTCK
STACK service routine

Information concerning the input to the TSO/E service routines is discussed in more detail in the chapters of this manual describing the individual service routines.

Chapter 3. Using the TSO/E Environment Service IKJTSOEV

This chapter describes how and when to use the TSO/E Environment Service, the major benefits of using this service, its functions and limitations, and the preconditions necessary to use it.

Overview of the TSO/E Environment Service

Chapter 2, “Considerations for using TSO/E services,” on page 9 describes considerations for invoking TSO/E service routines from a Command Processor under the control of the TSO/E TMP and Service Routines. In some situations, you may wish to invoke TSO/E services outside of the TSO/E TMP and Service Routines. For example, in a VTAM® application, you might wish to execute TSO/E CLISTs or REXX execs without the overhead of establishing a TSO/E session for each invocation. The TSO/E Environment Service (IKJTSOEV) builds and initializes a TSO/E environment, which enables you to invoke common TSO/E programming services outside of the TSO/E TMP and Service Routines.

The TSO/E Environment Service offers a number of performance benefits. First of all, performance is improved because you do not execute the TSO/E TMP and Service Routines. Instead, your application directly invokes TSO/E services and facilities, allowing you to fine tune your application to meet the needs of your installation. Also, you can take advantage of the benefits of APPC/MVS. For example, you can establish a link from your personal computer or workstation to TSO/E through your MVS application. For more information on writing APPC/MVS application programs, see *z/OS MVS Programming: Writing Transaction Programs for APPC/MVS*.

You can call the TSO/E Environment Service directly from an application program. It then becomes an integral part of your application, allowing you to access TSO/E services without logging on TSO/E. You can also invoke the service from a high-level language program, aiding program development and maintenance.

When you should use the TSO/E Environment Service

The TSO/E Environment Service offers an efficient alternative to the TSO/E TMP and Service Routines environment for MVS applications that use TSO/E services. By taking advantage of its ease of use and performance benefits, you can modify or create application interfaces to TSO/E that range from the invocation of a single TSO/E command or CLIST to a more generic TSO/E Command Processor. You should use the TSO/E Environment Service in MVS applications that require basic TSO/E services without TSO/E TMP support.

- Developing TSO/E Applications Outside of the TMP:

  Use the TSO/E Environment Service in developing applications that run outside of the TSO/E TMP and Service Routines. For example, if you want your batch program to use TSO/E dynamic allocation services, you can submit the program as an MVS batch job outside of the TSO/E TMP and Service Routines and use the TSO/E Environment Service to access dynamic allocation routines. You can also use TSO/E services from interactive VTAM applications. Using TSO/E services (for example, parsing and syntax checking), you can design an efficient terminal monitor that is tailored to your specific application.
- Establishing a Common TSO/E Interface:

  Use the TSO/E Environment Service to provide a common interface across multiple applications. For example, after calling the TSO/E Environment Service, you can use the TSO/E Service Facility to invoke TSO/E commands, CLISTs, or REXX execs. As a result, you can develop functions that are shared across applications.
- Accessing TSO/E Services From Other Environments:

Use the TSO/E Environment Service as a bridge to TSO/E from other application platforms in your installation. For example, you can integrate TSO/E services into your personal computer or workstation applications using an APPC/MVS application as a front-end processor. The TSO/E Environment Service can be invoked from standard transaction programs scheduled by the APPC/MVS transaction scheduler. The TSO/E Environment Service can also be invoked from multi-trans (multiple transaction) transaction programs; however, a number of restrictions can apply. Refer to "Multi-trans transaction program limitations" on page 20 for an explanation of these restrictions.

The TSO/E Environment Service is not a replacement for the TSO/E TMP and Service Routines environment. In situations where your application itself must run under TSO/E, IKJTSOEV is not appropriate. For example, if your program uses ISPF Dialog Manager display services, you should continue to use a standard interactive TSO/E session. ISPF facilities such as these are also designed to make TSO/E application development simple and efficient.

Function of the TSO/E Environment Service

The TSO/E Environment Service builds and initializes a TSO/E environment outside of the TSO/E TMP and Service Routines. The TSO/E environment provides access to common TSO/E programming services. This section discusses the function of the TSO/E Environment Service and the particular services that are available.

The TSO/E Environment Service establishes a TSO/E environment in background mode, where input is from an alternate input source, such as a data set. Unlike the TSO/E TMP and Service Routines environment, which may attach a Command Processor, you invoke the TSO/E Environment Service from your program, which then acts like a Command Processor. Your program can then issue TSO/E services and macros that use the TSO/E environment.

TSO/E environment initialization - inside IKJTSOEV

The TSO/E environment is initialized to indicate that SYSTSIN is the current input source and SYSTSPRT is the current output source. The TSO/E Environment Service uses existing allocations for the SYSTSIN and SYSTSPRT files if you have allocated them. Otherwise, it allocates them as DUMMY data sets. You must allocate SYSTSPRT and SYSTSIN correctly and ensure that they are closed upon entry to IKJTSOEV. During initialization, the TSO/E Environment Service opens SYSTSIN and SYSTSPRT, but it does not read from the SYSTSIN file or process any command input.

Note:

1. The TSO/E Environment Service associates the TSO/E environment with the highest jobstep task in the calling program's address space.
2. The TSO/E Environment Service establishes a REXX Language Processor Environment that is associated with the task from which you invoked IKJTSOEV.
3. Any jobstep tasks that your application creates must use the same jobstep control block (JSCB). If your program attaches a new jobstep task with a different JSCB after the TSO/E environment is created, the new task cannot invoke TSO/E services.
4. You may specify a user ID through the USER parameter of the JCL that you use to start your application. In your application, the user ID is available in the PSCBUSER field in the PSCB.
5. The TCB key (TCBPKF) of the task under which the caller of the TSO/E Environment Service runs must match the TCB key of the highest non-system jobstep task in the address space. If the keys do not match, return code 40 (decimal) is returned, along with an indication of the mismatched keys in register 0.

Capabilities available after initialization

After successful execution of IKJTSOEV, SYSTSIN and SYSTSPRT are open for processing. At this point, the calling program performs like a Command Processor under the TSO/E TMP and Service Routines; parameter 5 contains the address of the Command Processor parameter list (CPPL). Your program can issue TSO/E service calls and macros that use the TSO/E environment.

For example, you can use the TSO/E Environment Service to process a TSO/E CLIST or REXX exec through an input file. Use the GETLINE macro (see Chapter 9, "Using the TSO/E I/O service routines for terminal I/O," on page 173) to read the command line from the current input source. Then you can parse the input using the PARSE command (see Chapter 6, "Verifying command and subcommand operands with parse," on page 43) and execute the parsed command through the TSO/E Service Facility (IKJEFTSR). TSO/E writes the results of the invocation to the current output file.

Some restrictions apply to the use of TSO/E services in the environment created by IKJTSOEV (see "Requirements and restrictions for invoking the TSO/E Environment Service" on page 23). For more information about TSO/E service routines, see Chapter 2, "Considerations for using TSO/E services," on page 9.

Job step termination

When the jobstep task with which the TSO/E environment is associated terminates, termination services releases the TSO/E environment automatically. This frees resources that the TSO/E Environment Service acquired during initialization, including TSO/E files and the REXX Language Processor Environment (which is freed automatically at the end of the task under which it was initialized).

Restrictions and limitations on the use of TSO/E services

Some restrictions apply to the use of services in the TSO/E environment that IKJTSOEV creates. These restrictions result from task structure and background mode limitations inherent in the environment that the TSO/E Environment Service establishes.

Task structure limitations

The TSO/E Environment Service creates a TSO/E environment with which an application can efficiently use some TSO/E services outside of the TSO/E TMP and Service Routines; it does *not* create the TSO/E task structure that is required by some commands and programs. The commands and programs that require the TSO/E task structure include foreground initiated background commands (commands that control batch job activity), those that are run through the TSO/E Service Facility in an authorized state, and those authorized during TSO/E system generation using the AUTHCMD, AUTHPGM, and AUTHTSF statements in SYS1.PARMLIB, member IKJTSOxx. For more information on foreground initiated background commands, see *z/OS TSO/E Command Reference*. For more information on authorizing programs using SYS1.PARMLIB, see *z/OS TSO/E Customization*.

Further, it should be noted that the TSO/E Environment Service treats some control blocks differently than the TSO/E TMP and Service Routines do. This leads to restrictions on the use of MVS services that depend on the contents of these control blocks. For example, the protected step control block PSCB is initialized differently by the TSO/E Environment Service, which restricts the dynamic allocation (SVC 99) of internal readers while the TSO/E Environment Service is active. Nevertheless it is possible to overcome some of these limitations, the user should keep in mind the intended use of the TSO/E Environment Service; see also "Summary of TSO/E services available under IKJTSOEV" on page 20.

TCB key limitations

The system establishes a TSO/E environment in the TCB key of the caller of IKJTSOEV. Programs that use TSO/E services in that environment must be in the same TCB key as the caller of IKJTSOEV.

Background mode limitations

IKJTSOEV sets up the TSO/E environment in background mode; command invocation is identical to background processing under the TSO/E batch facility. Therefore, batch facility conventions and restrictions for prompting and command usage apply. The TSO/E environment established by IKJTSOEV will only use default authority attributes for the protected step control block (PSCB) and the user profile table (UPT). For more information on TSO/E background processing, see *z/OS TSO/E Command Reference*, particularly, the PROFILE command.

Background mode does not support TSO/E services that perform full- screen terminal I/O using the TPUT, TGET, and TPG macros. It does support supervisor and inter-user communication among terminals using TPUT with the ASID, ASIDLOC, or USERIDL parameters. The ISPF Dialog Manager display facilities cannot be used, because they perform full- screen terminal I/O at the user's terminal.

Multi-trans transaction program limitations

The TSO/E Environment Service can be invoked from standard transaction programs (TPs) scheduled by the APPC/MVS transaction scheduler. It can also be invoked from multi-trans TPs; however, there are some cases in which certain restrictions apply when the TSO/E Environment Service is invoked from multi-trans TPs. When a multi-trans TP invokes the TSO/E Environment Service and processes inbound work requests on behalf of only the generic user ID, no restrictions apply. When a multi-trans TP processes multiple userids under a single TSO/E environment, that is, a multi-trans TP invokes the TSO/E Environment Service from the multi-trans TP shell and processes inbound work requests on behalf of multiple user IDs, the following restrictions apply:

- TSO/E builds the TSO/E environment personalized to the generic user ID.
- TSO/E does not personalize the TSO/E environment for the other user IDs.

These restrictions have an affect on any program(s) that use fields associated with user ID information in the following TSO/E control blocks: PSCB, UPT, ECT, and ENVBLOCK. Programs, which use fields in those TSO/E control blocks associated with the generic user ID and are invoked by multi-trans TPs processing inbound work requests on behalf of multiple user IDs, might be subject to the above restriction, and the functions of these programs might be affected. Some of the TSO/E information and functions affected by these restrictions are described below:

- The PROFILE command settings associated with the generic user ID are initially used for all user IDs. Subsequent invocations of the PROFILE command reset the settings associated with the generic user ID and are used until they are reset. Refer to *z/OS TSO/E Command Reference* for the details about the PROFILE command settings.
- Attributes associated with the generic user ID's user definition are used for all user IDs. Some of these attributes are described below:
  - The default job class, output class, and held class for the SUBMITted jobs defined for the generic user ID are used for all user IDs.
  - The generic user ID's authorization to use the OPERATOR, ACCOUNT and CONSOLE commands will be used for all user IDs. Refer to *z/OS TSO/E Administration* for further information about "user definitions" and the details of these TSO/E functions.
  - The REXX environment settings associated with the generic user ID, for example, the language set by SETLANG, are used by all user IDs. The SYSUID variable is set to the generic user ID and the built-in function, USERID, returns the generic user ID. Refer to *z/OS TSO/E REXX Reference* for the details of these environment settings.

Summary of TSO/E services available under IKJTSOEV

Table 5 on page 21 summarizes the availability of functions under the TSO/E Environment Service.

Table 5. Summary of TSO/E service availability under IKJTSOEV

Type of service/facility	Name of service/facility	Supported
Command Invocation	**IKJSCAN** - Command Scan Service **IKJPARS** - Parse Service **IKJTBLS** - Table Look-up Service **IKJEFTSR** - TSO/E Service Facility **(Non-authorized invocations only)**	Yes Yes Yes Yes
Data Set and File I/O	**IKJADTAB** - Alternative Library Interface **DAIR** - Dynamic Allocation Interface **DAIRFAIL** - Dynamic Allocation Diagnostics **GNRLFAIL/VSAMFAIL** - VSAM Diagnostics **STACK Macro** - I/O stack handling PUTLINE, GETLINE, and PUTGET macros	Yes Yes Yes Yes Yes Yes
Foreground Initiated Background Commands	SUBMIT, OUTPUT, CANCEL, STATUS, CONSOLE, and ALLOCATE ALTFILE	No
TSO/E Pgm. Debugging	TSO/E TEST Command	No
Session Manager Facilities	SMCOPY All Other Session Manager Facilities	Yes No
System Information	**IKJEHCIR** - Catalog Information Routine **ICQGCL00** - Data Set List Routine **IKJEHDEF** - Default Service Routine **ICQAML10** - Names Facility **IKJEFF02** - Message Handling Routine **IKJCT441** - Variable Access Routine	Yes Yes Yes Yes Yes Yes
Terminal Attention	STAX Service Routine CLIST Attention Facility	Yes Yes

Table 5. Summary of TSO/E service availability under IKJTSOEV (continued)		
Type of service/facility	**Name of service/facility**	**Supported**
Terminal I/O	**QSAM and BSAM Macros** **TPUT, TGET, TPG** - Full-screen I/O **TPUT** - Supervisor and inter-user communication	Yes No Yes

Syntax and parameter descriptions

CALL IKJTSOEV (*parm1, parm2, parm3, parm4, parm5*)

Figure 3. Call syntax for the IKJTSOEV routine

IKJTSOEV supports five optional parameters. The parameters are positional and follow standard parameter passing conventions (see *z/Architecture® Principles of Operation*). In assembler language, the high-order bit of the last specified parameter must be set to 1 to indicate the end of the parameter list. If no parameters are specified, register 1 should be set to 0.

parm1
A fullword which is reserved for future use.

parm2
A fullword integer. Upon return from IKJTSOEV, this parameter contains a return code indicating the completion status of the call. For more information on this parameter, see Table 6 on page 23.

parm3
A fullword integer. Upon return from IKJTSOEV, this parameter contains a reason code that provides specific information about an unsuccessful completion. For more information on this parameter, see Table 7 on page 25 and Table 8 on page 25.

parm4
A fullword integer. Upon return from IKJTSOEV, this parameter contains a code that further describes an error indicated in parameter 3. For more information on this parameter, see Table 8 on page 25.

parm5
A fullword address. Upon return from IKJTSOEV, this parameter contains the address of the Command Processor parameter list (CPPL). For more information on the Command Processor parameter list, see “Interfacing with the TSO/E service routines” on page 13.

Invoking the TSO/E Environment Service

This section describes how to invoke the TSO/E Environment Service from an application program. Because IKJTSOEV is a callable routine, any high- or low-level language application that runs under MVS can call it. Figure 3 on page 22 illustrates the call syntax for the IKJTSOEV routine.

To invoke the TSO/E Environment Service, call IKJTSOEV from your program. For high-level languages, you can use the alias TSOENV to limit the length of the program name to 6 characters. See “Examples using the TSO/E Environment Service” on page 26 for sample COBOL and assembler programs.

To create a callable module, include IKJTSOEV in the link-edit of your calling routine. The IKJTSOEV module contains the entry point IKJTSOEV.

Requirements and restrictions for invoking the TSO/E Environment Service

In addition to the restrictions on the use of TSO/E services discussed in "Restrictions and limitations on the use of TSO/E services" on page 19 there are additional guidelines, which you must follow in developing applications that call IKJTSOEV. These guidelines are listed below.

Addressability requirements

- The application cannot be executing in a cross-memory mode.
- The application can be in primary mode or access register mode. In access register mode, the parameter addresses for IKJTSOEV must reference memory locations in the primary address space. Setting the access list entry token (ALET) to 0 ensures that address translation uses the primary segment table to resolve the addresses within the primary address space.
- The application must invoke IKJTSOEV in 31-bit addressing mode.

  Note: You must invoke TSO/E and MVS services in the appropriate addressing mode. You can use the entry specifications for the service you are calling to determine the required addressing mode.

  See *z/Architecture Principles of Operation* for more information on addressing modes.

Resource allocation requirements

- The application cannot be holding any MVS system locks higher than the LOCAL lock when it invokes IKJTSOEV. When the TSO/E Environment Service detects that the calling program is holding a lock, it ignores the request for initialization and returns to the calling program with return code 32. See *z/OS MVS Diagnosis: Reference* for information about MVS/ESA system locks.
- The user's task must share subpool 78 with its jobstep task as well as any lower-level subtasks.
- An application should not attempt to free any TSO/E control blocks or TSO/E files. The job scheduler deallocates the TSO/E environment and its acquired resources when it deallocates the address space that the TSO/E environment resides in.

REXX ADDRESS TSO support requirements

- If you want REXX ADDRESS TSO support, you must ensure that no REXX Language Processor Environment exists in your address space when you invoke IKJTSOEV. If you invoke IKJTSOEV from an address space that already contains a REXX Language Processor Environment and the REXX environment does not include the ADDRESS TSO host command environment, the REXX Language Processor Environment will continue to be available without ADDRESS TSO support.

Return and reason codes from the TSO/E Environment Service

IKJTSOEV uses the return code parameter to provide general information about the completion status of TSO/E environment initialization. IKJTSOEV uses the reason code parameter to indicate a specific condition that caused the return code condition. Table 6 on page 23 lists return codes from the initialization routines in IKJTSOEV. Table 7 on page 25 and Table 8 on page 25 contain reason codes for the unsuccessful initialization of the REXX and TSO/E environments, respectively. The following table describes the return codes that are issued by the IKJTSOEV initialization routines:

Table 6. Return codes for TSO/E environment initialization

Return code	Description
0	TSO/E environment initialization successful. Parameter 5 contains the address of the CPPL.

Table 6. Return codes for TSO/E environment initialization (continued)

Return code	Description
8	TSO/E environment initialized, but could not initialize a REXX Language Processor Environment. Parameter 5 contains the address of the CPPL. Parameter 3 contains the IKJTSOEV reason code for the REXX initialization failure. You can still use all of the TSO/E services listed in Table 5 on page 21. REXX services may be limited or unavailable, depending on whether a REXX Language Processor Environment was present in the calling program's address space before the invocation of the TSO/E Environment Service. For more information on REXX service availability for this return code, see Table 7 on page 25.
16	The request for initialization was ignored because a TSO/E environment is being initialized or has been initialized already at the request of another task in the same address space as the calling program.
20	The request for initialization was ignored because the address space of the calling program contains multiple job step control blocks (JSCB's). An application cannot use the TSO/E Environment Service if it attached multiple job step tasks in its address space.
24	The request for initialization was ignored because the TSO/E Environment Service was invoked from a TSO/E TMP and Service Routines environment.
32	The request for initialization was ignored because the caller is in cross-memory mode or holding a lock. See “Requirements and restrictions for invoking the TSO/E Environment Service” on page 23 for addressability and resource allocation requirements for the TSO/E Environment Service.
36	TSO/E environment initialization unsuccessful. The reason code (parameter 3) indicates the specific cause of the failure. See Table 8 on page 25 for more information.
40	TSO/E environment initialization unsuccessful. The TCB key of the caller's TCB does not match the TCB key of the first non-system jobstep TCB in the address space. Parameter 3 contains a reason code that is formed as follows: • Byte 0 is X'00' • Byte 1 contains the TCB key (TCBPFK) of the caller's TCB • Byte 2 is X'00' • Byte 3 contains the TCB key (TCBPFK) of the first non-system jobstep TCB in the address space

Table 6. Return codes for TSO/E environment initialization (continued)

Return code	Description
44	Invalid parameters were passed to IKJTSOEV. Either the parameter list is bad, or at least one of the parameters pointed to by an address in the parameter list is bad. IKJTSOEV might be called with 0, or 1 to some maximum number of parameters. The parameter list is invalid if it contains more than the maximum allowed number of parameters. The high-order bit of the last address in the parameter list must be turned on to indicate the end of the parameter list. If more than the maximum number of parameters are passed, the parameter list is bad. (Currently the maximum number is 5.) Also, if any parameter pointed to by one of the addresses in the parameter list is not read/write accessible in the key of the caller, it is considered invalid. If a bad parameter is found, IKJTSOEV returns normally to its caller with RC=44 (Bad_Parms) in R15, and with a reason code 600 (Abend_Happened) in R0 to indicate that an abend occurred during parameter validation. No other information is returned in any parameters, since some parameters are not valid.

If the return code is 8, parameter 3 contains the reason code for a failure to initialize a REXX Language Processor Environment. The reason codes are as follows:

Table 7. Reason codes for REXX initialization failure

Reason code	Description
40	A previous REXX Language Processor Environment exists in the calling program's address space. A Language Processor Environment cannot be initialized.
60	IKJTSOEV failed to load the REXX initialization routine (IRXINIT). Make sure that REXX is properly installed and your JCL specifies enough region space to load and execute the IRXINIT module.
80	IRXINIT failed. Parameter 4 contains the reason code from IRXINIT. For more information on reason codes from IRXINIT, see *z/OS TSO/E REXX Reference*.

If the return code is 36, parameter 3 contains the reason code for an unsuccessful TSO/E environment initialization. Parameter 4 contains an MVS service routine code or abend code corresponding to an error condition.

Table 8. Reason codes for TSO/E environment initialization failure

Reason code	Description
100	Request for virtual storage failed. Parameter 4 contains the return code from GETMAIN. For information about GETMAIN return codes, see *z/OS MVS Programming: Authorized Assembler Services Reference EDT-IXG*.
200	Global serialization of the PARMLIB resource failed. Parameter 4 contains the return code from ENQ. For information about ENQ return codes, see *z/OS MVS Programming: Authorized Assembler Services Reference EDT-IXG*.
300	Dynamic allocation for SYSTSIN failed. Parameter 4 contains the error reason code (S99ERROR) from SVC 99. For information about SVC 99 error reason codes, see *z/OS MVS Programming: Authorized Assembler Services Guide*.

Table 8. Reason codes for TSO/E environment initialization failure (continued)

Reason code	Description
400	Dynamic allocation for SYSTSPRT failed. Parameter 4 contains the error reason code (S99ERROR) from SVC 99. For information about SVC 99 error reason codes, see *z/OS MVS Programming: Authorized Assembler Services Guide*.
500	TSO/E I/O stack creation failed. Parameter 4 contains the return code from the STACK service routine. For information about STACK return codes, see Chapter 12, "Using the STAX service routine to handle attention interrupts," on page 285.
600	An abend occurred in a macro or service called by the TSO/E Environment Service. The TSO/E Environment Service returned control to the calling program with the system completion (abend) code in parameter 4. For information about MVS abend codes, see *z/OS MVS System Codes*.

Examples using the TSO/E Environment Service

The following examples illustrate how to create an application that uses IKJTSOEV. Figure 4 on page 27 and Figure 7 on page 30 are COBOL and assembler coding examples. Figure 8 on page 31 and Figure 9 on page 32 show JCL to execute the COBOL and assembler programs, respectively.

COBOL

In Figure 4 on page 27, a COBOL program calls IKJTSOEV to establish a TSO/E environment. The COBOL program then verifies that the environment has been initialized successfully by checking the return code from the TSO/E Environment Service. If an error occurs, program DISPLAY statements write error messages to the SYSOUT file, and the program ends. After IKJTSOEV successfully creates a TSO/E environment, the program invokes a TSO/E REXX exec named 'TEST1' (Figure 5 on page 29) using the TSO/E Service Facility. TSO/E writes the output from the REXX exec to the SYSTSPRT file. Finally, the program verifies successful invocation of the REXX exec by checking the return code from the call to the TSO/E Service Facility.

Note: In the sample COBOL routine, notice that:

- The first 6 characters of each line are reserved for numbers or labels, or should be blank. In the sample, the first 6 characters of each line are blanks.
- The comment lines (for example, the lines beginning with "*") must start in column 7. That is, the * character must be in column 7, and the command text should flow so that the comment on any given line does not go past column 72.
- These items must begin somewhere in columns 8-11:
  - Division headers (IDENTIICATION, ENVIRONMENT, DATA, and PROCEDURE divisions)
  - Section headers
  - Paragraph headers or paragraph names
  - Level indicators or level-numbers (01 and 77)
  - DECLARATIVES and END DECLARATIVES
  - End program, end class, and end method marker
- These items must begin in or after column 12:
  - Entries, sentences, statements, and clauses
  - Continuation line

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID. ENVCOBRX.

*  CHANGES - COMMENTS MOVED TO START IN COL7
*          - PARAGRAPH MUST START IN COL8
*          - STATEMENTS IN PARAGRAPH MUST START COL 11 OR LATER
*          - USE DOUBLE QUOTES, NOT SINGLE AROUND CALL TARGETS
*          - MOVE EXEC 'TEST' TO TSF-PARM2 .. SHOULD BE
*            MOVE "EXEC 'TEST'" TSF-PARM2


************************************************************
* THIS IS A SAMPLE COBOL PROGRAM TO DEMONSTRATE THE USE OF
* THE TSO/E ENVIRONMENT SERVICE.  FIRST, THE PROGRAM CALLS
* IKJTSOEV TO ESTABLISH A TSO/E ENVIRONMENT.  NEXT, THE
* PROGRAM CALLS THE TSO SERVICE FACILITY (IKJEFTSR) TO
* INVOKE A REXX EXEC CALLED 'TEST1'.  AFTER THE REXX EXEC IS
* INVOKED, THE PROGRAM DISPLAYS THE RETURN, REASON, AND
* ABEND CODES FROM THE CALL TO THE TSO SERVICE FACILITY.
************************************************************

 EJECT
 ENVIRONMENT DIVISION.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
 I-O-CONTROL.
 DATA DIVISION.
 FILE SECTION.
 WORKING-STORAGE SECTION.

   01 TSOEV-PARM1          PIC S9(9) VALUE +0   COMP-4.
   01 TSOEV-RETURN-CODE    PIC S9(9) VALUE +0   COMP-4.
   01 TSOEV-REASON-CODE    PIC S9(9) VALUE +0   COMP-4.
   01 TSOEV-ABEND-CODE     PIC S9(9) VALUE +0   COMP-4.
   01 TSOEV-CPPL-ADDR      PIC S9(9) VALUE +0   COMP-4.

   01 TSF-PARM1            PIC S9(9)            COMP-4.
   01 TSF-PARM2            PIC X(80).
   01 TSF-PARM3            PIC S9(9) VALUE +80  COMP-4.
   01 TSF-PARM4            PIC S9(9) VALUE +0   COMP-4.
   01 TSF-PARM5            PIC S9(9) VALUE +0   COMP-4.
   01 TSF-PARM6            PIC S9(9) VALUE +0   COMP-4.

  *01 UNAUTH               PIC S9(9) VALUE +0   COMP-4.
  * TO REQ TSR BE INVOKED FROM AN UNAUTH (UNISOLATED) ENVIRONMENT
   01 UNAUTH               PIC S9(9) VALUE +65536  COMP-4.
   01 REQUEST-DUMP         PIC S9(9) VALUE +256 COMP-4.
   01 INVOKE-REXX          PIC S9(9) VALUE +1   COMP-4.



 EJECT
 PROCEDURE DIVISION.
 MAIN-PROGRAM.
************************************************************
* MAIN PROGRAM - INVOKE THE TSO/E ENVIRONMENT SERVICE TO
* INITIALIZE A TSO/E ENVIRONMENT.
*
* TSOEV-RETURN-CODE IS A FULLWORD THAT WILL CONTAIN THE
*          RETURN CODE FROM THE TSO/E ENVIRONMENT SERVICE.
*
* TSOEV-REASON-CODE IS A FULLWORD THAT WILL CONTAIN THE
*          REASON CODE FROM THE TSO/E ENVIRONMENT SERVICE.
*
* TSOEV-CPPL-ADDR IS A FULLWORD THAT WILL CONTAIN THE
* ADDRESS OF THE CPPL
*          ON RETURN FROM THE TSO/E ENVIRONMENT SERVICE.
************************************************************
```

Figure 4. Sample COBOL routine

Figure of 'Sample COBOL routine' **(Continued)**

```
           CALL "IKJTSOEV" USING TSOEV-PARM1
                                 TSOEV-RETURN-CODE
                                 TSOEV-REASON-CODE
                                 TSOEV-ABEND-CODE
                                 TSOEV-CPPL-ADDR.
************************************************************
      * NOW THAT WE'RE BACK FROM THE TSO/E ENVIRONMENT SERVICE,
      * CHECK THE RETURN CODE.
      *
      * IF THE RETURN CODE WAS ZERO, ISSUE IKJEFTSR TO INVOKE A
      * REXX EXEC.  IF THE RETURN CODE WAS NON-ZERO, DISPLAY AN
      * ERROR MESSAGE.
      ************************************************************

           IF RETURN-CODE = 0 THEN
              PERFORM EXEC-REXX THROUGH EXEC-REXX-EXIT
           ELSE
              PERFORM DISPLAY-MESSAGE THROUGH DISPLAY-MESSAGE-EXIT.

           STOP RUN.
       EXEC-REXX.
      ************************************************************
      * EXEC-REXX.  - EXECUTE THE REXX EXEC 'TEST1' USING THE TSO
      * FSERVICE ACILITY
      *
      * PARM1 WILL INDICATE THAT A TSO/E COMMAND, CLIST, OR REXX
      *       EXEC IS BEING INVOKED AND A DUMP SHOULD BE PRODUCED
      *       IF AN ABEND OCCURS.
      *
      * PARM2 WILL CONTAIN THE NAME OF THE REXX EXEC - 'TEST1'
      *
      * PARM3 WILL CONTAIN THE RETURN CODE FROM THE INVOCATION OF
      *       THE REXX EXEC 'TEST1'.
      *
      * PARM4 WILL CONTAIN THE RETURN CODE FROM THE TSO SERVICE
      * FACILITY
      *
      * PARM5 WILL CONTAIN THE REASON CODE FROM THE TSO SERVICE
      * FACILITY
      *
      * PARM6 WILL CONTAIN THE ABEND CODE FROM THE TSO SERVICE
      * FACILITY
      *
      ************************************************************
      * INITIALIZE PARM1
           MOVE 0 TO TSF-PARM1.
           ADD UNAUTH TO TSF-PARM1.
           ADD REQUEST-DUMP TO TSF-PARM1.
           ADD INVOKE-REXX TO TSF-PARM1.

      * INITIALIZE PARM2
           MOVE SPACES TO TSF-PARM2.
           MOVE "EXEC 'TEST1'" TO TSF-PARM2.
```

Figure of 'Sample COBOL routine' **(Continued)**

```
      * INVOKE THE TSO SERVICE FACILITY
            CALL "TSOLNK" USING TSF-PARM1
                               TSF-PARM2
                               TSF-PARM3
                               TSF-PARM4
                               TSF-PARM5
                               TSF-PARM6.

            DISPLAY "IKJEFTSR RETURNED THE FOLLOWING:"
            DISPLAY "  FUNCTION RETURN CODE - " TSF-PARM4.
            DISPLAY "  TSF REASON CODE ..OR.. ".
            DISPLAY "  ...ABEND REASON CODE - " TSF-PARM5.
            DISPLAY "  ABEND CODE           - " TSF-PARM6.
       EXEC-REXX-EXIT. EXIT.


      ************************************************************
      * DISPLAY MESSAGE - DISPLAY THE RETURN, REASON, AND ABEND
      *                   CODES FROM IKJTSOEV.
      ************************************************************
       DISPLAY-MESSAGE.

            DISPLAY "IKJTSOEV RETURNED THE FOLLOWING:"
            DISPLAY "  RETURN CODE          - " TSOEV-RETURN-CODE.
            DISPLAY "  REASON CODE          - " TSOEV-REASON-CODE.
            DISPLAY "  ABEND CODE           - " TSOEV-ABEND-CODE.

       DISPLAY-MESSAGE-EXIT. EXIT.
```

```
/* REXX */
SAY HI, IN TEST1
```

Figure 5. REXX exec 'TEST1' executed by COBOL

```
HI, IN TEST1
```

Figure 6. Output from the invocation of 'TEST1'

Assembler

Figure 7 on page 30 shows a sample assembler program that processes a TSO/E command. The program uses the TSO/E Environment Service to establish a TSO/E environment. The program then verifies that the TSO/E environment has been initialized successfully by checking the return code from the TSO/E Environment Service. If the TSO/E Environment Service fails to initialize a TSO/E environment, the program generates an abend and terminates. After IKJTSOEV successfully establishes a TSO/E environment, the program invokes the TSO/E ALTLIB command using the TSO/E Service Facility, and TSO/E writes the output from the ALTLIB command to the SYSTSPRT file.

You can modify this program and use it as a subroutine to process TSO/E commands that you specify.

```
*********************************************************************
* THIS IS A SAMPLE ASSEMBLER PROGRAM TO DEMONSTRATE THE USE OF THE TSO/E
* ENVIRONMENT SERVICE.  IT DOES THE FOLLOWING:
*
* 1. CALLS IKJTSOEV TO ESTABLISH A TSO/E ENVIRONMENT.
* 2. CALLS THE TSO SERVICE FACILITY TO INVOKE THE TSO ALTLIB COMMAND.
*********************************************************************
ENVTSCMD  CSECT
ENVTSCMD  AMODE  31
ENVTSCMD  RMODE  ANY
          STM    R14,R12,12(R13)
          BALR   R12,0
          USING  *,R12
          ST     R13,SAVEAREA+4
          LA     R11,SAVEAREA
          ST     R11,8(,R13)
          LA     R13,SAVEAREA
*********************************************************************
* CALTSOEV - CALL THE TSO/E ENVIRONMENT SERVICE TO ESTABLISH A TSO/E
*            ENVIRONMENT IN THIS PROGRAM'S ADDRESS SPACE.
*   PARM1 IS RESERVED
*   PARM2 IS A FULLWORD THAT WILL CONTAIN THE RETURN CODE FROM IKJTSOEV.
*   PARM3 IS A FULLWORD THAT WILL CONTAIN THE REASON CODE ON RETURN FROM
*         IKJTSOEV.
*   PARM4 IS A FULLWORD THAT WILL CONTAIN THE ABEND CODE, IF AN ABEND
*         OCCURS DURING TSO/E ENVIRONMENT SERVICE PROCESSING.
*   PARM5 IS A FULLWORD THAT WILL CONTAIN THE ADDRESS OF THE CPPL.
*********************************************************************
CALTSOEV  DS     0H
          L      R15,=V(IKJTSOEV)
          CALL   (15),(PARM1,PARM2,PARM3,PARM4,PARM5),VL
*********************************************************************
* CHKEVRC - CHECK THE RETURN CODE FROM IKJTSOEV
*********************************************************************
CHKEVRC   DS     0H
          L      R2,PARM2
          LTR    R2,R2
          BNZ    BADEVRC
*********************************************************************
* CALLTSR - CALL IKJEFTSR TO INVOKE THE TSO/E COMMAND 'ALTLIB DISPLAY'.
*           THE OUTPUT FROM THIS COMMAND WILL GO TO THE PREVIOUSLY
*           STACKED DATA SET.
*********************************************************************
CALLTSR   DS     0H
          L      R15,CVTPTR
          L      R15,CVTTVT(,R15)
          L      R15,TSVTASF-TSVT(,R15)
          CALL   (15),(FLAGS,CMDBUF,BUFLEN,RETCODE,RSNCODE,ABNDCODE),VL
*********************************************************************
* DOALL - AT THIS POINT, YOU CAN PROCESS THE RETURN VALUES FROM
*         IKJEFTSR AND THE INVOKED FUNCTION, ALTLIB.
*********************************************************************
DOALL     DS     0H
          B      EXIT
*********************************************************************
* BADEVRC - BRANCH HERE IF IKJTSOEV RETURNED A NON-ZERO RETURN CODE.
*           IF THE PROGRAM BRANCHES HERE, IT WILL ABEND WITH A DUMP.
*           IN THE DUMP, THE CONTENTS OF THE REGISTERS WILL BE AS FOLLOWS:
*              REGISTER 2 - THE RETURN CODE FROM IKJTSOEV
*              REGISTER 3 - THE REASON CODE FROM IKJTSOEV
*              REGISTER 4 - THE ABEND  CODE FROM IKJTSOEV
*********************************************************************
BADEVRC   DS     0H
          L      R2,PARM2
          L      R3,PARM3
          L      R4,PARM4
          ABEND  100,DUMP
```

Figure 7. Sample assembler routine

Figure of 'Sample assembler routine' **(Continued)**

```
*********************************************************************
* EXIT - RETURN TO CALLING PROGRAM
*********************************************************************
EXIT     DS    0H
         L     R13,4(,R13)
         LM    R14,R12,12(R13)
         SLR   R15,R15
         BR    R14
* REGISTER EQUATES
R2       EQU   2
R3       EQU   3
R4       EQU   4
R5       EQU   5
R11      EQU   11
R12      EQU   12
R13      EQU   13
R14      EQU   14
R15      EQU   15
* PARAMETERS USED TO INVOKE THE TSO/E ENVIRONMENT SERVICE
PARM1    DS    F                    RESERVED FIELD
PARM2    DS    F                    RETURN CODE FIELD
PARM3    DS    F                    REASON CODE FIELD
PARM4    DS    F                    FUNCTION ABEND CODE
PARM5    DS    F                    CPPL ADDRESS
* PARAMETERS USED TO INVOKE THE TSO SERVICE FACILITY
FLAGS    DS    0F                  FULLWORD OF FLAGS
RESFLAGS DC    H'0001'               ESTABLISH UNAUTHORIZED ENVIRONMENT
ABFLAGS  DC    X'01'                 PRODUCE A DUMP IF FUNCTION ABENDS
FNCFLAGS DC    X'01'                 INVOKE A TSO/E CMD, REXX EXEC, OR CLIST
CMDBUF   DC    C'ALTLIB DISPLAY'   COMMAND BUFFER
BUFLEN   DC    A(L'CMDBUF)         LENGTH OF COMMAND BUFFER
RETCODE  DS    F                   FUNCTION RETURN CODE
RSNCODE  DS    F                   FUNCTION REASON CODE
ABNDCODE DS    F                   FUNCTION ABEND CODE
CVTPTR   EQU   16                  THESE TWO PARMS ARE USED TO DETERMINE
CVTTVT   EQU   X'9C'               THE ADDRESS OF THE TSO SERVICE FACILITY
* SAVEAREA AND OTHER PROGRAM STORAGE
SAVEAREA DS    18F
* TSVT MAPPING MACRO (USED TO OBTAIN THE ADDRESS OF THE TSO SERVICE FACILITY)
         IKJTSVT
END
```

JCL for COBOL and assembler program invocation

Figure 8 on page 31 shows sample JCL to run the COBOL program. The program ENVCOBRX resides in IBMUSER.LOAD. Because neither the program nor the REXX exec that the program executes requires input, the JCL allocates SYSTSIN to a dummy data set. TSO/E uses the SYSTSPRT file to output messages issued by the REXX exec. Program DISPLAY statements send program error messages to the SYSOUT file.

```
//IBMUSERA JOB 'IKJTSOEV SAMPLE1',MSGLEVEL=(1,1),TIME=2,
//           CLASS=A,MSGCLASS=H
//*
//DOTSO    EXEC PGM=ENVCOBRX
//STEPLIB  DD   DSN=IBMUSER.LOAD,DISP=SHR
//SYSPROC  DD   DSN=IBMUSER.TSOENV.CLIST,DISP=SHR
//SYSTSPRT DD   SYSOUT=*
//SYSTSIN  DD   DUMMY
//SYSOUT   DD   SYSOUT=*
```

Figure 8. Execution JCL for the COBOL program

Figure 9 on page 32 shows sample JCL to run the assembler program. The program ENVTSCMD resides in IBMUSER.LOAD. Because the program does not use SYSTSIN for input, the JCL allocates SYSTSIN to a dummy data set. The program redirects output for the TSO/E command invocation to MYPRTDD, which is allocated to a data set. The program sends all other TSO/E output is sent to the SYSTSPRT file.

```
//IBMUSERA JOB 'IKJTSOEV SAMPLE1',MSGLEVEL=(1,1),TIME=2,
//           CLASS=A,MSGCLASS=H
//*
//DOTSO    EXEC PGM=ENVTSCMD
//STEPLIB  DD   DSN=IBMUSER.LOAD,DISP=SHR
//SYSPROC  DD   DSN=IBMUSER.TSOENV.CLIST,DISP=SHR
//SYSTSPRT DD   SYSOUT=*
//SYSTSIN  DD   DUMMY
//MYPRTDD  DD   SYSOUT=*
//SYSOUT   DD   SYSOUT=*
```

Figure 9. Execution JCL for the assembler program

Chapter 4. Invoking TSO/E service routines with CALLTSSR

This chapter describes how to use the CALLTSSR macro instruction to pass control to certain TSO/E service routines.

When to use the CALLTSSR macro instruction

You can use the CALLTSSR macro instruction to generate a branch to certain TSO/E service routines. The CALLTSSR macro instruction can be issued in either 24- or 31-bit addressing mode.

The CALLTSSR macro instruction can be used to invoke the following TSO/E service routines only:

IKJADTAB
Alternate library interface routine

IKJCAF
CLIST attention facility

IKJDAIR
Dynamic allocation interface routine

IKJEFF02
TSO/E message issuer routine

IKJEFTSI
TSO/E Service Facility initialization routine

IKJEFTST
TSO/E Service Facility termination routine

IKJEHCIR
Catalog information routine

IKJEHDEF
Default routine

IKJGETL
GETLINE service routine

IKJPARS
Parse Service Routine

IKJPTGT
PUTGET service routine

IKJPUTL
PUTLINE service routine

IKJSCAN
Command Scan Service Routine

IKJSTCK
STACK service routine

IKJTBLS
Table look-up service

IKJURPS
Unauthorized resource processor service

Notes:

1. A module that uses the CALLTSSR macro instruction must include the CVT mapping macro (CVT), which is provided in SYS1.MACLIB.

2. A module that invokes IKJADTAB, IKJCAF, IKJEFTSI, IKJEFTST, IKJBLS and IKJURPS must also include the TSVT mapping macro (IKJTSVT), which is provided in SYS1.MACLIB.

Syntax and operands

Figure 10 on page 34 shows the execute form of the CALLTSSR macro instruction. There is no list form. Each operand is explained following the figure.

```
[symbol]      CALLTSSR      EP=entry point name
                           [MF=(E,{list address })]
                           [   (  {(register)   })]
```

Figure 10. The CALLTSSR macro instruction

EP=*entry point name*
specifies one of the following names: IKJADTB (for IKJADTAB), IKJCAF, IKJDAIR, IKJEFF02, IKJEHCIR, IKJEHDEF, IKJGETL, IKJPARS, IKJPTGT, IKJPUTL, IKJSCAN, IKJSTCK, IKJTBLS, IKJTSFI (for IKJEFTSI), IKJTSFT (for IKJEFTST), or IKJURPS.

MF=E
indicates that this is the execute form of the macro instruction.

***list address* | (*register*)**
specifies the address, or register that contains the address, of a parameter list to be passed to the service routine.

Example using TSO/E service routines with CALLTSSR

This example shows how the CALLTSSR macro instruction can be used to invoke the Parse Service Routine (IKJPARS) and pass the parse parameter list (PPL) as input.

```
CALLTSSR EP=IKJPARS,MF=(E,PPL)
```

Chapter 5. Verifying subcommand names with IKJSCAN

This chapter describes how a Command Processor can use the Command Scan Service Routine to determine the validity of a subcommand name

Functions performed by the Command Scan Service Routine

If you write your own command processors, you need a method of determining whether subcommand names entered into the system are syntactically correct. The Command Scan Service Routine provides this function by searching the command buffer for a valid subcommand name. Command scan can be invoked by any Command Processor that processes subcommands. It can also be used to scan the reply to a prompt message.

Figure 11 on page 35 shows the format of the command buffer.

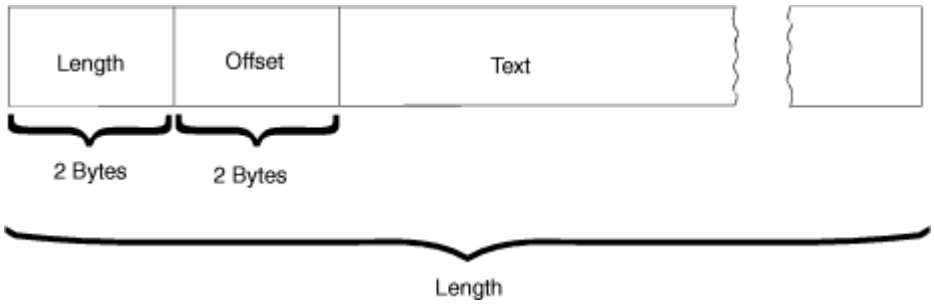


Figure 11. Format of the command buffer

When your Command Processor invokes the Command Scan Service Routine, the two-byte length field contains the length of the command buffer. The two-byte offset field is set to zero.

The Command Scan Service Routine examines the command buffer and performs the following functions:

- It translates all lowercase characters in the subcommand name to uppercase.
- If a valid operand is present, it resets the offset to the number of text bytes preceding the first non-blank character in the operand field. If a valid operand is not present, the offset equals the length of the text portion of the buffer.
- It returns a pointer to the subcommand name, the length of the subcommand name, and a code explaining the results of its scan to the calling routine.
- It optionally checks the syntax of the subcommand name.
- It recognizes an implicit EXEC command that has a percent sign as the first character.
- It handles leading blanks and embedded comments.

Syntax requirements for command and subcommand names

If you write your own Command Processor, and you intend to use the Command Scan Service Routine to check for a valid subcommand name, the name you choose must meet the following syntax requirements:

- The first character must be alphabetic or one of the special characters $, #, @.
- The remaining characters must be alphanumeric.
- The length of the subcommand name must not exceed eight characters.
- The command delimiter must be a separator character.

Include one or more numerals in the name to differentiate it from the IBM-supplied command names, which do not include numerals.

The Command Scan Service Routine accepts double-byte character set (DBCS) strings in addition to EBCDIC character strings. The shift-out character (X'0E') indicates a change from EBCDIC to DBCS; the shift-in character (X'0F') indicates the reverse. Each double-byte character requires a double-byte representation so that valid DBCS strings contain an even number of bytes. With the exception of blank, which is X'4040', each byte has a value from X'41' to X'FE'.

Double-byte characters can appear in comments and certain types of strings of user data. For a discussion of the types of strings that can contain double-byte characters, see Chapter 6, “Verifying command and subcommand operands with parse,” on page 43.

The following table shows the various character types recognized by the Command Scan Service Routine. Unless otherwise indicated, alphanumeric characters are (1) alphabetic (A-Z), (2) numeric (0-9), and (3) the special characters $, #, @.

Table 9. Character types recognized by the parse service routine

		Separator	$ # @	Alphabetic	Numeric	Command delimiter	Delimiter	Special
Comment	/*	X						
Horizontal Tab	HT	X				X		
Blank	b	X				X		
Comma	,	X				X		
Dollar Sign	$		X					
Number Sign	#		X					
At Sign	@ a-z A-Z 0-9		X	 X X	 X			
New line	NL					X	X	
Period	.					X		X
Left parenthesis	(					X	X	
Right parenthesis	)					X	X	
Ampersand	&					X		X
Asterisk	*							X
Semicolon	;					X	X	
Minus sign, hyphen	-					X		X
Slash	/					X	X	
Apostrophe	‘					X	X	
Equal sign	=					X	X	
Cent sign	c							X
Less than	<							X
Greater than	>							X
Plus sign	+							X
Logical OR	|						X
Exclamation point	!							X
Logical NOT	¬							X
Percent sign	%							X
Dash	-							X
Question mark	?							X

Table 9. Character types recognized by the parse service routine (continued)								
		Separator	$ # @	Alphabetic	Numeric	Command delimiter	Delimiter	Special
Colon	:							X
Quotation Mark	"							X
Shift-out[1]	X'0E'							X
Shift-in[1]	X'0F'							X
[1] The shift-out and shift-in characters indicate the beginning and end of a string of double-byte character set data.								

Invoking the Command Scan Service Routine (IKJSCAN)

Your Command Processor can invoke the Command Scan Service Routine by using either the CALLTSSR or LINK macro instructions, specifying IKJSCAN as the entry point name. However, you must first create the command scan parameter list (CSPL) and place its address into general register 1.

The Command Scan Service Routine can be invoked in either 24-bit or 31-bit addressing mode. IKJSCAN can be passed input that resides above or below 16 MB in virtual storage. The caller's parameters must be in the primary address space.

The command scan parameter list

The command scan parameter list (CSPL) is a six-word parameter list containing addresses required by the Command Scan Service Routine. To ensure that your Command Processor is reentrant, build the CSPL in subpool 1 in an area that the Command Processor obtains by issuing the GETMAIN macro instruction. Figure 12 on page 37 shows the parameter list structure that your command processor must create as input to the Command Scan Service Routine.

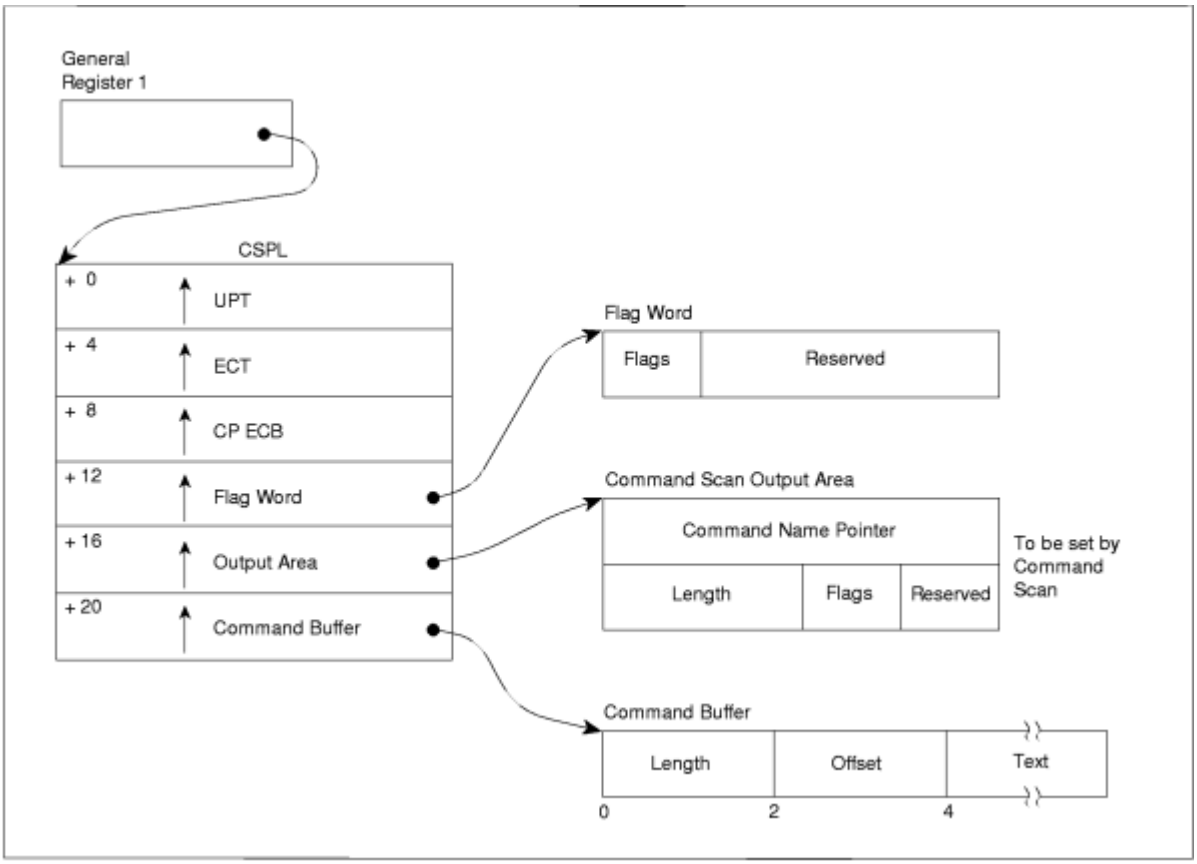


Figure 12. The parameter list structure passed to command scan

Use the IKJCSPL DSECT, which is provided in SYS1.MACLIB, to map the fields in the CSPL. Table 10 on page 38 shows the format of the command scan parameter list.

Table 10. The command scan parameter list

Number of bytes	Field name	Contents or meaning
4	CSPLUPT	The address of the user profile table. This address is passed to a Command Processor in the CPPL.
4	CSPLECT	The address of the environment control table. This address is passed to a Command Processor in the CPPL.
4	CSPLECB	The address of the command processor's event control block.
4	CSPLFLG	The address of a fullword, obtained via the GETMAIN macro instruction by the routine linking to command scan, and located in subpool 1. The first byte of the word pointed to contains flags set by the calling routine.
4	CSPLOA	The address of an 8-byte command scan output area, located in subpool 1. The output area is obtained by the calling routine via a GETMAIN macro instruction. It is filled in by the Command Scan Service Routine before it returns control to the calling routine. (See Figure 12 on page 37.)
4	CSPLCBUF	The address of the command buffer.

Passing flags to the Command Scan Service Routine

The fourth word of the CSPL, CSPLFLG, is a flag word that your Command Processor must build in subpool 1 in an area that the Command Processor obtains by issuing the GETMAIN macro instruction. Command scan uses only the first byte of the field.

Your Command Processor must set the flag byte before invoking the Command Scan Service Routine to indicate whether you want the command to be syntax checked. The flag byte has the following meanings:

Value	Meaning
X'00'	Syntax check the command name.
X'80'	Do not syntax check the command name.

After your Command Processor invokes the Command Scan Service Routine, it should free the area obtained for the flag field.

The command scan output area

The Command Scan Service Routine returns the results of its scan to the calling program by filling in a two-word command scan output area (CSOA). Your Command Processor must build the CSOA in subpool 1 in an area that your command processor obtains by issuing the GETMAIN macro instruction. Your command processor must then store the address of the CSOA into the fifth word of the command scan parameter list before invoking IKJSCAN.

You can use the IKJCSOA DSECT, which is provided in SYS1.MACLIB, to map the fields in the CSOA. Table 11 on page 38 shows the format of the command scan output area.

Table 11. The command scan output area

Number of bytes	Field name	Contents or meaning
4	CSOACNM	The address of the command name if the command name is present and valid. Zero otherwise.
2	CSOALNM	Length of the command name if the command name is present and valid. Zero otherwise.
1	CSOAFLG	A flag field. Command scan sets these flags to indicate the results of its scan. See Table 12 on page 39.
1		Reserved.

After your Command Processor invokes the Command Scan Service Routine and processes its output, it should free the area obtained for the CSOA.

Output from the Command Scan Service Routine

The Command Scan Service Routine scans the command buffer and returns the results of its scan to the calling routine by filling in the command scan output area, and by updating the offset field in the command buffer. Table 12 on page 39 shows the possible CSOA settings and command buffer offset settings upon return from the Command Scan Service Routine.

Table 12. Return from command scan - CSOA and command buffer settings

Command scan output area			**Command buffer**
Flag	**Meaning**	**Length field**	**Offset set to:**
X'80'	The command name is valid and the remainder of the buffer contains non-separator characters.	Length of command name	The first non-separator following the command name.
X'40'	The command name is valid and there are no non-separator characters remaining.	Length of command name	The end of the buffer.
X'20'	The command name is a question mark.	Zero	Unchanged.
X'10'	The buffer is empty or contains only separators.	Zero	The end of the buffer.
X'08'	The command name is syntactically not correct.	Zero	Unchanged.
X'04'	The command is an implicit EXEC command.	Length of command name	The first non-separator following the command name.

Return codes from the Command Scan Service Routine

The Command Scan Service Routine returns the following codes in general register 15 to the program that invoked it:

Code	Meaning
0	Command scan completed successfully.
4	Command scan was passed incorrect parameters.

Example using the Command Scan Service Routine

The sample assembler code in Figure 13 on page 40 demonstrates the use of the Command Scan Service Routine to syntax check a subcommand name. Suppose the command buffer passed to command scan contains the following subcommand:

```
SUBCMD  OPERAND1  OPERAND2
```

When IKJSCAN returns control, the offset field in the command buffer contains the value 7, the number of bytes that precede OPERAND1 in the command buffer.

```
SCANEX   CSECT ,
SCANEX   AMODE 31                COMMAND'S ADDRESSING MODE
SCANEX   RMODE ANY               COMMAND'S RESIDENCY  MODE
SCANEX   CSECT
         STM   R14,R12,12(R13)    SAVE CALLER'S REGISTERS
         LR    R11,R15            ESTABLISH ADDRESSABILITY WITHIN
         USING SCANEX,R11         THIS CSECT
         LR    R9,R1              SAVE THE POINTER TO THE CPPL
         GETMAIN RU,LV=WORKSIZE   OBTAIN A DYNAMIC WORK AREA
*
         LR    R10,R1
         USING WORK_AREA,R10      ESTABLISH ADDRESSABILITY
         ST    R10,8(R13)         PUT THE ADDRESS OF MY SAVE AREA
*                                 INTO CALLER'S SAVE AREA
         ST    R13,4(R10)         PUT THE ADDRESS OF MY SAVE AREA
*                                 INTO MY SAVE AREA FOR CALLING
         LR    R13,R1             LOAD GETMAINED AREA ADDRESS
*
         ST    R9,CPPL_PTR
         USING  CPPL,R9           GET ADDRESSABILITY TO THE CPPL
*
         LA  R2,DYN_CSPL          POINT TO MY CSPL
         ST  R2,CSPL_PTR          SAVE CSPL POINTER
         USING CSPL,R2            GET ADDRESSABILITY TO THE CSPL
         MVC CSPLCBUF,CPPLCBUF    GET THE ADDRESS OF THE COMMAND BUFFER
         LA  R4,OUT_AREA          GET THE ADDRESS OF THE OUTPUT AREA
         ST  R4,CSPLOA            AND STORE IT IN THE CSPL
         MVC CSPLUPT,CPPLUPT      MOVE IN THE UPT ADDRESS
         MVC CSPLECT,CPPLECT      MOVE IN THE ECT ADDRESS
         LA  R4,ECB               GET THE ADDRESS OF THE ECB
         ST  R4,CSPLECB           AND STORE IT IN THE CSPL
         LA  R4,FLAGWORD          GET THE FLAGWORD ADDRESS
         ST  R4,CSPLFLG           AND STORE IT IN THE CSPL
         XC  ECB,ECB              SET THE ECB TO ZERO
*
         CALLTSSR EP=IKJSCAN,MF=(E,CSPL)   INVOKE IKJSCAN
*
         ST    R15,RETCODE         SAVE THE RETURN CODE
*
*
*   TEST THE RETURN CODE AND EXAMINE THE COMMAND SCAN OUTPUT AREA.
*   PROCESS ACCORDINGLY.
*        .
*        .
*        .
*
         DROP R2
         DROP R9
*
* PERFORM CLEANUP PROCESSING
*
*
         L     R5,RETCODE         GET THE RETURN CODE
         LR    R1,R13             POINT TO THE WORK AREA
         L     R13,4(R13)         CHAIN TO PREVIOUS SAVE AREA
         FREEMAIN RU,LV=WORKSIZE,A=(1)
         L     R14,12(R13)        HERE'S OUR RETURN ADDRESS
         LR    R15,R5             HERE'S THE RETURN CODE
         LM    R0,R12,20(R13)     RESTORE REGS 0-12
         BSM   0,14               RETURN TO INVOKER
```

Figure 13. An example using the Command Scan Service Routine

Figure of 'An example using the Command Scan Service Routine' **(Continued)**

```
*********************************************************************
*                                                                   *
* DECLARES FOR DYNAMIC VARIABLES                                    *
*                                                                   *
*********************************************************************
WORK_AREA     DSECT
SAVEAREA      DS 0CL72            STANDARD SAVE AREA
              DS  F               UNUSED
              DS  F               BACKWARD SAVE AREA POINTER
              DS  F               FORWARD SAVE AREA POINTER
REG14         DS  F               CONTENTS OF REGISTER 14
REG15         DS  F               CONTENTS OF REGISTER 15
REG0          DS  F               CONTENTS OF REGISTER 0
REG1          DS  F               CONTENTS OF REGISTER 1
REG2          DS  F               CONTENTS OF REGISTER 2
REG3          DS  F               CONTENTS OF REGISTER 3
REG4          DS  F               CONTENTS OF REGISTER 4
REG5          DS  F               CONTENTS OF REGISTER 5
REG6          DS  F               CONTENTS OF REGISTER 6
REG7          DS  F               CONTENTS OF REGISTER 7
REG8          DS  F               CONTENTS OF REGISTER 8
REG9          DS  F               CONTENTS OF REGISTER 9
REG10         DS  F               CONTENTS OF REGISTER 10
REG11         DS  F               CONTENTS OF REGISTER 11
REG12         DS  F               CONTENTS OF REGISTER 12
CPPL_PTR      DS  F               ADDRESS OF THE CPPL
CSPL_PTR      DS  F               ADDRESS OF THE CSPL
DYN_CSPL      DS  6F              STORAGE FOR THE CSPL
OUT_AREA      DS  2F              COMMAND SCAN OUTPUT AREA
FLAGWORD      DS  F               FLAG WORD
ECB           DS  F               ECB
RETCODE       DS  F               RETURN CODE
WORKSIZE  EQU   *-WORK_AREA       DESCRIBES LENGTH OF THE
*                                 DYNAMIC WORK AREA
*
         IKJCPPL                  COMMAND PROCESSOR PARAMETER LIST
LCPPL    EQU   *-CPPL             DESCRIBES LENGTH OF THE CPPL
*
         CVT  DSECT=YES           CVT NEEDED FOR CALLTSSR
         IKJCSPL                  COMMAND SCAN PARAMETER LIST
         IKJCSOA                  COMMAND SCAN OUTPUT AREA
*********************************************************************
*                                                                   *
*     REGISTER EQUATES                                              *
*                                                                   *
*********************************************************************
R0       EQU   0
R1       EQU   1
R2       EQU   2
R3       EQU   3
R4       EQU   4
R5       EQU   5
R6       EQU   6
R7       EQU   7
R8       EQU   8
R9       EQU   9
R10      EQU   10
R11      EQU   11
R12      EQU   12
R13      EQU   13
R14      EQU   14
R15      EQU   15
         END   SCANEX
```

Chapter 6. Verifying command and subcommand operands with parse

This chapter describes how to use the Parse Service Routine in a command processor to determine the validity of command and subcommand operands. The first three sections, "Overview of the Parse Service Routine (IKJPARS)" on page 43, "Character types accepted by the Parse Service Routine" on page 44, and "Services provided by the Parse Service Routine" on page 47, present the terminology and concepts that are necessary to understand the functions of the parse service routine. The remainder of this chapter consists of a step-by-step explanation of how to use the parse service routine, followed by detailed discussions of each of the steps in the process.

Overview of the Parse Service Routine (IKJPARS)

If you write your own Command Processors to run under TSO/E, you need a method of determining whether command or subcommand operands entered into the system are syntactically correct. The Parse Service Routine performs this function by searching the command buffer for valid operands.

There are two types of operands that are recognized by the Parse Service Routine: positional operands and keyword operands. Positional operands occur first, and must be in a specific order. Keyword operands can be entered in any order, as long as they follow all of the positional operands. Positional operands or their placeholders (,) cannot be omitted when followed by keyword operands.

Before invoking the Parse Service Routine, your Command Processor must create a parameter control list (PCL), which describes the permissible operands. Parse compares the information supplied by your Command Processor in the PCL to the operands in the command buffer. Each acceptable operand must have an entry built for it in the PCL; an individual entry is called a parameter control entry (PCE).

The Parse Service Routine returns the results of scanning and checking the operands in the command buffer to the Command Processor in a parameter descriptor list (PDL). The entries in the PDL, called parameter descriptor entries (PDEs), contain indications of specified options, pointers to data set names, or pointers to the subfields entered with the command operands.

When your Command Processor invokes the Parse Service Routine, it must pass a parse parameter list (PPL), which contains pointers to control blocks and data areas that are needed by parse. Addresses needed to access the PCL and PDL are included in the parse parameter list.

The parse macro instructions

Use the parse macro instructions in your Command Processor to:

- Build a PCL describing the valid command or subcommand operands.
- Establish symbolic references for the PDL returned by the Parse Service Routine. The labels used by your Command Processor on the various parse macro instructions allow you to access the fields in the DSECT which maps the PDL.

See Table 15 on page 66 for a description of the parse macro instructions and their functions.

Figure 14 on page 44 shows the interaction between a Command Processor and the Parse Service Routine.

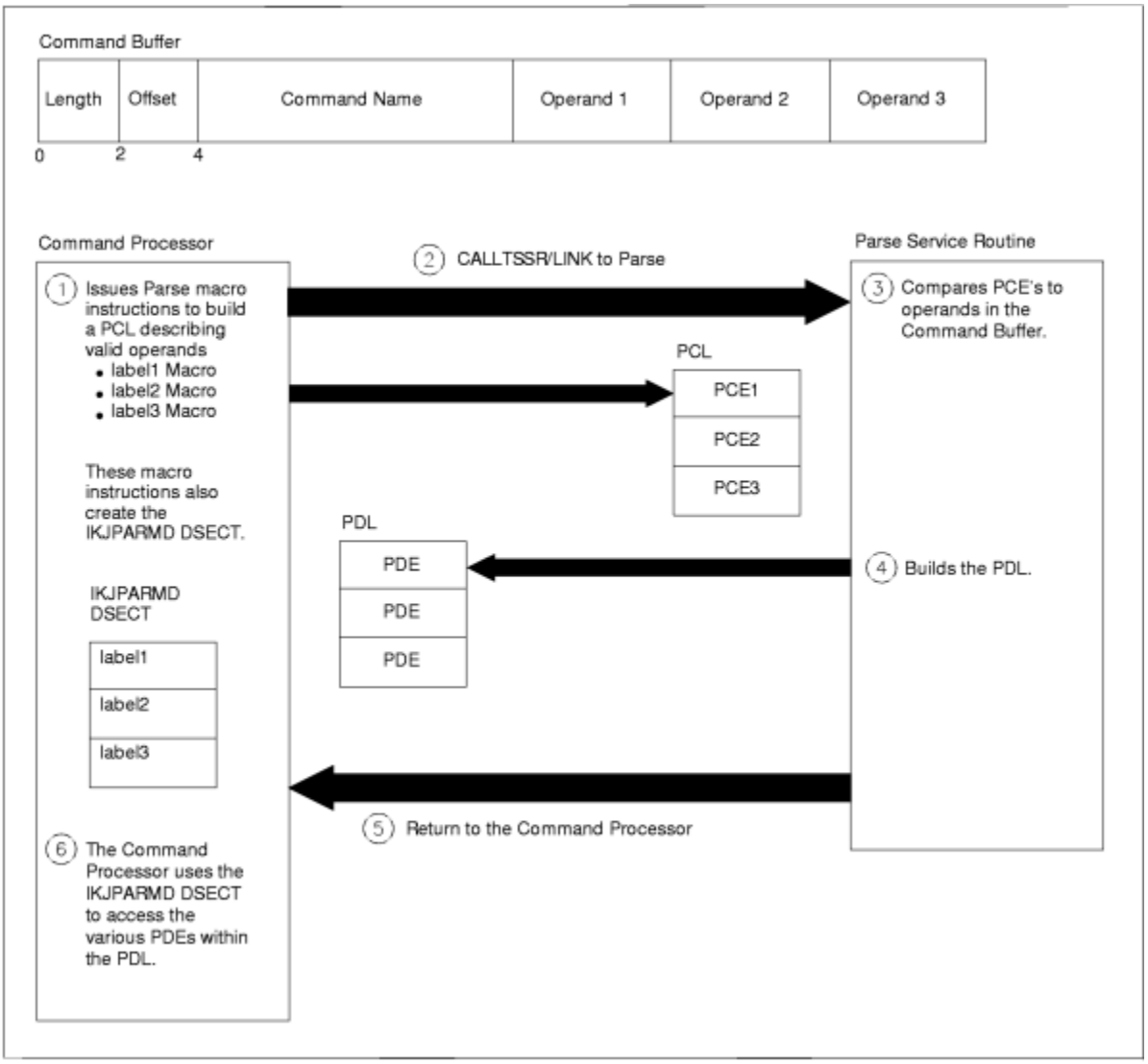


Figure 14. An example of a Command Processor using the Parse Service Routine

Character types accepted by the Parse Service Routine

The following table shows the various character types that are recognized by the Parse Service Routine. Throughout this chapter, the alphanumeric characters are as follows, unless otherwise indicated.

Alpha
A - Z

Numeric
0 - 9

Special
$, #, @

Table 13. Character types recognized by the parse service routine

		Separator	$ # @	Alphabetic	Numeric	Command delimiter	Delimiter	Special
Comment	/*	X						
Horizontal Tab	HT	X				X		
Blank	b	X				X		
Comma	,	X				X		

Table 13. Character types recognized by the parse service routine (continued)

		Separator	$ # @	Alphabetic	Numeric	Command delimiter	Delimiter	Special
Dollar Sign	$		X					
Number Sign	#		X					
At Sign	@ a-z A-Z 0-9		X	 X X	 X			
New line	NL					X	X	
Period	.					X		X
Left parenthesis	(					X	X	
Right parenthesis	)					X	X	
Ampersand	&					X		X
Asterisk	*							X
Semicolon	;					X	X	
Minus sign, hyphen	-					X		X
Slash	/					X	X	
Apostrophe	‘					X	X	
Equal sign	=					X	X	
Cent sign	c							X
Less than	<							X
Greater than	>							X
Plus sign	+							X
Logical OR	|							X
Exclamation point	!							X
Logical NOT	¬							X
Percent sign	%							X
Dash	-							X
Question mark	?							X
Colon	:							X
Quotation Mark	"							X
Shift-out[1]	X'0E'							X
Shift-in[1]	X'0F'							X

[1] The shift-out and shift-in characters indicate the beginning and end of a string of double-byte character set data.

Treatment of comment character /* by the Parse Service Routine

The Parse Service Routine recognizes blanks, tabs, commas, and comments as separator characters between command operands. Comments are used with TSO/E commands in two flavors:

1. As an embedded comment, separated from the command part by a starting delimiter of /* and an ending delimiter of */, for example:

   ```
   listd /* my data sets */ (data_set_list)
   ```

   This is the required form if the command is continued after the comment.

2. As an open comment that is not ended by an ending delimiter of */, for example:

   ```
   listd (data_set_list) /* my data sets
   ```

Here, the comment is the last part of the line and the ending delimiter of */ is not required. Everything what follows the starting /* on this logical line is treated as a comment.

Also ending a comment with */ is a convention, it is not a requirement.

The Parse Service Routine treats the starting delimiter /* and the ending delimiter */ as separator characters in the same manner as it does with tabs, blanks, and commas, when scanning and checking the command buffer content.

- If found within a quoted string, /* and */ are treated as literal characters, no matter whether they appear paired, single, or in reverse order.
- Outside quoted strings /* and */ are treated as comment delimiters. The delimiters and everything between them is removed by the Parse Service Routine and are not accessible for further processing.

  A single occurrence of /* without ending */ makes the Parse Service Routine to ignore the starting delimiter and everything what follows on the logical line.

  A single occurrence of */ without starting /* makes the Parse Service Routine to treat */ as literal characters.

Acceptance of double-byte character set data

The Parse Service Routine accepts double-byte character set (DBCS) strings in addition to EBCDIC character strings. The shift-out character (X'0E') indicates a change from EBCDIC to DBCS; the shift-in character (X'0F') indicates the reverse. Each double-byte character requires a double-byte representation so that valid DBCS strings contain an even number of bytes. Except for blank, which is X'4040', each byte has a value from X'41' to X'FE'. If the DBCS string contains an incorrect character, parse replaces it with X'4195'.

Double-byte characters can appear in comments and certain types of strings of user data. If the programming language you are using supports DBCS data, default values can also contain valid DBCS strings. DBCS strings that appear where they are not accepted could cause an error condition. The types of user strings that can contain DBCS data along with the associated parse macro and operand follows:

Type of String	Macro	Operand
Self delimiting string	IKJPOSIT	STRING
Quoted string	IKJPOSIT	QSTRING
Parenthesized string	IKJPOSIT	PSTRING
Value string	IKJPOSIT	VALUE
Quoted character constant	IKJTERM	CONSTANT
Quoted character string	IKJIDENT	CHAR or HEX

Parse does not accept DBCS strings in prompting mode. In addition, you cannot use DBCS strings in quoted data set names, quoted passwords for data sets, or quoted passwords for user IDs because MVS does not accept DBCS strings in those cases. However, the parse macro, IKJPOSIT, treats X'0E' and X'0F' as DBCS delimiters in quoted data set names (DSNAME and DSTHING parameters), quoted passwords for data sets (DSNAME and DSTHING parameters), and quoted passwords for user IDs (USERID, USERID8, UID2PSWD, and UID82PWD parameters). Special characters within passwords are allowed for USERID8 and UID82PWD.

Check all hexadecimal data that you pass to parse to be sure that X'0E' and X'0F' represent the shift-out and shift-in characters when appropriate. Previously, parse treated those characters simply as hexadecimal data. Now, when used in the strings mentioned earlier in this topic, parse treats them as DBCS delimiters. Therefore, change X'0E' and X'0F' to some other values if they do not represent the shift-out and shift-in characters and you are passing them through the parse service.

Services provided by the Parse Service Routine

The function of the Parse Service Routine is to syntax check command operands within the command buffer against the PCL, and build a PDL containing the results of the syntax check. In addition, the Parse Service Routine provides the following services that can be selected by the calling routine:

- It prompts the user if required operands are missing or incorrect.
- It issues messages for certain error conditions, or the validity checking routine or verify exit routine issues messages before requesting that parse terminate.
- It appends second-level messages, supplied by the calling program, to prompting messages.
- It passes control to a validity checking routine, supplied by the calling program, to do additional checking on a positional operand.
- It passes control to a verify exit routine, supplied by the calling program, to perform checking on a keyword operand that is not specifically defined in the PCL.
- It translates the command operands to uppercase.
- It substitutes default values for missing operands.
- It inserts implied keywords.

Prompting the user for missing or required operands

The Parse Service Routine prompts the terminal user if the command operands found are incorrect or if required operands are missing. It allows the terminal user to enter a missing operand or correct an incorrect one without having to reenter the entire command. The Parse Service Routine prompts, and the terminal user must respond, in the following situations:

- A userid or dsname was entered with a slash but without a password.
- An operand is syntactically not valid.
- A keyword is ambiguous, that is, it is not clear to the Parse Service Routine which keyword of several similar ones is being entered.
- A required positional operand is missing. The requirement for a particular positional operand and the prompting message to be issued if that operand is not present, are specified to the Parse Service Routine through the PROMPT operand of the IKJPOSIT, IKJTERM, IKJOPER, IKJRSVWD, and IKJIDENT macro instructions. The Parse Service Routine issues the prompting message supplied in the macro instruction.
- A validity checking routine indicates that an operand is incorrect.
- A verify exit routine indicates that an operand is incorrect and that parse should prompt the user.

How parse processes responses

There are several rules that govern how the Parse Service Routine processes responses entered from the terminal after a prompt:

1. All of the new data entered is parsed before the scan of the original command is resumed.
2. Unless otherwise stated in the command syntax definition, the new operand must be entered as it is entered in the original command. See “Defining command operand syntax” on page 51 for exceptions to this rule.
3. In general, a user can enter additional operands along with the data prompted for. It must be kept in mind, however, that all of the new data entered is parsed before the scan of the material in the original command buffer is resumed.

   Positional operands must occur first in the string of operands, and they must be in a specific order. Therefore, a problem could occur in a situation where a command is entered followed by two positional operands and a keyword, and the first positional operand is not valid. The Parse Service Routine issues a prompt for the first positional operand. When the user at the terminal reenters that first positional operand, it would be incorrect to enter additional keywords along with it. The additional

keywords would be scanned before the second positional operand and an error condition would result when the Parse Service Routine returned to the original command buffer and found a positional operand.

Note: If the operand prompted for is within a subfield, only operands valid within that subfield can be entered along with the operand prompted for.

4. In general, a null response is acceptable only for optional operands. However, if the user enters a null response for an optional operand that has a default, parse inserts the default. If a prompt for a required operand is answered by a null response from the terminal, parse reissues the prompt message. The Parse Service Routine continues prompting until a correct operand is entered. The terminal user can request termination by entering an attention.

   Parse always accepts a null response to a prompt for a password, whether or not the dsname or userid operands are required. The program that invokes the Parse Service Routine must ensure that the correct password was entered if one was required, by checking the password pointed to by the PDE returned by the Parse Service Routine.

5. If a required operand which can be entered in the form of a list is missing, or if it was entered as a single operand (not as a list), and that single operand is incorrect, parse will not accept a list after the prompt. The user at the terminal must enter a single operand.

   If, however, the item was entered as a list but an item within the list is incorrect, the Parse Service Routine accepts one or more operands after the prompt. The Parse Service Routine considers these newly entered operands to be part of the original list. Operands that are not valid in the list cannot be entered from the terminal in response to this prompt.

   If the last item in a list is found to be not valid, parse only accepts one operand after a prompt.

6. If the Parse Service Routine determines that an operand is not valid, the not valid portion of the operand is indicated in the error message. The remainder of the operand is not yet parsed. The user must reenter as much of the incorrect operand as was indicated in the error message. For example, this can occur if a dsname operand or userid operand is entered with blanks between the dsname or userid and the password. The dsname or userid can be not valid but the password is still good and will be parsed after a new dsname or userid is entered in response to the prompt.

Although the Parse Service Routine always attempts to obtain syntactically correct operands before returning to the calling routine, this is not always possible. The terminal user could have requested that no prompt messages be sent to the terminal, or the command being parsed could have come from a procedure. In these cases, the Parse Service Routine issues an error message and returns a code to the calling routine indicating that a correct command could not be obtained. Any second-level messages that would ordinarily be appended to the request for new data are appended to the error message.

Issuing error messages when parse does not complete successfully

If the Parse Service Routine does not complete successfully, register 15 contains a return code. If that return code is 4, parse has already issued a message. When the return code is either 20 or 32, the validity checking routine or verify exit routine, respectively, has issued a message before it requested that parse terminate.

Issuing second-level messages

Your Command Processor can supply second-level messages to be chained to any prompt message issued for a positional operand (keyword operands are never required). Use the HELP operand of the IKJPOSIT, IKJTERM, IKJOPER, IKJRSVWD or IKJIDENT macro instructions to supply these second-level messages to the Parse Service Routine. You can supply up to 255 second-level messages for each positional operand. One second-level message is issued each time a question mark is entered from the terminal.

If a user-provided validity checking routine returns the address of a second-level message to the Parse Service Routine, that second-level message or chain will be written out in response to question marks entered from the terminal. The original second-level chain, if one was present, is deleted.

The format of these second-level messages is the same as the HELP second-level message portion of the PCE for the macro from which the validity checking routine received control.

Using the prompt mode HELP function

If a question mark is entered and no second-level messages were provided, or they have all been issued in response to previous question marks, parse determines whether it can generate a valid HELP command to provide the user with additional information.

If the ECTNOQPR bit in the environment control table (ECT) is zero, then the prompt mode HELP function is active and parse processing generates a HELP command on the user's behalf. Parse ensures that only one HELP command is issued during a prompting sequence for a given operand. If the user enters another question mark after viewing the online usage information, the NO INFORMATION AVAILABLE message is issued.

When your Command Processor receives control, the ECTNOQPR bit in the ECT is set to zero, which activates the prompt mode HELP function. However, parse sets ECTNOQPR to one before it returns control to the Command Processor. Therefore, the prompt mode HELP function is not active during subsequent invocations of parse from your Command Processor or from any subcommands attached by your Command Processor.

If your Command Processor accepts subcommands and wants the prompt mode HELP function to be available for a subcommand, it should set ECTNOQPR to zero before attaching the subcommand. The Command Processor should also ensure that the ECTPCMD and ECTSCMD fields in the ECT contain the command name and the subcommand name respectively.

If you do not want the prompt mode HELP function to be active, your Command Processor should set the ECTNOQPR bit to one before it invokes parse for the first time.

Passing control to validity checking routines

Your Command Processor can provide a validity checking routine to do additional checking on a positional operand. This routine receives control after the Parse Service Routine has determined that the operand is non-null and syntactically correct. Each positional operand can have a unique validity checking routine. “Using validity checking routines” on page 100 describes what you must do to provide a validity checking routine.

Passing control to verify exit routines

Your Command Processor can provide verify exit routines to perform checking when the Parse Service Routine encounters either of the following in the command buffer:

- Unidentified keyword operands
- Unidentified keyword operands within a subfield.

To indicate the presence of a verify exit routine, specify its address on the IKJUNFLD macro instruction. When the Parse Service Routine encounters a keyword operand or subfield operand in the command buffer that is not specifically defined in the PCL, it determines whether a PCE has been created by the IKJUNFLD macro instruction. If parse encounters such a PCE, it gives control to the verify exit routine; if it does not, the operand is treated as not valid. The Parse Service Routine uses only the first specification of the IKJUNFLD macro instruction when unidentified keyword operands are present in the command buffer. Similarly, parse uses only the first specification of the IKJUNFLD macro instruction within a subfield specification when an unidentified keyword is present within a subfield. “Using verify exit routines” on page 102 describes what you must do to provide verify exit routines.

Translation to uppercase

The Parse Service Routine normally translates positional operands to uppercase unless the calling routine specifies ASIS in the IKJPOSIT or IKJIDENT macro instructions. The first character of a value operand, the type-character, is always translated to uppercase, however. Parse translates the string that follows the type character to uppercase unless ASIS is coded in the describing macro instructions.

When you specify ASIS on the IKJPOSIT or IKJIDENT macro instruction, syntax checking is done only on the characters, and will not ensure that the operands are actually valid. That is, if the data set name specified in the DSNAME operand is in lowercase, parse will return a return code of zero, although lowercase data set names are usually invalid. If this type of validity check needs to be performed, the user can create a validity checking routine and specify it on the VALIDCK= operand on IKJPOSIT or IKJIDENT macros.

Double-byte character set strings are an exception to this rule. Regardless of whether you specify ASIS, parse does not translate the contents of the double-byte character set string to upper case.

Insertion of default values

Positional operands (except delimiter and space) and keyword operands can have default values. These default values are indicated to the Parse Service Routine through the DEFAULT= operand of the IKJPOSIT, IKJTERM, IKJOPER, IKJRSVWD, IKJIDENT, and IKJKEYWD macro instructions. When a positional or a keyword operand is omitted, for which a default value has been specified, the Parse Service Routine inserts the default value.

The Parse Service Routine also inserts the default value you specified if an operand is not valid and the terminal user enters a null line in response to a prompt.

Insertion of keywords

Some keyword operands can imply other keyword operands. You can specify that other keywords are to be inserted into the parameter string when a certain keyword is entered. Use the INSERT operand of the IKJNAME macro instruction to indicate that a keyword or a list of keywords is to be inserted following the named keyword. Parse processes inserted keywords as though they were entered from the terminal.

What you need to do to use the Parse Service Routine

This section gives a step-by-step description of what you must do to use the Parse Service Routine. The sections that follow provide more detailed information on each of the major steps.

Follow these steps when using the Parse Service Routine:

1. Define the syntax of the operands of the command or subcommand. This topic is discussed in “Defining command operand syntax” on page 51.
2. Use the parse macro instructions to build the parameter control list (PCL) that describes the command or subcommand operand syntax. The parse macro instructions are described in “Using the parse macro instruction to define command syntax” on page 66.
   - Use the IKJPARM macro instruction to begin the parameter control list (PCL).
   - Use the appropriate parse macro instructions to build the parameter control entries (PCEs) that parse will use to check the syntax of the operands.
   - Use the IKJENDP macro instruction to indicate the end of the parameter control list (PCL) for the command or subcommand.
3. Provide installation exits for operand checking (optional).
   - Write validity checking routines to do additional checking on positional operands. See “Using validity checking routines” on page 100 for a discussion of this topic.
   - Write verify exit routines to check unidentified keyword operands or unidentified keyword operands within a subfield. See “Using verify exit routines” on page 102 for a discussion of this topic.
4. Pass control to the Parse Service Routine. See “Passing control to the Parse Service Routine” on page 105.
5. Check the return code passed by the Parse Service Routine in general register 15. Return codes are listed in “Checking return codes from the Parse Service Routine” on page 106.

6. Examine the results of the scan of the command buffer returned by parse in the parameter descriptor list (PDL). See “Examining the PDL returned by the Parse Service Routine” on page 108 for a description of the PDEs returned by parse.

Defining command operand syntax

If you write your own command processors, and you intend to use the Parse Service Routine to determine which operands have been entered following the command name, your command operands must adhere to the syntactical structure described in this section.

Command operands must be separated from one another by one or more of the separator characters: blank, tab, comma, or a comment (see Table 13 on page 44). The command operands end either at the end of a logical line (carrier return), or at a semicolon. If the command operands end with a semicolon, and other characters are entered after the semicolon but before the end of the logical line, the Parse Service Routine ignores the portion of the line that follows the semicolon. The parse service routine does not issue a message to indicate this condition.

The Parse Service Routine recognizes two types of command operands:

Positional operands
This type must be entered first in the parameter string, and they must be entered in a specific order.

Keyword operands
This type can be entered anywhere in the command as long as they follow all positional operands. See “Keyword operands” on page 65 for more information.

Positional operands

Positional operands must be entered first in the parameter string, and they must be in a specific order.

In general, the Parse Service Routine considers a positional operand to be missing if the first character of the operand scanned is not the character expected. For example, if an operand is supposed to begin with a numeric character and the Parse Service Routine finds an alphabetic character in that position, the numeric operand is considered missing. The Parse Service Routine then prompts for the missing operand if it is required, substitutes a default value if one is available, or ignores the missing operand if the operand is optional.

For the purpose of syntax checking, positional operands are divided into two categories:

- *Delimiter-dependent operands* include delimiters as part of their definition. See “Delimiter-dependent operands” on page 51 for more information.
- *Non-delimiter-dependent operands* do not include delimiters as part of their definition. See “Positional operands not dependent on delimiters” on page 64 for more information.

Delimiter-dependent operands

Those operands that include delimiters as part of their definition are called delimiter-dependent operands. Table 14 on page 52 shows the delimiter-dependent syntaxes that the Parse Service Routine recognizes and the macro instruction that is used to specify each type.

Table 14. Delimiter-dependent operands	
Operand	**macro instruction used to describe operand**
DELIMITER STRING VALUE ADDRESS PSTRING USERID USERID8 UID2PSWD UID82PWD DSNAME DSTHING QSTRING SPACE JOBNAME	IKJPOSIT
CONSTANT VARIABLE STATEMENT NUMBER	IKJTERM
EXPRESSION	IKJOPER
RESERVED WORD	IKJRSVWD
HEX CHAR INTEG	IKJIDENT

DELIMITER
A delimiter can be any character other than an asterisk, left parenthesis, right parenthesis, semicolon, blank, comma, tab, carrier return, digit, shift-out character (X'0E'), or shift-in character (X'0F'). A self-defining delimiter character is represented in this discussion by the symbol #. The delimiter operand is used only along with the string operand.

STRING
A string is the group of characters between two alike self-defining delimiter characters, such as

```
#string#
```

or, the group of characters between a self-defining delimiter character and the end of a logical line, such as

```
#string
```

The same self-defining delimiter character can be used to delimit two contiguous strings, such as

```
#string#string#
```

or

```
#string#string
```

A null string, which indicates that a positional operand has not been entered, is defined as two contiguous delimiters or a delimiter and the end of the logical line. If the missing string is a required operand, the null string must be entered as two contiguous delimiters. Note that a string received from a prompt or a default must not include the delimiters. See “Acceptance of double-byte character

set data" on page 46 for information about using double-byte character set data in a self-delimiting string.

VALUE

A value consists of a character followed by a string enclosed in apostrophes, such as

```
X'string'
```

The character must be alphabetic or one of the special characters $, #, @. The string can be of any length and can consist of any combination of enterable characters. If the ending apostrophe is omitted, the Parse Service Routine assumes that the string ends at the end of the logical line. If the Parse Service Routine encounters two successive apostrophes, it assumes they are part of the string and continues to scan for a single ending apostrophe. The Parse Service Routine always translates the character preceding the first apostrophe to uppercase. The value is considered missing if the first character is not alphabetic or one of the special characters $, #, @, or if the second character is not an apostrophe. See "Acceptance of double-byte character set data" on page 46 for information about using double-byte character set data in a value string.

ADDRESS

There are several forms of the ADDRESS operand. Note that blanks are not allowed within any form of the ADDRESS operand.

Absolute address

consists of from one to six hexadecimal digits followed by a period, or, in extended mode, from one to eight hexadecimal digits followed by a period. An extended absolute address must not exceed the address represented by the hexadecimal value X'7FFFFFFF'. (For more information on extended addressing, see the description of the EXTENDED operand in "Using IKJPOSIT to describe a delimiter-dependent positional operand" on page 68.)

Relative address

consists of from one to six hexadecimal digits preceded by a plus sign, or, in extended mode, from one to eight hexadecimal digits preceded by a plus sign.

General register address

consists of a decimal integer in the range 0 to 15 followed by the letter R. R can be entered in either uppercase or lowercase.

Floating-point register address

consists of an even decimal integer in the range 0 to 6 followed by the letter D (for double precision) or E (for single precision). The letter E or D can be entered in either uppercase or lowercase.

Vector register address

is of the form:

```
register-number {V} (element-number)
                {W}
```

register-number

consists of a decimal integer in the range 0-15, if V is specified. If W is specified, the register number must be an even decimal integer in the range 0-14.

V

indicates single precision. V can be entered in either uppercase or lowercase.

W

indicates double precision. W can be entered in either uppercase or lowercase.

element-number

consists of a decimal integer in the range 0 through one less than the section size, or an asterisk, (*). Asterisk indicates that all elements of the vector register are considered.

The section size, which is the number of elements in a vector register, is dependent upon the model of the CPU that has the vector facility installed. See *System/370 Vector Operations* for information on the vector facility.

Vector mask register address
consists of the decimal integer 0 followed by the letter M. M can be entered in either uppercase or lowercase.

Access register address
consists of a decimal integer in the range 0 to 15 followed by the letter A. A can be entered in either uppercase or lowercase.

Symbolic address
consists of any combination of alphanumeric characters and the break character, and may be up to 32 characters long. The first character must be either alphabetic or one of the special characters $, #, @.

Qualified address
has one of the following formats:

1. `module_name.entry_name.relative_address`
2. `module_name.entry_name`
3. `module_name.entry_name.symbolic_address`
4. `.entry_name.symbolic_address`
5. `.entry_name.relative_address`
6. `.entry_name`

module_name
any combination of one to eight alphanumeric characters, where the first is an alphabetic character or one of the special characters $, #, @

entry_name
same syntax as a *module_name*, and always preceded by a period

symbolic_address
syntax as defined above, and always preceded by a period

relative_address
syntax as defined above, and always preceded by a period.

The user can qualify symbolic or relative addresses to indicate that they apply to a particular module and CSECT as in formats “1” on page 54 to “3” on page 54. However, if the address applies to the currently active module, it is not necessary to specify *module_name*, as in formats “4” on page 54 to “6” on page 54.

Indirect address
is an absolute, relative, or symbolic address (or general register containing an address), followed by 1 to 255 indirection symbols (% or ?). When the EXTENDED keyword is specified on the IKJPOSIT macro, the user can specify the 31-bit indirection symbol, ?. The 24-bit indirection symbol, %, can also be specified. If EXTENDED is not specified, only the 24-bit indirection symbol can be used.

Note: In the following examples, hash marks indicate that the byte is not used to determine the 24-bit address.

Figure 15 on page 55 shows an example of an indirect address that is made up of a relative address with one level of 24-bit indirect addressing.

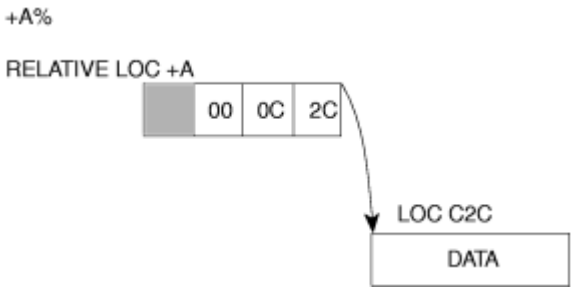


Figure 15. Example of 24-Bit Indirect Addressing

The number of indirection symbols following the address indicates the In Figure 15 on page 55, the data is at the location pointed to by bits 0-24 of relative address +A.

Figure 16 on page 55 shows how the substitution of a 31-bit indirection symbol, ?, changes the result of the resolution of an indirect address. The example assumes that EXTENDED has been specified on IKJPOSIT.

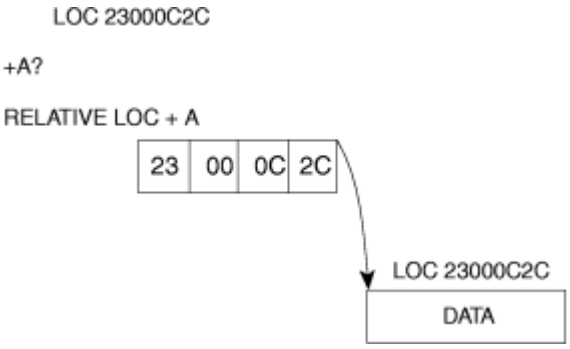


Figure 16. Example of 31-Bit Indirect Addressing

Figure 17 on page 56 shows an example of an indirect address in which 24- and 31-bit indirection symbols are combined. The example assumes that EXTENDED has been specified on IKJPOSIT.

In Figure 17 on page 56, four levels of indirect addressing are processed to resolve the indirect address.

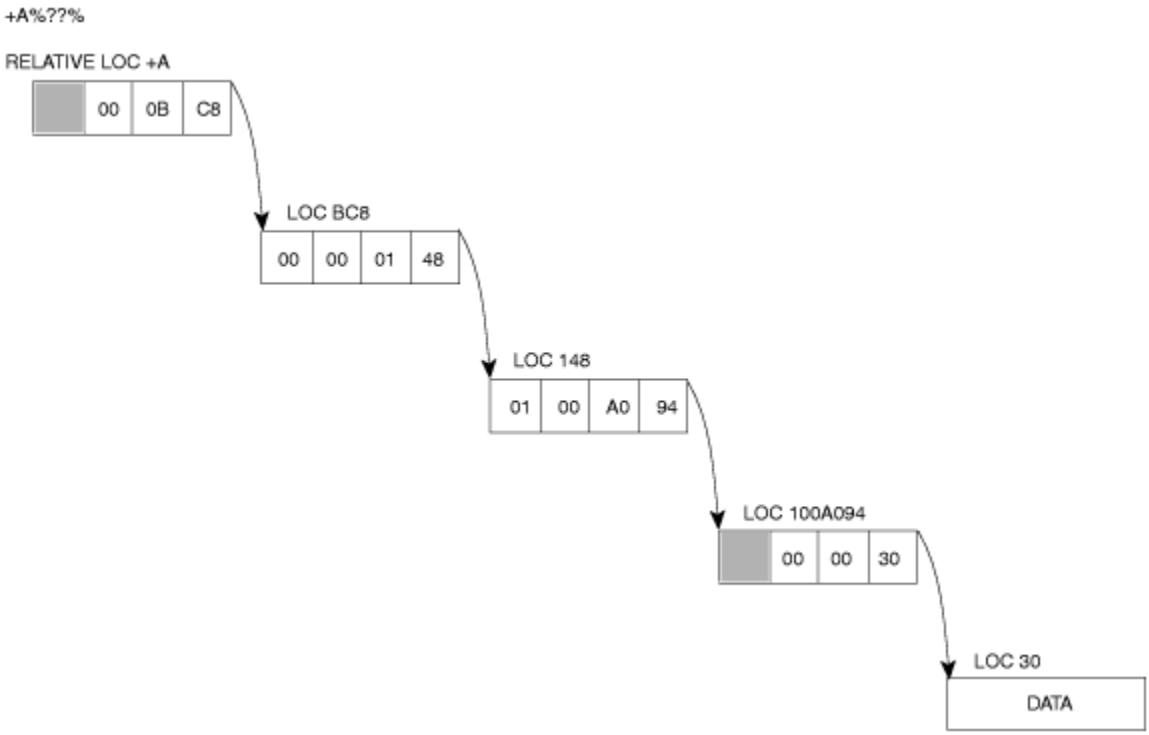


Figure 17. An Indirect Address with Mixed Indirection Symbols

Address expression
has one of two formats, depending on whether the EXTENDED keyword is specified on the IKJPOSIT macro.

1. EXTENDED not specified:

   An address expression has the following format when EXTENDED has not been specified:

   ```
   address{±}expression_value[%…][+
   {±}expression_value [%…]]…
   ```

   address
   can be an absolute, symbolic, indirect, relative, or general register address. If a general register is specified, it must be followed by at least one indirection symbol.

   expression_value
   a plus or minus displacement from an address in storage, consisting of from one to six decimal or hexadecimal digits

   - Decimal displacement is indicated by an "N" or "n" following the offset. The absence of an "N" or "n" indicates hexadecimal displacement.
   - There is no limit to the number of expression values in an address expression.

   Each expression value can be followed by from one to 255 percent signs, one for each level of indirect addressing.

   For example, *addr1*+124n, an address expression in decimal format, indicates a location 124 decimal bytes beyond *addr1*. Another example, *addr2*-AC, is an address expression in hexadecimal format and indicates a location 172 decimal bytes before *addr2*.

   The processing of an address expression, 12R%%+4N%, involving 24-bit indirect addressing, is shown in Figure 18 on page 57. The address in the expression is a general register address with two levels of indirect addressing. The result of the processing of this part of the address expression is location 1D0. The expression value indicates a displacement of four bytes beyond location 1D0 with one level of indirect addressing. The data, then, is at location 474.

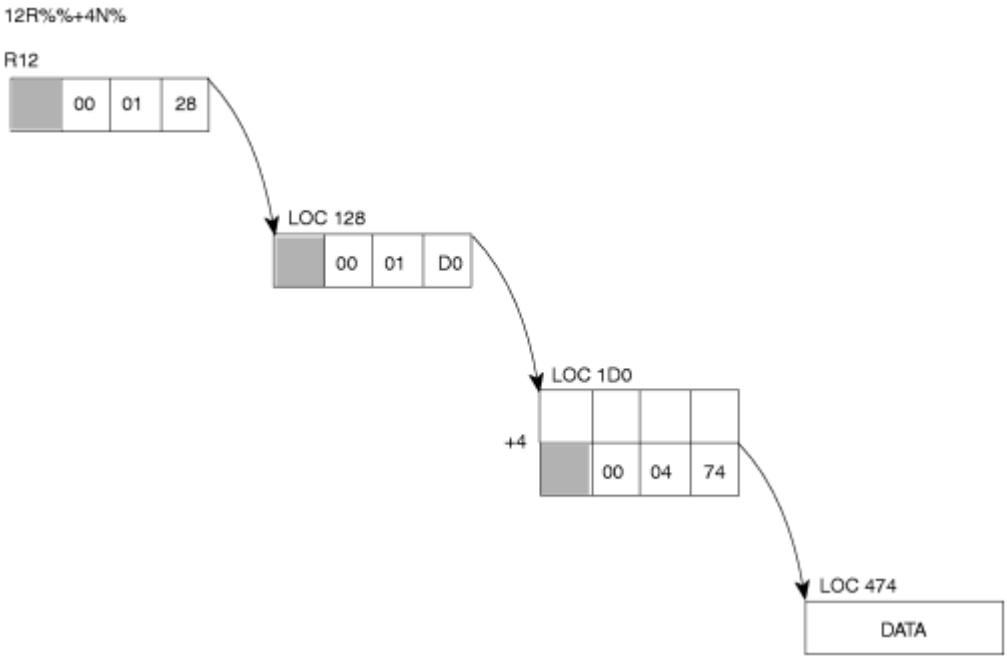


Figure 18. An Address Expression with 24-Bit Indirect Addressing

2. EXTENDED specified:

   An address expression has the following format when EXTENDED has been specified:

```
                          [        ]+
                     [       ]
 address{±}expression_value[[%]     ][+
{±}expression_value[[%]    ]]
                           [[&?] …+
][                   [[?] …]] …
```

   address
   can be an absolute, symbolic, indirect, relative, or general register address. If a general register is specified, it must be followed by at least one indirection symbol.

   expression_value
   a plus or minus displacement from an address in storage, consisting of a one- to ten-digit decimal number, or a one- to eight-digit hexadecimal number.

   - Decimal displacement is indicated by an "N" or "n" following the offset. The absence of an "N" or "n" indicates hexadecimal displacement.
   - There is no limit to the number of expression values in an address expression.

   Each expression value can be followed by from one to 255 indirection symbols (including any valid combination of question marks and percent signs), one for each level of indirect addressing.

The processing of an address expression involving both 24- and 31-bit indirect addressing is shown in Figure 19 on page 58.

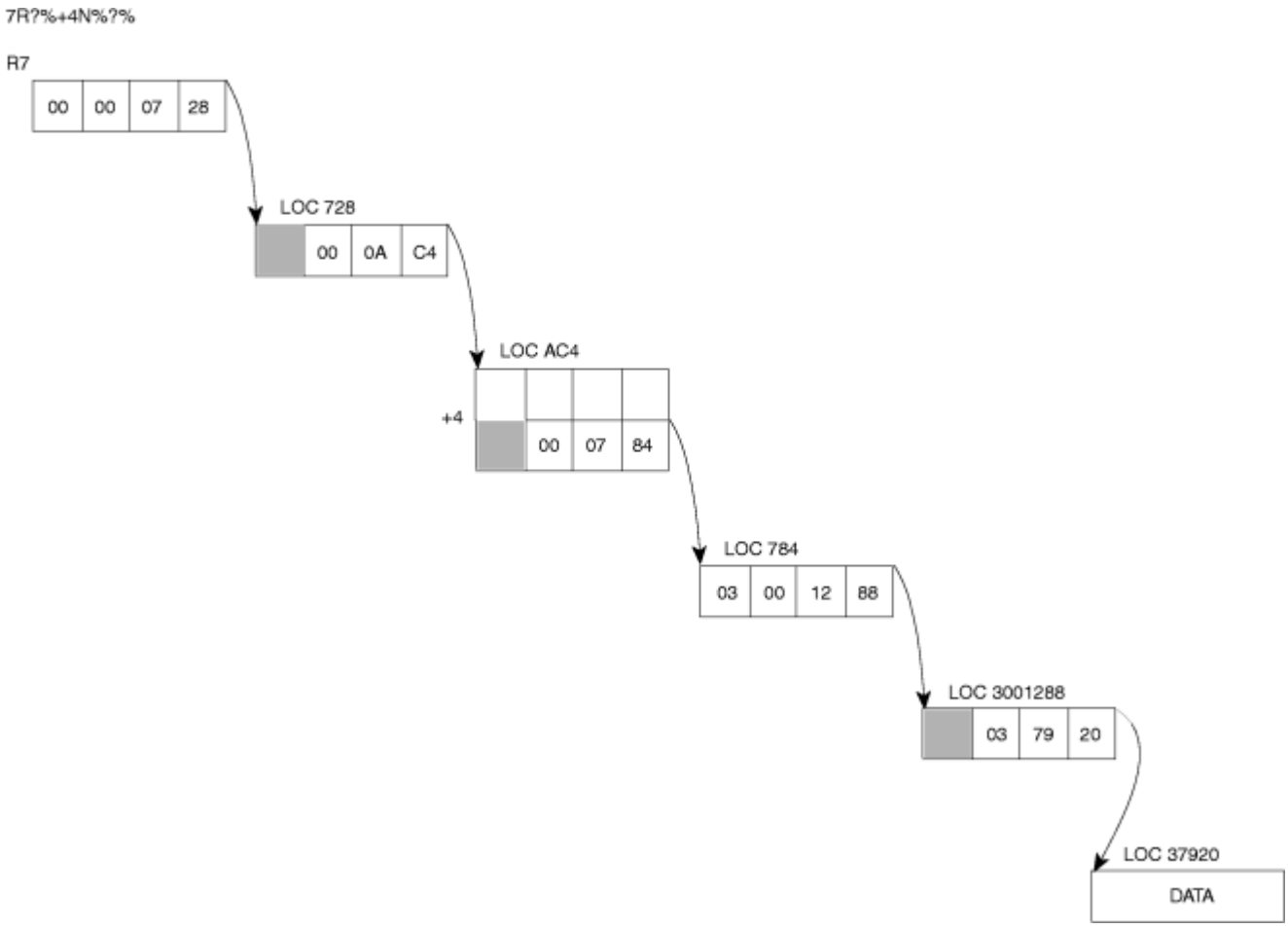


Figure 19. An Address Expression with Mixed Indirection Symbols

PSTRING

A parenthesized string is a string of characters enclosed within a set of parentheses, such as:

```
(string)
```

The string can consist of any combination of characters of any length, with one restriction; if it includes parentheses, they must be balanced. However, the enclosing right parenthesis of a PSTRING can be omitted if the string ends at the end of a logical line.

A null PSTRING is defined as a left parenthesis followed by either a right parenthesis or the end of a logical line. See "Acceptance of double-byte character set data" on page 46 for information about using double-byte character set data in a parenthesized string.

USERID

A user ID consists of an identification optionally followed by a slash and a password. The format is:

```
identification[/password]
```

identification

can be any combination of alphanumeric characters up to seven characters in length, the first of which must be an alphabetic character or one of the special characters $, #, @.

password

can be any combination of alphanumeric characters up to eight characters in length. If delimiters are used, the password must be enclosed in quotation marks. If quotation marks are to be used in

the password, two quotation marks must be entered consecutively. One of them is eliminated by the Parse Service Routine.

Separators can be inserted between the identification and the slash, and between the slash and the password.

If just the identification is entered, the Parse Service Routine does not prompt for a password. If the identification is entered followed by a slash and no password, the Parse Service Routine prompts for a password. The password entered by the terminal user does not print at the terminal. The terminal user can reply to a prompt for password by entering either a password or a null line. If the user enters a null line, the Parse Service Routine builds the PDE and leaves the respective password field zero.

USERID8

A user ID consists of an identification optionally followed by a slash and a password. The format is:

```
identification[/password]
```

identification

can be any combination of alphanumeric characters up to eight characters in length, the first of which must be an alphabetic character or one of the special characters $, #, @.

password

can be any combination of alphanumeric characters up to eight characters in length. If delimiters are used, the password must be enclosed in quotation marks. If quotation marks are to be used in the password, two quotation marks must be entered consecutively. One of them is eliminated by the Parse Service Routine. Special characters within passwords are allowed for USERID8.

Separators can be inserted between the identification and the slash, and between the slash and the password.

If just the identification is entered, the Parse Service Routine does not prompt for a password. If the identification is entered followed by a slash and no password, the Parse Service Routine prompts for a password. The password entered by the terminal user does not print at the terminal. The terminal user can reply to a prompt for password by entering either a password or a null line. If the user enters a null line, the Parse Service Routine builds the PDE and leaves the respective password field zero.

UID2PSWD

A user ID consists of an identification optionally followed by two passwords. The delimiter between the three values is a slash. The format is:

```
identification[/password1[/password2]]
```

identification

can be any combination of alphanumeric characters up to seven characters in length, the first of which must be an alphabetic character or one of the special characters $, #, @.

password1

can be any combination of alphanumeric characters up to eight characters in length. If delimiters are used, the password must be enclosed in quotation marks. If quotation marks are to be used in the password, two quotation marks must be entered consecutively. One of them is eliminated by the Parse Service Routine.

password2

Same as *password1*.

Separators can be inserted between the identification and the slash, and between a slash and any of the passwords.

If just the identification is entered, the parse service routine does not prompt for a password.

If the identification is entered followed by a slash and no password1, the parse service routine prompts for password1. The password1 entered by the terminal user does not print at the terminal.

If password1 is entered followed by a slash and no password2, the parse service routine prompts for password2. The password2 entered by the terminal user does not print at the terminal.

The terminal user can reply to a prompt for a password by entering either a password or a null line. If the user enters a null line, the parse service routine builds the PDE and leaves both password fields zero.

IKJPOSIT generates a variable-length parameter control entry (PCE). Within the PCE, a field contains a hexadecimal number indicating the type of positional operand described by the PCE.

UID82PWD

A user ID consists of an identification optionally followed by two passwords. The delimiter between the three values is a slash. The format is:

```
identification[/password1[/password2]]
```

identification

can be any combination of alphanumeric characters up to eight characters in length, the first of which must be an alphabetic character or one of the special characters $, #, @.

password1

can be any combination of alphanumeric characters up to eight characters in length. If delimiters are used, the password must be enclosed in quotation marks. If quotation marks are to be used in the password, two quotation marks must be entered consecutively. One of them is eliminated by the Parse Service Routine. Special characters within passwords are allowed for UID82PWD.

password2

Same as *password1*.

Separators can be inserted between the identification and the slash, and between a slash and any of the passwords.

If just the identification is entered, the parse service routine does not prompt for a password.

If the identification is entered followed by a slash and no password1, the parse service routine prompts for password1. The password1 entered by the terminal user does not print at the terminal.

If password1 is entered followed by a slash and no password2, the parse service routine prompts for password2. The password2 entered by the terminal user does not print at the terminal.

The terminal user can reply to a prompt for a password by entering either a password or a null line. If the user enters a null line, the parse service routine builds the PDE and leaves both password fields zero.

IKJPOSIT generates a variable-length parameter control entry (PCE). Within the PCE, a field contains a hexadecimal number indicating the type of positional operand described by the PCE. For UID2PSWD, the hexadecimal number is C.

DSNAME

The data set name operand has three possible formats:

```
dsname [ (membername)] [/password]
[dsname] (membername) [password]
'dsname [ (membername)] ' [/password]
```

dsname

May be either a qualified or an unqualified name.

An unqualified name is any combination of alphabetic, numeric (0-9), hyphen ("-"), or special characters ($, #, @) up to eight characters in length, the first of which must be an alphabetic character or one of the special characters $, #, @.

A qualified name is made up of several unqualified names, each unqualified name separated by a period. A qualified name, including the periods, can be up to 44 characters in length.

membername
One to eight alphanumeric characters, the first of which must be an alphabetic character or one of the special characters $, #, @.

The Parse Service Routine considers the entire DSNAME operand missing if the first character scanned is not an apostrophe, an alphabetic character, a special character $, #, @, or a left parenthesis. If the VOLSER option is specified, the first character can be numeric.

If it is numeric, only six characters are accepted for VOLSER. VOLSER is valid only for DSNAME or DSTHING. If USID is specified, the Parse Service Routine will prefix all data set names not entered in quotation marks with the user identification contained in the user profile table (UPT). Note that the user identification is not necessarily the user ID but can be any dsname-prefix specified as parameter with the PROFILE PREFIX command.

If uppercase data set names are required, the ASIS operand should not be specified. Lowercase data set names are allowed as input by PARSE, but converted to uppercase when UPPERCASE is specified, either explicitly or by default. To disallow lowercase data set names as input, a validity check exit must be used.

If the slash and the password are not entered, the parse service routine does not prompt for the password. If the slash is entered and not the password, the Parse Service Routine prompts for the password. This ensures that the terminal user's reply does not print at the terminal.

DSTHING
A DSTHING is a dsname operand as previously defined except that an asterisk can be substituted for an unqualified name or for each qualifier of a qualified name. The parse service routine processes the asterisk as if it were a dsname. The asterisk is used to indicate that all data sets at that particular level are considered.

Note: If the first character of a dsname is an asterisk, the parse service routine will not prefix the USERID.

QSTRING
A quoted string is a string of characters enclosed within apostrophes, such as:

```
'string'
```

The string can consist of a combination of characters, of any length, with one restriction: If the user wants to enter apostrophes within the string, two successive apostrophes must be entered for each single apostrophe wanted. One of the apostrophes is removed by the Parse Service Routine.

The ending apostrophe is not required if the string ends at the end of the logical line.

A null quoted string is defined as two contiguous apostrophes or an apostrophe at the end of the logical line. See “Acceptance of double-byte character set data” on page 46 for information about using double-byte character set data in a quoted string.

SPACE
Space is a special purpose operand; it allows a string operand that directly follows a command name to be entered without a preceding self- defining delimiter character. The space operand must always be followed by a string operand. If the delimiter of the command name is a tab, the tab is the first character of the string. The string always ends at the end of the logical line.

JOBNAME
The jobname can have an optional job identifier. Each job identifier is a maximum of eight alphanumeric characters, the first of which must be an alphabetic character or one of the special characters $, #, @. There is no separator character between the jobname and job identifier. The syntax is `jobname(jobid)`.

CONSTANT
There are several forms of the constant operand.

Fixed-point numeric literal
consists of a string of digits (0 through 9) preceded optionally by a sign (+ or -), such as:

```
+1234.43
```

This literal can contain a decimal point anywhere in the string except as the rightmost character. The total number of digits cannot exceed 18. Embedded blanks are not allowed.

Floating-point numeric literal
takes the following form:

```
+1234.56E+10
```

This literal is a string of digits (0 through 9) preceded optionally by a sign (+ or -) and must contain a decimal point. This is immediately followed by the letter E and then a string of digits (0 through 9) preceded optionally by a sign (+ or -). Embedded blanks are not allowed. The string of digits preceding the letter E cannot be greater than 16 and the string following E cannot be greater than 2.

Non-numeric literal
consists of a string of characters from the EBCDIC character set, excluding the apostrophe, and enclosed in apostrophes, entered as:

```
‘numbers (1234567890) and letters are ok’
```

The length of the string excluding apostrophes can be from 1 to 120 characters in length.

Figurative constant
is one of a set of reserved words supplied by the caller of the Parse Service Routine such as:

```
test123
```

A figurative constant consists of a string of characters up to 255 in length. Embedded blanks are not allowed. All characters of the EBCDIC character set are allowed except the blank, comma, tab, semicolon, and carrier return, however, the first operand must be alphabetic.

See “Acceptance of double-byte character set data” on page 46 for information about using double-byte character set data in a quoted character constant.

VARIABLE
The following is the form of the variable operand.

```
[program_id.]data_name[{OF}qualification]
                      [{IN}             ]
                      [ (subscript)     ]
```

program_id
consists of the first eight characters of a program identifier followed by a period. The first character must be alphabetic (A through Z) and the remaining characters must be alphabetic or numeric (0 through 9).

data_name
consists of a maximum of 30 characters of the following types: alphabetic (A through Z), numeric (0 through 9), and hyphen (-).

An example is:

```
mydataset-123
```

The data-name cannot begin or end with a hyphen and must contain at least one alphabetic character.

```
here55.mydataset-123
```

qualification
is applied by placing one or more data-names (preceded by the qualifiers IN or OF) after a data-name. An example is:

```
mydataset-123 of yourdataset-456
```

The number of qualifiers that can be entered for a data-name is limited to 255.

subscript
consists of a data-name with subscripts enclosed in parentheses following the data-name entered as:

```
yourdataset-456 (mydataset-123)
```

A separator between the data-name and the subscript is optional. Subscripts are a list of constants or variables.

The number of subscripts that can be entered for a data-name is limited to 3, entered as:

```
here55 (abc def h15)
```

A separator character between subscripts is required.

STATEMENT NUMBER
The following is the form of a statement number:

```
[program_id.]line_number[.verb_number]
```

An example is:

```
here.23.7
```

program_id
consists of the first eight characters of a program identifier followed by a period. The first character must be alphabetic (A through Z) and the remaining characters must be alphanumeric (A through Z or 0 through 9).

line_number
consists of a string of digits (0 through 9) and cannot exceed a length of six digits.

verb_number
consists of one digit (0 through 9) that is preceded by a period.

Embedded blanks are not allowed in a statement number.

EXPRESSION
An expression takes the form:

```
(operand1 operator operand2)
```

The operator in the expression shows a relationship between the operands, such as:

```
(abc equals 123)
```

An expression must be enclosed in parentheses. An expression is defined by the IKJOPER macro. The operands are defined by the IKJTERM macro, and the operator is defined by the IKJRSVWD macro instruction.

RESERVED WORD
has three uses depending on the presence of operands on the IKJRSVWD macro instruction. The uses are:

- When used with the RSVWD keyword of the IKJTERM macro instruction, the IKJRSVWD macro identifies the beginning of a list of reserved words, any one of which can be entered as a constant.
- When used with the RSVWD keyword of the IKJOPER macro instruction, the IKJRSVWD macro identifies the beginning of a list of reserved words, any one of which can be an operator in an expression.
- When used by itself, the IKJRSVWD macro instruction defines a positional reserved word operand.

The IKJRSVWD macro instruction is followed by a list of IKJNAME macros that contain all of the possible reserved words used as figurative constants or operators.

HEX
A hexadecimal value is any quantity of the form X'nn', 'ABC' (quoted string), or any non-quoted character string where a separator or delimiter indicates the end. See "Acceptance of double-byte character set data" on page 46 for information about using double-byte character set data in a quoted string of characters.

CHAR
A character string is any data in the form of a quoted or non-quoted string. See "Acceptance of double-byte character set data" on page 46 for information about using double-byte character set data in a quoted string of characters.

INTEG
An integer is a numeric quantity in one of the following forms:

- (X'nn') - where *n* is a valid hexadecimal digit (A-F, 0-9), and there is a maximum of 8 digits.
- (B'mm') - where *m* is a valid binary bit (0-1), and there is a maximum of 32 digits.
- dddddd - where *d* is a decimal digit 0-9, and there is a maximum of 10 digits.

The Parse Service Routine converts an integer operand into its equivalent binary value. The maximum decimal value for INTEG is 2147843647.

Positional operands not dependent on delimiters

A positional operand that is not dependent on delimiters is passed as a character string with restrictions on the beginning character, additional characters, and length. These restrictions are passed to the Parse Service Routine as operands on the IKJIDENT macro instruction.

The Parse Service Routine recognizes the following character types as the beginning character and additional characters of a non-delimiter-dependent positional operand:

ALPHA
indicates an alphabetic character or one of the special characters $, #, @.

NUMERIC
indicates a number 0-9.

ALPHANUM
indicates an alphabetic character, one of the special characters $, #, @, or a number.

ANY
indicates that the character to be expected can be any character other than a blank, comma, tab, semicolon, or carrier return. A right parenthesis must, however, be balanced by a left parenthesis.

NONATABC
indicates only an alphabetic character is accepted; special characters $, #, @ are not accepted.

NONATNUM
indicates numbers and alphabetic characters are accepted; special characters $, #, @ are not accepted.

An asterisk can be entered in place of any positional operand that is not dependent on delimiters.

Entering positional operands as lists of ranges

You might want to have some positional operands of your command entered in the form of a list, a range, or a list of ranges. The macro instructions that describe positional operands to the Parse Service Routine, IKJPOSIT, IKJTERM and IKJIDENT, provide a LIST and a RANGE operand. If coded in the macro instruction, they indicate that the positional operands expected can be in the form of a list or a range.

LIST
indicates to the Parse Service Routine that one or more of the same type of positional operands can be entered enclosed in parentheses as follows:

```
(positional-operand positional-operand …)
```

If one or more of the items contained in the list are to be entered enclosed in parentheses, both the left and the right parenthesis must be included for each of those items.

The following positional operand types can be used in the form of a list: VALUE, ADDRESS, USERID, USERID8, UID2PSWD, UID82PWD, DSNAME, DSTHING, JOBNAME, CONSTANT, STATEMENT NUMBER, VARIABLE, HEX, CHAR, INTEG, and any positional operands that are not dependent upon delimiters.

RANGE
indicates to the Parse Service Routine that two positional operands are to be entered separated by a colon as follows:

```
positional-operand:positional-operand
```

The following positional operand types can be used in the form of a range or a list of ranges: HEX (form X'' only), ADDRESS, VALUE, CONSTANT, STATEMENT NUMBER, VARIABLE, INTEG, and any positional operand that is not dependent upon delimiters.

If the user at the terminal wants to enter an operand that begins with a left parenthesis, and you have specified in either the IKJPOSIT or IKJIDENT macro instruction that the operand can be entered as a list or a range, the user must enclose the operand in an extra set of parentheses to obtain the correct result.

For instance, if you have used the IKJPOSIT macro instruction to specify that the DSNAME operand can be entered as a list, and the terminal user wants to enter a dsname of the form:

```
(membername)/password
```

The user must enter it as:

```
((membername)/password)
```

Keyword operands

Keyword operands can be entered anywhere in the command as long as they follow all positional operands. They can consist of any combination of alphanumeric characters up to 31 characters long, the first of which must be an alphabetic character.

Describe keyword operands to the Parse Service Routine with the IKJKEYWD, IKJUNFLD, IKJNAME, and IKJSUBF macro instructions.

Subfields associated with keyword operands

A keyword operand can have a subfield of operands associated with it. A subfield contains positional and/or keyword operands, and must be enclosed in parentheses directly following its associated keyword operand.

Separators can appear between a keyword operand and the opening parenthesis of its subfield. In addition, separators can appear after the closing parenthesis of a subfield and the following keyword operand. In the following example, posn1 and kywd2 are operands in the subfield of keyword1:

```
keyword1(posn1 kywd2)
```

The same syntax rules that apply to commands apply within keyword subfields.

- Keyword operands must follow positional operands.
- Enclosing right parenthesis can be eliminated if the subfield ends at the end of a logical line.

- The subfield cannot contain unbalanced right parentheses.

If a user enters a keyword with a subfield in which there is a required operand, but does not enter the subfield, the Parse Service Routine prompts for the required operand. The terminal user must not include the subfield parentheses when he enters the required operand.

If a subfield has a positional operand that can be entered as a list, and if this is the only operand in the subfield, the list must be enclosed by the same parentheses that enclose the subfield, such as:

```
keyword(item1 item2 item3)
```

where item1, item2, and item3 are members of a list.

If a subfield has as its first operand a positional operand that can be entered as a list, and there are additional operands in the subfield, a separate set of parentheses is required to enclose the list, such as:

```
keyword((item1 item2 item3) param)
```

where item1, item2, and item3 are members of a list, and param is an operand not included in the list.

Using the parse macro instruction to define command syntax

A Command Processor that uses the Parse Service Routine must build a parameter control list (PCL) to define the syntax of acceptable command or subcommand operands. Each acceptable operand is described by a parameter control entry (PCE) within the PCL. The Parse Service Routine compares the operands within the command buffer against the PCL to determine if valid command or subcommand operands have been entered.

The command processor builds the PCL, and the PCEs within it, using the parse macro instructions. These macro instructions generate the PCL and establish symbolic references for the parameter descriptor list (PDL). The Parse Service Routine returns the PDL to the command processor to describe the results of comparing the operands in the command buffer with the PCL. The PDL is composed of separate entries (PDEs) for each of the command operands found in the command buffer.

Table 15 on page 66 describes the functions of each of the parse macro instructions.

Table 15. The parse macro instructions

Macro instruction	Function
IKJPARM	Begins the PCL and establishes a symbolic reference for the PDL.
IKJPOSIT	Builds a PCE to describe a positional operand that contains delimiters, but not including positional operands described by IKJTERM, IKJOPER, IKJIDENT or IKJRSVWD.
IKJTERM	Builds a PCE for a positional operand that can be a constant, statement number or variable.
IKJOPER	Builds a PCE that describes an expression.
IKJRSVWD	Builds a PCE to describe a reserved word operand. It can also be used with IKJTERM to describe a reserved word constant, or with IKJOPER to describe the operator portion of an expression.
IKJIDENT	Builds a PCE that describes a positional operand that does not depend upon a particular delimiter.
IKJKEYWD	Builds a PCE that describes a keyword operand.
IKJNAME	Builds a PCE that describes the possible names that can be entered for a keyword or reserved word operand.

Table 15. The parse macro instructions (continued)

Macro instruction	Function
IKJSUBF	Builds a PCE that indicates the beginning of a keyword subfield description.
IKJUNFLD	Builds a PCE to indicate that unidentified keyword operands can be encountered and specifies the address of a verify exit routine to be given control.
IKJENDP	Indicates the end of the PCL.
IKJRLSA	Releases any virtual storage allocated by the parse service routine for the PDL that remains after parse returns control to its caller.

These macro instructions perform the following additional functions:

- The IKJPOSIT, IKJTERM, IKJOPER, IKJRSVWD, IKJIDENT, IKJKEYWD, IKJNAME, and IKJSUBF macro instructions describe the positional and keyword operands valid for the command processor. The command processor uses the label fields of these macro instructions to reference fields within the DSECT that maps the PDL returned by the Parse Service Routine.

The macros that generate input to parse can be issued by a program that is loaded above 16 MB in virtual storage. The IKJRLSA macro can be issued in either 24-or 31-bit addressing mode. If the PCL resides above 16 MB in virtual storage, you should not attempt to update it in a validity checking routine after the PCL has been passed to parse. However, if the PCL resides below 16 MB, you can update the PCL in a validity checking routine after passing it to parse.

Using IKJPARM to begin the PCL and the PDL

Use the IKJPARM macro instruction to begin the parameter control list (PCL) and to provide a symbolic address for the beginning of the parameter descriptor list (PDL) returned by the Parse Service Routine. The PCL is constructed in the CSECT named by the label field of the macro instruction; the PDL is mapped by the DSECT named in the DSECT operand of the macro instruction.

Figure 20 on page 67 shows the format of the IKJPARM macro instruction. Each of the operands is explained following the figure.

```
label     IKJPARM      DSECT={dsect_name }
                              { IKJPARMD  }
```

Figure 20. The IKJPARM macro instruction

label
The name you provide is used as the name of the CSECT in which the PCL is constructed.

DSECT=*dsect_name* | IKJPARMD
provides a name for the DSECT created to map the parameter descriptor list. This can be any name; the default is IKJPARMD.

The parameter control entry built by IKJPARM

The IKJPARM macro instruction generates the parameter control entry (PCE) shown in Table 16 on page 68. This PCE begins the parameter control list.

Table 16. The parameter control entry built by IKJPARM

Number of bytes	Field name	Contents or meaning
2		Length of the parameter control list. This field contains a hexadecimal number representing the number of bytes in this PCL. The maximum allowable value is X'7FFF'. Specifying a PCL greater than X'7FFF' will produce unpredictable results.
2		Length of the parameter descriptor list. This field contains a hexadecimal number representing the number of bytes in the parameter descriptor list returned by the Parse Service Routine.
2		This field contains a hexadecimal number representing the offset within the PCL to the first IKJKEYWD PCE or to an end-of-field indicator if there are no keywords. An end-of-field indicator can be either an IKJSUBF or an IKJENDP PCE.

Using IKJPOSIT to describe a delimiter-dependent positional operand

Use the IKJPOSIT macro instruction to describe the following delimiter-dependent positional operands:

```
SPACE          DELIMETER      STRING         QSTRING        UID82PWD
ADDRESS        PSTRING        USERID         VALUE          JOBNAME
DSNAME         DSTHING        USERID8        UID2PSWD
```

Use the IKJIDENT macro instruction to describe the other delimiter-dependent positional operands.

The order in which you code the macros for positional operands is the order in which the Parse Service Routine expects to find the positional operands in the command string.

Figure 21 on page 68 shows the format of the IKJPOSIT macro instruction. Each of the operands is explained following the figure.

```
label     IKJPOSIT         SPACE
                         {   DELIMITER             }
                         {   STRING                }
                         {   VALUE                 }
                         {   ADDRESS  [,EXTENDED] }   [,LIST][,RANGE]
                         {            [,VECTOR]   }
                         {            [,AR]       }
                         {   PSTRING               }
                         {   USERID                }
                         {   USERID8,              }
                         {   UID2PSWD              }
                         {   UID82PWD,             }
                         {   DSNAME                }   [,VOLSER][,DDNAM][,USID]
                         {   DSTHING               }
                         {   QSTRING               }
                         {   JOBNAME               }
                           [,SQSTRING]
                          [,UPPERCASE ,PROMPT='prompt data'    ]
                          [,ASIS      ,DEFAULT='default value' ]
                           [,HELP=('help data','help data',…)]
                           [,VALIDCK=symbolic-address]
```

Figure 21. The IKJPOSIT macro instruction

label
This name is used as the symbolic address within the PDL DSECT of the parameter descriptor entry (PDE) for the operand described by this IKJPOSIT macro instruction.

SPACE through JOBNAME
specifies the type of delimiter-dependent positional operand. The positional operand types are described in detail in “Delimiter-dependent operands” on page 51.

Positional operand type	Where described
SPACE	SPACE of “Delimiter-dependent operands” on page 51
DELIMITER	DELIMETER of “Delimiter-dependent operands” on page 51
STRING	STRING of “Delimiter-dependent operands” on page 51
VALUE	VALUE of “Delimiter-dependent operands” on page 51
ADDRESS	ADDRESS of “Delimiter-dependent operands” on page 51
PSTRING	PSTRING of “Delimiter-dependent operands” on page 51
USERID	USERID of “Delimiter-dependent operands” on page 51
USERID8	USERID8 of “Delimiter-dependent operands” on page 51
UID2PSWD	UID2PSWD of “Delimiter-dependent operands” on page 51
UID82PWD	UID82PWD of “Delimiter-dependent operands” on page 51
DSNAME	DSNAME of “Delimiter-dependent operands” on page 51
DSTHING	DSTHING of “Delimiter-dependent operands” on page 51
QSTRING	QSTRING of “Delimiter-dependent operands” on page 51
JOBNAME	JOBNAME of “Delimiter-dependent operands” on page 51

SQSTRING
The command operand is processed either as a string or as a quoted string. If the delimiter is an apostrophe, the command operand is processed as a quoted string. If the delimiter is any of the other acceptable delimiter characters, the command operand is processed as a string. The SQSTRING option can only be specified if STRING is specified for the operand type.

For example, if SQSTRING is coded in the IKJPOSIT macro instruction, a terminal user entering a command could specify either:

```
/string/string…
```

or

```
'string' 'string' …
```

EXTENDED
specifies that the user can enter 31-bit addresses. This operand is valid only with ADDRESS. For more information, refer to the descriptions of absolute, relative, and indirect addresses and address expressions under the description of the address operand (see “Delimiter-dependent operands” on page 51).

VECTOR
specifies that the user can enter:

- Vector registers as addresses
- Vector mask registers as addresses

- 31-bit addresses.

This operand is valid only when ADDRESS is specified as the operand type. For more information, see the discussion of vector addresses under the description of the address operand (see "Delimiter-dependent operands" on page 51).

AR
specifies that the user can enter:

- Access registers as addresses
- Vector registers as addresses
- Vector mask registers as addresses
- 31-bit addresses.

This operand is valid only when ADDRESS is specified as the operand type. For more information, see the description of the address operand (see "Delimiter-dependent operands" on page 51.

LIST
The command operands can be entered by the terminal user as a list:

```
commandname (operand,operand, …)
```

This list option can be used with the following delimiter-dependent positional operands: USERID, USERID8, DSNAME, DSTHING, ADDRESS, VALUE, JOBNAME, and PSTRING (within a subfield only).

RANGE
The command operands can be entered by the terminal user as a range:

```
commandname operand:operand
```

The range option can be used with the following delimiter-dependent positional operands: ADDRESS, VALUE.

VOLSER
specifies that a data set name is to be a volume serial name. This operand is valid only with DSNAME or DSTHING. If the first character is numeric, a maximum of six characters are allowed.

DDNAM
specifies a data definition name. This option causes an `INVALID DDNAME` message if the name is not valid.

USID
specifies that the user identification is to prefix all data set names that either are not entered in quotation marks or start with an asterisk. If you specify USID and DSTHING and the first character of a data set name is an asterisk (*), the parse service routine does not prefix the user identification. Note that the user identification is not necessarily the user ID but can be any dsname-prefix specified as parameter with the PROFILE PREFIX command.

The following options (UPPERCASE, ASIS, PROMPT, DEFAULT, HELP, and VALIDCK) can be used with all delimiter-dependent positional operands except SPACE and DELIMITER.

UPPERCASE
The operand is to be translated to uppercase.

ASIS
The operand is to be left as it was entered by the terminal user.

PROMPT='*prompt data*'
The operand described by this IKJPOSIT macro instruction is required; the prompting data is the message to be issued if the operand is not entered by the terminal user. If prompting is necessary and the terminal is in prompt mode, the Parse Service Routine supplies a message-identifying number (message ID) and adds the word ENTER to the beginning of this message before writing it to the terminal. If prompting is necessary but the terminal is in no-prompt mode, the Parse Service Routine supplies a message ID and adds the word MISSING to the beginning of this message before writing it to the terminal.

DEFAULT=‘*default value*’
The operand described by this IKJPOSIT macro instruction is required, but the terminal user need not enter it. If the operand is not entered, the value specified as the default value is used.

Note: If neither PROMPT nor DEFAULT is specified, the operand is optional. The Parse Service Routine takes no action if the operand specified by this IKJPOSIT macro instruction is not present in the command buffer.

HELP=(‘*help data*’,‘*help data*’,...)
You can provide up to 255 second-level messages. (Note, however, that the assembler in use can limit the number of characters that a macro operand with a sublist can contain.) Enclose each message in apostrophes and separate the messages by single commas. These messages are issued one at a time after each question mark is entered by the terminal user in response to a prompting message from the parse service routine. These messages are not sent to the user when the prompt is for a password on a DSNAME or USERID operand.

Parse adds a message ID and the word ENTER (in prompt mode) or MISSING (in no-prompt mode) to the beginning of each message before writing it to the terminal.

VALIDCK=*symbolic-address*
Supply the symbolic address of a validity checking routine if you want to perform additional validity checking on this operand. Parse calls this routine after first determining that the operand is syntactically correct.

The parameter control entry built by IKJPOSIT

The IKJPOSIT macro instruction generates the variable-length parameter control entry (PCE) shown in Table 17 on page 72.

Table 17. The parameter control entry built by IKJPOSIT

Number of bytes	Field name	Contents or meaning
2		Flags. These flags are set to indicate which options were specified in the IKJPOSIT macro instruction. Byte 1: **001. ....** This is an IKJPOSIT PCE. **...1 ....** PROMPT **.... 1...** DEFAULT **.... .1..** This is an extended format PCE. If the VALIDCK parameter was specified, the length of the field containing the address of the validity checking routine is four bytes. **.... ..1.** HELP **.... ...1** VALIDCK Byte 2: **1... ....** LIST **.1.. ....** ASIS **..1. ....** RANGE **...1 ....** MEMNAME **.... 1...** SQSTRING **.... .1..** USID **.... ..1.** VOLSER **.... ...1** DDNAME
2		Length of the parameter control entry. This field contains a hexadecimal number representing the number of bytes in this IKJPOSIT PCE.
2		Contains a hexadecimal offset from the beginning of the parameter descriptor list to the related parameter descriptor entry built by the Parse Service Routine.

Table 17. The parameter control entry built by IKJPOSIT (continued)

Number of bytes	Field name	Contents or meaning
1		This field contains a hexadecimal number indicating the type of positional operand described by this PCE. These numbers have the following meaning: **X'1'** DELIMITER **X'2'** STRING **X'3'** VALUE **X'4'** ADDRESS **X'5'** PSTRING **X'6'** USERID **X'7'** DSNAME **X'8'** DSTHING **X'9'** QSTRING **X'A'** SPACE **X'B'** JOBNAME **X'C'** UID2PSWD **X'D'** EXTENDED ADDRESS **X'E'** VECTOR ADDRESS **X'F'** AR ADDRESS **X'10'** USERID8 **X'11'** UID82PWD
1		Contains the length minus one of the default or prompting information supplied on the IKJPOSIT macro instruction. This field and the next field are present only if DEFAULT or PROMPT was specified on the IKJPOSIT macro instruction.
Variable		This field contains the prompting or default information supplied on the IKJPOSIT macro instruction.
2		This field contains a hexadecimal figure representing the length in bytes of all the PCE fields used for second-level messages. The figure includes the length of this field. The fields are present only if HELP is specified on the IKJPOSIT macro instruction.
1		This field contains a hexadecimal number representing the number of second-level messages specified by HELP on this IKJPOSIT PCE.
2		This field contains a hexadecimal number representing the length of this HELP segment. The length figure includes the length of this field, the message segment offset field, and the length of the information. These fields are repeated for each second-level message specified by HELP on the IKJPOSIT macro instruction.
2		This field contains the message segment offset. It is set to X'0000'.
Variable		This field contains one second-level message supplied on the IKJPOSIT macro instruction specified by HELP. This field and the two preceding ones are repeated for each second-level message supplied on the IKJPOSIT macro instruction. These fields do not appear if second-level message data was not supplied.

Table 17. The parameter control entry built by IKJPOSIT (continued)

Number of bytes	Field name	Contents or meaning
3 or 4		This field contains the address of a validity checking routine if VALIDCK was specified on the IKJPOSIT macro. If the "extended format PCE" bit is on in the IKJPOSIT PCE, the address is four bytes long; if the bit is off, the address is three bytes long. This field is not present if VALIDCK was not specified.

Using IKJTERM to describe a delimiter-dependent positional operand

Use the IKJTERM macro instruction to describe a positional operand that is one of the following:

- Statement number
- Constant
- Variable
- Constant or variable

The order in which you code the macros for positional operands is the order in which the Parse Service Routine expects to find the operands in the command string.

Figure 22 on page 74 shows the format of the IKJTERM macro instruction. Each of the operands is explained following the figure.

```
label    IKJTERM    'parameter-type'[,LIST][,RANGE]
                     [,UPPERCASE ]  [      {STMT }]
                     [,ASIS      ] ,[TYPE= {CNST }]
                                    [      {VAR  }]
                                    [      {ANY  }]
                    [,SBSCRPT[=label-PCE]] ,[PROMPT='prompt data'    ]
                                           ,[DEFAULT='default value' ]
                    [,HELP=('help data','help data',…)]
                    [,VALIDCK=symbolic-address][,RSVWD=label-PCE]
```

Figure 22. The IKJTERM macro instruction

label
This name is used to address the PCE built by the IKJTERM macro. The hexadecimal offset to the parameter descriptor entry (PDE) built by the Parse Service Routine for this operand is contained in the PCE.

Note: The hexadecimal offset to the PDE will contain binary zero when the IKJTERM macro is used to describe a subscript of a data name.

'*parameter-type*'
This field is required so that the operand can be identified when an error message is necessary. This field differs from the PROMPT field in that the PROMPT field is not required and, if supplied, is used only for a required operand that is not entered by the terminal user. Blanks within the apostrophes are allowed.

LIST
The command operands can be entered by the terminal user as a list, in the form:

```
commandname (operand,operand,…)
```

The LIST option can be used with any of the TYPE= positional operands.

RANGE
The command operands can be entered by the terminal user as a range, in the form:

```
commandname operand:operand
```

The RANGE option can be used with any of the TYPE= positional operands.

Note: The LIST and RANGE options cannot be used when the IKJTERM macro instruction is used to describe a subscript of a data-name.

UPPERCASE
The operand is to be translated to uppercase.

ASIS
The operand is to be left as it was entered by the terminal user.

TYPE=STMT | CNST | VAR | ANY
describes the type of the operand as one of the following:

STMT
Statement number

CNST
Constant

VAR
Variable

ANY
Constant or variable

See "Delimiter-dependent operands" on page 51 for a syntactical definition of these operands.

SBSCRPT[=*label-PCE*]
specifies one of two conditions:

1. If you specify SBSCRPT with a label-PCE, then the data-name described by the IKJTERM macro can be subscripted. Supply the name of the label of an IKJTERM macro instruction that describes the subscript. Only TYPE=VAR or TYPE=ANY operands can be subscripted.
2. If you specify SBSCRPT without a label-PCE, then the IKJTERM macro describes the subscript of a data-name. All TYPE= parameters can be used on a subscript except TYPE=STMT. The LIST and RANGE options cannot be used on an IKJTERM macro that describes a subscript.

Note: You must use two IKJTERM macro instructions to describe a subscripted data-name. The first IKJTERM macro describes the data name and specifies the SBSCRPT option with the label of the second IKJTERM macro. The second IKJTERM macro describes the subscript of the data-name and specifies SBSCRPT without a label-PCE. The second macro instruction must immediately follow the first.

PROMPT='*prompt data*'
The operand described by this IKJTERM macro instruction is required. The prompting data that you specify is issued as a message if the operand is not entered by the terminal user. If prompting is necessary and the terminal is in prompt mode, the Parse Service Routine adds a message-identifying number (message ID) and the word ENTER to the beginning of the message before writing it to the terminal.

If prompting is necessary but the terminal is in no-prompt mode, the Parse Service Routine adds a message ID and the word MISSING to the beginning of the message before writing it to the terminal. If a subscripted data-name requires prompting, the terminal user is prompted for the entire name including the subscript.

DEFAULT='*default value*'
The operand described by this IKJTERM macro instruction is required, but the terminal user need not enter it. If the operand is not entered, the value specified as the default value is used.

Note: If neither PROMPT nor DEFAULT is specified, the operand is optional. The Parse Service Routine takes no action if the operand is not present.

HELP=('*help data*','*help data*',...)
You can provide up to 255 second-level messages. (Note, however, that the assembler in use can limit the number of characters that a macro operand with a sublist can contain.) Enclose each message in apostrophes and separate the messages by single commas. These messages are issued one at a time

after each question mark entered by the terminal user in response to a prompting message from the Parse Service Routine.

Parse adds a message ID and the word ENTER (in prompt mode) or MISSING (in no-prompt mode) to the beginning of each message before writing it to the terminal.

VALIDCK=*symbolic-address*
Supply the symbolic address of a validity checking routine if you want to perform additional checking on this operand. Parse calls this routine after first determining that the operand is syntactically correct.

RSVWD=*label-PCE*
Use this option when TYPE=CNST or TYPE=ANY is specified to indicate that this operand can be a figurative constant. Supply the address of the PCE (label on a IKJRSVWD macro instruction) that begins the list of reserved words that can be entered as a figurative constant.

This list of reserved words is defined by a series of IKJNAME macros that contain all possible names and immediately follow the IKJRSVWD macro.

Note: The IKJRSVWD macro can be coded anywhere in the list of macros that build the PCL except following an IKJSUBF macro instruction. This permits other IKJTERM macro instructions to refer to the same list.

The parameter control entry built by IKJTERM

The IKJTERM macro instruction generates the variable parameter control entry (PCE) shown in Table 18 on page 76.

Table 18. The parameter control entry built by IKJTERM

Number of bytes	Field name	Contents or meaning
2		Flags. These flags are set to indicate options on the IKJTERM macro instruction. Byte 1: **110. ....** This is an IKJTERM PCE. **...1 ....** PROMPT **.... 1...** DEFAULT **.... .1..** This is an extended format PCE. If the VALIDCK parameter was specified, the length of the field containing the address of the validity checking routine is four bytes. **.... ..1.** HELP **.... ...1** VALIDCK Byte 2: **1... ....** LIST **.1.. ....** ASIS **..1. ....** RANGE **...1 ....** This term can be SUBSCRIPTED. **.... 1...** A reserved word PCE is chained from this term. **.... .000** Reserved
2		The hexadecimal length of this PCE.

Table 18. The parameter control entry built by IKJTERM (continued)

Number of bytes	Field name	Contents or meaning
2		Contains a hexadecimal offset from the beginning of the parameter descriptor list to the parameter descriptor entry built by the parse routine.
1		This field indicates the type of positional parameter described by this PCE. **1... ....** STATEMENT NUMBER **.1.. ....** VARIABLE **..1. ....** CONSTANT **...1 ....** ANY (constant or variable) **.... 1...** This term is a SUBSCRIPT term. **.... .000** Reserved
4		Byte 1-2 contain the hexadecimal length of the parameter-type field. Byte 3-4 contain the offset of the parameter-type field. It is set to X'0012'.
Variable		Contains the parameter-type field.
1		Contains the length of the default or prompting information supplied on the macro instruction.
Variable		Contains the default or prompting information supplied on the macro instruction.
2		If a subscript is specified on the macro, this field contains the offset into the parameter control list of the subscript PCE.
2		If a reserved word PCE is specified on the macro, this field contains the offset into the parameter control list of the reserved word PCE.
2		Contains the length (including this field) of all the PCE fields used for second-level messages if HELP is specified on the macro.
1		The number of second-level messages specified on the macro instruction by the HELP parameter.
2		Contains the length of this segment including this field, the message offset field and second-level message. **Note:** This field and the following two are repeated for each second-level message specified by HELP on the macro.
2		This field contains the message segment offset.
Variable		This field contains one second-level message specified by HELP on the macro instruction. This field and the two preceding fields are repeated for each second-level message specified.
3 or 4		This field contains the address of a validity checking routine if VALIDCK was specified on the IKJTERM macro. If the "extended format PCE" bit is on in the IKJTERM PCE, the address is four bytes long; if the bit is off, the address is three bytes long. This field is not present if VALIDCK was not specified.

Using IKJOPER to describe a delimiter-dependent positional operand

Use the IKJOPER macro instruction to provide a parameter control entry (PCE) that describes an expression. An expression consists of three parts; two operands and one operator in the form:

```
(operand1 operator operand2)
```

typically entered as:

```
(abc eq 123)
```

The parts of an expression are described by PCEs that are chained to the IKJOPER PCE. Use the IKJTERM macro instruction to identify the operands, and use the IKJRSVWD macro instruction to identify the operator.

Figure 23 on page 78 shows the format of the IKJOPER macro instruction. Each of the operands is explained following the figure.

```
label     IKJOPER      'parameter-type'[,PROMPT='prompt data'    ]
                                       [,DEFAULT='default value' ]
                       [,HELP=('help data','help data',…)]
                       [,VALIDCK=symbolic-address],OPERND1=label1
                       ,OPERND2=label2,RSVWD=label3
                       [,CHAIN=label4]
```

Figure 23. The IKJOPER macro instruction

label
This name is used to address the PCE built by the IKJOPER macro. The hexadecimal offset to the parameter descriptor entry built by the Parse Service Routine for this operand is contained in the PCE.

'*parameter-type*'
This field is required so that the operand can be identified when an error message is necessary. This field differs from the PROMPT field in that the PROMPT field is not required and if supplied is used only for a required operand that is not entered by the terminal user. Blanks within the apostrophes are allowed.

Note: Parse uses this field only for error messages for the complete expression. The IKJTERM and IKJRSVWD PCEs are used when parse issues error messages for missing operands or a missing operator. If a validity check routine indicates that the expression is not valid, parse prompts for the entire expression.

PROMPT='*prompt data*'
The operand described by this IKJOPER macro instruction is required. The prompting data that you specify is issued as a message if the operand is not entered by the terminal user. If prompting is necessary and the terminal is in prompt mode, the Parse Service Routine adds a message- identifying number (message ID) and the word ENTER to the beginning of the message before writing it to the terminal. If prompting is necessary but the terminal is in no-prompt mode, the Parse Service Routine adds a message ID and the word MISSING to the beginning of the message before writing it to the terminal.

DEFAULT='*default value*'
The operand described by this IKJOPER macro instruction is required, but the terminal user need not enter it. If the operand is not entered, the Parse Service Routine uses the value specified as the default value.

Note: If neither PROMPT nor DEFAULT is specified, the operand is optional. The Parse Service Routine takes no action if the operand is not present.

HELP=('*help data*','*help data*',...)
You can provide up to 255 second-level messages. (Note, however, that the assembler in use can limit the number of characters that a macro operand with a sublist can contain.) Enclose each message in apostrophes and separate the messages by single commas. These messages are issued one at a time after each question mark entered by the terminal user in response to a prompting message from the Parse Service Routine.

Parse adds a message ID and the word ENTER (in prompt mode) or MISSING (in no-prompt mode) to the beginning of each message before writing it to the terminal.

VALIDCK=*symbolic-address*
Supply the symbolic address of a validity checking routine if you want to perform additional checking on this expression. The Parse Service Routine calls this routine after first determining that the expression is syntactically correct.

OPERND1=*label1*
Supply the name of the label field of the IKJTERM macro instruction that is used to describe the first operand in the expression. This IKJTERM macro instruction should be coded immediately following the IKJOPER macro instruction that describes the expression.

OPERND2=*label2*
Supply the name of the label field of the IKJTERM macro instruction that is used to describe the second operand in the expression. This IKJTERM macro instruction should be coded immediately following the IKJNAME macro instructions that describe the operator in the expression under the associated IKJRSVWD macro instruction.

RSVWD=*label3*
Supply the name of the label field of the IKJRSVWD macro instruction that begins the list of reserved words that are used to describe the possible operators to be entered for the expression. The IKJRSVWD and associated IKJNAME macro instructions should be coded immediately following the IKJTERM macro that describes the first operand, and immediately preceding the IKJTERM macro that describes the second operand.

CHAIN=*label4*
indicates that this operand described by the IKJOPER macro instruction can be entered as an expression or as a variable. Supply the name of the label field of an IKJTERM macro instruction that describes the variable term. The LIST and RANGE options are not permitted on this IKJTERM macro instruction. Code this IKJTERM macro instruction immediately following the IKJTERM macro that describes the second operand.

Note: The Parse Service Routine first determines if the operand is entered as an expression. If the operand is an expression, that is, enclosed in parentheses, then it is processed as an expression. If it is not an expression, then it is processed using the chained IKJTERM PCE to control the scan of the operand.

The parameter control entry built by IKJOPER

The IKJOPER macro instruction generates the variable parameter control entry (PCE) shown in Table 19 on page 80.

Table 19. The parameter control entry built by IKJOPER

Number of bytes	Field name	Contents or meaning
2		Flags. These flags are set to indicate options on the IKJOPER macro instruction. Byte 1: **111. ....** This is an IKJOPER PCE. **...1 ....** PROMPT **.... 1...** DEFAULT **.... .1..** This is an extended format PCE. If the VALIDCK parameter is specified, the length of the field containing the address of the validity checking routine is four bytes. **.... ..1.** HELP **.... ...1** VALIDCK Byte 2: **0000 0000** Reserved
2		The hexadecimal length of this PCE.
2		Contains a hexadecimal offset from the beginning of the parameter descriptor list to the parameter descriptor entry built by the parse service routine.
4		Byte 1-2 contain the hexadecimal length of the parameter-type field. Byte 3-4 contain the offset of the parameter-type field (X'0012').
Variable		Contains the parameter-type field.
2		If a reserved word PCE is specified on the macro, this field contains the offset into the parameter control list of the reserved word PCE.
2		Contains the offset into the parameter control list of the OPERND1 PCE.
2		Contains the offset into the parameter control list of the OPERND2 PCE.
2		Contains the offset into the parameter control list of the chained term PCE if present. Zero if not present.
1		Contains the length of the default or prompting information supplied on the macro instruction.
Variable		Contains the default or prompting information supplied on the macro instruction.
2		Contains the length (including this field) of all the PCE fields used for second-level messages if HELP is specified on the macro.

Table 19. The parameter control entry built by IKJOPER (continued)

Number of bytes	Field name	Contents or meaning
1		The number of second-level messages specified on the macro instruction by the HELP= parameter.
2		Contains the length of this segment including this field, the message offset field and second-level message. **Note:** This field and the following two are repeated for each second-level message specified by HELP on the macro.
2		This field contains the message segment offset.
Variable		This field contains one second-level message specified by HELP on the macro instruction. This field and the two preceding fields are repeated for each second-level message specified.
3 or 4		This field contains the address of a validity checking routine if VALIDCK was specified on the IKJOPER macro. If the "extended format PCE" bit is on in the IKJOPER PCE, the address is four bytes long; if the bit is off, the address is three bytes long. This field is not present if VALIDCK was not specified.

Using IKJRSVWD to describe a delimiter-dependent positional parameter

Use the IKJRSVWD macro instruction to do the following:

- Define a positional reserved word operand.

  In this case, use the IKJRSVWD macro instruction by itself and specify at least the ‘parameter-type’ operand.
- Describe the operator portion of an expression.

  In this case, use the RSVWD operand of the IKJOPER macro instruction to define the beginning of a list of the possible reserved words that can be an operator in an expression. To identify the possible reserved words that can be operators in an expression, specify a list of IKJNAME macro instructions that immediately follow the IKJRSVWD macro instruction.

  You must specify at least the ‘parameter-type’ operand on the IKJRSVWD macro instruction.
- Describe a reserved word constant.

  In this case, use the RSVWD keyword of the IKJTERM macro instruction to define the beginning of a list of possible reserved words that can be used as a figurative constant. To define the possible figurative constants, specify a list of IKJNAME macros that immediately follow the IKJRSVWD macro instruction.

  When you use the IKJRSVWD macro instruction to define a reserved word constant, code the macro without any operands as follows:

```
label     IKJRSVWD
```

The order in which you code the macros for positional operands is the order in which the Parse Service Routine expects to find the operands in the command string.

Figure 24 on page 82 shows the format of the IKJRSVWD macro instruction. Each of the operands is explained following the figure.

```
label     IKJRSVWD   'parameter-type' [,PROMPT='prompt data'   ]
                                      [,DEFAULT='default value']
                     [,HELP=('help data','help data',…)]
```

Figure 24. The IKJRSVWD macro instruction

label
This name is used to address the PCE built by the IKJRSVWD macro. The hexadecimal offset to the parameter descriptor entry (PDE) built by the Parse Service Routine for this operand is contained in the PCE.

Code the following operands on the IKJRSVWD macro when you use it either by itself to describe a positional reserved word operand, or with IKJOPER to describe the operator portion of an expression.

'*parameter-type*'
This field is required so that the operand can be identified when an error message is necessary. This field differs from the PROMPT field in that the PROMPT field is not required and if supplied is used only for a required operand that is not entered by the terminal user. Blanks within the apostrophes are allowed.

PROMPT='*prompt data*'
The operand described by this IKJRSVWD macro instruction is required. The prompting data that you specify is issued as a message if the operand is not entered by the terminal user. If prompting is necessary and the terminal is in prompt mode, parse adds a message-identifying number (message ID) and the word ENTER to the beginning of the message before writing it to the terminal. If prompting is necessary but the terminal is in no-prompt mode, parse adds a message ID and the word MISSING to the beginning of the message before writing it to the terminal.

DEFAULT='*default value*'
The operand described by this IKJRSVWD macro instruction is required, but the terminal user need not enter it. If the operand is not entered, the value specified as the default value is used.

Note: If neither PROMPT nor DEFAULT is specified, the operand is optional. The Parse Service Routine takes no action if the operand is not present.

HELP=('*help data*','*help data*',...)
You can provide up to 255 second-level messages. (Note, however, that the assembler in use can limit the number of characters that a macro operand with a sublist can contain.) Enclose each message in apostrophes and separate the messages by single commas. These messages are issued one at a time after each question mark entered by the terminal user in response to a prompting message from the parse routine.

The Parse Service Routine adds a message ID and the word ENTER (in prompt mode) or MISSING (in no-prompt mode) to the beginning of each message before writing it to the terminal.

The parameter control entry built by IKJRSVWD

The IKJRSVWD macro instruction generates the variable parameter control entry (PCE) shown in Table 20 on page 83.

Table 20. The parameter control entry built by IKJRSVWD

Number of bytes	Field name	Contents or meaning
2		Flags. These flags are set to indicate options on the IKJRSVWD macro instruction. Byte 1: **101. ....** This is an IKJRSVWD PCE. **...1 ....** PROMPT **.... 1...** DEFAULT **.... .0..** Reserved **.... ..1.** HELP **.... ...0** Reserved Byte 2: **1... ....** This PCE is used with the IKJTERM macro as a figurative constant. **0... ....** This PCE is not used with the IKJTERM macro as a figurative constant. **.000 0000** Reserved.
2		The hexadecimal length of this PCE.
2		Contains a hexadecimal offset from the beginning of the parameter descriptor list to the parameter descriptor entry built by the parse service routine. **Note:** The following fields are omitted if this PCE is used with the IKJTERM macro to describe a figurative constant.
4		Byte 1-2 contain the hexadecimal length of the parameter-type field. Byte 3-4 contain the offset of the parameter-type field (X'0012').
Variable		Contains the parameter-type field.
1		Contains the length of the default or prompting information supplied on the macro instruction.
Variable		Contains the default or prompting information supplied on the macro instruction.
2		Contains the length (including this field) of all the PCE fields used for second-level messages if HELP is specified on the macro.
1		The number of second-level messages specified on the macro instruction by the HELP= parameter.
2		Contains the length of this segment including this field, the message offset field and second-level message. **Note:** This field and the following two are repeated for each second-level message specified by HELP on the macro.
2		This field contains the message segment offset.
Variable		This field contains one second-level message specified by HELP on the macro instruction. This field and the two preceding fields are repeated for each second-level message specified.

Using IKJIDENT to describe a non-delimiter-dependent positional operand

Use the IKJIDENT macro instruction to describe a positional operand that does not depend upon a particular delimiter for its syntactical definition. These operands are discussed in “Positional operands not dependent on delimiters” on page 64.

These positional operands must be in the form of a character string, with restrictions on the beginning character, additional characters, and length, decimal integers, or hexadecimal characters.

The order in which you code the macro instructions for positional operands is the order in which the Parse Service Routine expects to find the positional operands in the command string.

Figure 25 on page 84 shows the format of the IKJIDENT macro instruction. Each of the operands is explained following the figure.

```
label     IKJIDENT      'parameter-type' [,LIST][,RANGE][,PTBYPS]
                        [,ASTERISK][,UPPERCASE] [,MAXLNTH=number]
                                   [,ASIS    ]
                       [         { ALPHA   } ]  [         {ALPHA    }]
                       [         { NUMERIC } ]  [         {NUMERIC  }]
                       [ ,FIRST={ ALPHANUM} ]  [,OTHER= {ALPHANUM }]
                       [         { ANY     } ]  [         {ANY      }]
                       [         { NONATABC} ]  [         {NONATABC }]
                       [         { NONATNUM} ]  [         {NONATNUM }]
                       [ ,PROMPT='prompt data'    ]
                       [ ,DEFAULT='default value' ]
                       [ ,CHAR ]
                       [ ,INTEG]
                       [ ,HEX  ]
                        [,VALIDCK=symbolic-address]
                        [,HELP=('help data', 'help data',...)]
```

Figure 25. The IKJIDENT macro instruction

label
This name is used within the PDL DSECT as the symbolic address of the parameter descriptor entry for this positional operand.

'*parameter-type*'
This field is required so that the operand can be identified when an error message is necessary. This field differs from the PROMPT field in that the PROMPT field is not required and if supplied is used only for a required operand that is not entered by the terminal user. Blanks within the apostrophes are allowed.

LIST
This positional operand can be entered by the terminal user as a list, that is, in the form:

```
commandname (operand,operand,…)
```

RANGE
This positional operand can be entered by the terminal user as a range, that is, in the form:

```
commandname operand:operand
```

If you specify RANGE and OTHER=ANY, parse treats any colons that it finds as delimiters. For example, the first colon after RANGE marks the end of the first part of the range and the start of the next part of the range. To include the colon in your data, you must use the CHAR operand and enclose the colon in quotation marks.

PTBYPS
All prompting for the operand is to be done in print inhibit mode. This option can be specified only when the PROMPT option is specified.

ASTERISK
An asterisk can be substituted for this positional operand.

Note: ASTERISK and INTEG are mutually exclusive.

UPPERCASE | ASIS

UPPERCASE
The operand is to be translated to uppercase.

ASIS
The operand is to be left as it was entered.

MAXLNTH=*number*
The maximum number of characters the string can contain. The number must be a value from 1 to 255. If you do not code the MAXLNTH operand, the Parse Service Routine accepts a character string of any length. MAXLNTH is determined by the length of the input string. This might not be the same as the length returned in the PDE.

FIRST=
Specify the character type restriction on the first character of the string.

OTHER=
Specify the character type restriction on the characters of the string other than the first character.

Specify the restrictions on the characters of the string by coding one of the following character types after the FIRST= and the OTHER= operands. This is true unless HEX, INTEG, or CHAR is specified; FIRST= and OTHER= serve no purpose in these cases.

ALPHA
An alphabetic character or one of the special characters $, #, @. ALPHA is the default value for both the FIRST and the OTHER operands.

NUMERIC
A digit, 0-9.

ALPHANUM
An alphabetic character, one of the special characters $, #, @, or a number 0-9.

ANY
Any character within the IBM code page 037 (EBCDIC code page) other than a blank, comma, tab, or semicolon. Parentheses must be balanced.

NONATABC
An alphabetic character only. The special characters $, #, @, and numerics are excluded.

NONATNUM
An alphabetic or numeric character. The special characters $, #, @ are excluded.

PROMPT='*prompt data*'
The operand described by this IKJIDENT macro instruction is required. The prompting data that you specify is issued as a message if the operand is not entered by the terminal user. If prompting is necessary and the terminal is in prompt mode, the Parse Service Routine adds a message-identifying number (message ID) and the word ENTER to the beginning of this message before writing it to the terminal. If prompting is necessary but the terminal is in no-prompt mode, the Parse Service Routine adds a message ID and the word MISSING to the beginning of this message before writing it to the terminal.

DEFAULT='*default value*'
The operand is required, but a default value can be used. If the operand is not entered by the terminal user, the value specified as the default value is used.

Note: The operand is optional if PROMPT or DEFAULT is specified. The Parse Service Routine takes no action if the operand specified by this IKJIDENT macro instruction is not present in the command buffer.

CHAR
Specifies that the Parse Service Routine is to accept a string of characters as input. This input string can be either quoted or unquoted.

INTEG
Specifies that the Parse Service Routine is to accept a numeric quantity as input. This quantity can be decimal, hexadecimal, or binary. The number is stored internally as a fullword binary value, regardless of how INTEG was specified.

Note: A maximum length is automatically implied if the INTEG option is specified. For binary input, the maximum number of characters is 32. For hexadecimal input, the maximum length is 8. For decimal input, the maximum length is 10.

HEX
Specifies that the Parse Service Routine is to accept a hexadecimal value as input. This string quantity can be hexadecimal or a quoted or non-quoted string.

Note: All input entered in the form X'n...' must be valid hexadecimal digits (0-9, A-F). All input entered in the form B'n...' must be valid binary digits (0,1). All input entered as unquoted decimals must be valid decimal digits 0-9.

VALIDCK=*symbolic-address*
Supply the symbolic address of a validity checking routine if you want to perform additional validity checking on this operand. The Parse Service Routine calls the addressed routine after first determining that the operand is syntactically correct.

HELP=('*help data*','*help data*',...)
You can provide up to 255 second-level messages. (However, note that the assembler in use can limit the number of characters that a macro operand with a sublist can contain.) Enclose each message in apostrophes and separate the messages by single commas. These messages are issued one at a time after each question mark entered by the terminal user in response to a prompting message from the Parse Service Routine. These messages are not sent to the user when the prompt is for a password on a DSNAME or USERID operand.

The Parse Service Routine adds a message ID and the word ENTER (in prompt mode) or MISSING (in no-prompt mode) to the beginning of each message before writing it to the terminal.

The parameter control entry built by IKJIDENT

The IKJIDENT macro instruction generates the variable-length parameter control entry (PCE) shown in Table 21 on page 86.

Table 21. The parameter control entry built by IKJIDENT

Number of bytes	Field name	Contents or meaning
2		Flags. These flags are set to indicate which options were specified in the IKJIDENT macro instruction. Byte 1: **100. ....** This is an IKJIDENT PCE. **...1 ....** PROMPT **.... 1...** DEFAULT **.... .1..** This is an extended format PCE. If the VALIDCK parameter is specified, the length of the field containing the address of the validity checking routine is four bytes. **.... ..1.** HELP **.... ...1** VALIDCK Byte 2: **1... ....** LIST **.1.. ....** ASIS **..1. ....** RANGE **...0 0000** Reserved

Table 21. The parameter control entry built by IKJIDENT (continued)

Number of bytes	Field name	Contents or meaning
2		Length of the parameter control entry. This field contains a hexadecimal number representing the number of bytes in this IKJIDENT PCE.
2		Contains a hexadecimal offset from the beginning of the parameter descriptor list to the related parameter descriptor entry built by the Parse Service Routine.
1		A flag field indicating the options coded on the IKJIDENT macro instruction. **1... ....** ASTERISK **.1.. ....** MAXLNTH **..1. ....** PTBYPS **...1 ....** Integer **.... 1...** Character **.... .1..** Hexadecimal **.... ..00** Reserved
1		This field contains a hexadecimal number indicating the character type restriction on the first character of the character string described by the IKJIDENT macro instruction. **HEX** **Acceptable characters** **X'0'** Any (except blank, comma, tab, semicolon) **X'1'** Alphabetic or one of the special characters $, #, @ **X'2'** Numeric **X'3'** Alphabetic, numeric, or one of the special characters $, #, @ **X'4'** Alphabetic **X'5'** Alphabetic or numeric

Table 21. The parameter control entry built by IKJIDENT (continued)

Number of bytes	Field name	Contents or meaning
1		This field contains a hexadecimal number indicating the character type restriction on the other characters of the character string described by the IKJIDENT macro instruction. **HEX** **Acceptable characters** **0** Any (except blank, comma, tab, semicolon) **1** Alphabetic or one of the special characters $, #, @ **2** Numeric **3** Alphabetic, numeric, or one of the special characters $, #, @ **4** Alphabetic **5** Alphabetic or numeric
2		This field contains a hexadecimal number representing the length of the parameter type segment. This figure includes the length of this field, the length of the message segment offset field, and the length of the parameter type field supplied on the IKJIDENT macro instruction.
2		This field contains the message segment offset. It is set to X'0012'.
Variable		This field contains the field supplied as the parameter type operand of the IKJIDENT macro instruction.
1		This field contains a hexadecimal number representing the maximum number of characters the string can contain. This field is present only if the MAXLNTH operand was coded on the IKJIDENT macro instruction.
1		This field contains the length minus one of the defaults or prompting information supplied on the IKJIDENT macro instruction. This field and the next are present only if DEFAULT or PROMPT were specified on the IKJIDENT macro instruction.
Variable		This field contains the prompting or default information supplied on the IKJIDENT macro instruction.
2		This field contains a hexadecimal figure representing the length in bytes of all the PCE fields used for second-level messages. The figure includes the length of this field. The fields are present only if HELP is specified on the IKJIDENT macro instruction.
1		This field contains a hexadecimal number representing the number of second-level messages specified by HELP on this IKJIDENT PCE.
2		This field contains a hexadecimal number representing the length of this HELP segment. The figure includes the length of this field, the message segment offset field, and the length of the information. These fields are repeated for each second-level message specified by HELP on the IKJIDENT macro instruction.
2		This field contains the message segment offset. It is set to X'0000'.
Variable		This field contains one second-level message supplied on the IKJIDENT macro instruction specified by HELP. This field and the two preceding ones are repeated for each second-level message supplied on the IKJIDENT macro instruction; these fields do not appear if no second-level message data was supplied.
3 or 4		This field contains the address of a validity checking routine if VALIDCK was specified on the IKJIDENT macro. If the "extended format PCE" bit is on in the IKJIDENT PCE, the address is four bytes long; if the bit is off, the address is three bytes long. This field is not present if VALIDCK was not specified.

Using IKJKEYWD to describe a keyword operand

To describe a keyword operand, use the IKJKEYWD macro instruction immediately followed by a series of IKJNAME macro instructions that indicate the possible names for the keyword operand. See “Using IKJNAME to list keyword or reserved word operand names” on page 90 for information on the IKJNAME macro instruction.

Keyword operands can appear in any order in the command but must follow all positional operands. A user is never required to enter a keyword operand; if he does not, the default value you supply, if you choose to supply one, is used. Keywords can consist of any combination of alphanumeric characters up to 31 characters in length, the first of which must be an alphabetic character.

Figure 26 on page 89 shows the format of the IKJKEYWD macro instruction. Each of the operands is explained following the figure.

```
label     IKJKEYWD     [DEFAULT='default-value']
```

Figure 26. The IKJKEYWD macro instruction

label
This name is used within the PDL DSECT as the symbolic address of the parameter descriptor entry for this operand.

DEFAULT=‘*default-value*’
The default value you specify is the value that is used if this keyword is not present in the command buffer. Specify the valid keyword names with IKJNAME macro instructions following this IKJKEYWD macro instruction.

The parameter control entry built by IKJKEYWD

The IKJKEYWD macro instruction generates the variable-length parameter control entry (PCE) shown in Table 22 on page 89.

Table 22. The parameter control entry built by IKJKEYWD

Number of bytes	Field name	Contents or meaning
2		Flags. These flags are set to indicate which options were coded in the IKJKEYWD macro instruction. Byte 1: **010. ....** This is an IKJKEYWD PCE. **...0 ....** Reserved. **.... 1...** DEFAULT **.... .000** Reserved. Byte 2: **0000 0000** Reserved
2		Length of the parameter control entry. This field contains a hexadecimal number representing the number of bytes in this IKJKEYWD PCE.
2		This field contains a hexadecimal offset from the beginning of the parameter descriptor list to the related parameter descriptor entry built by the Parse Service Routine.
1		This field contains the length minus one of the default information supplied on the IKJKEYWD macro instruction. This field and the next are present only if DEFAULT was specified on the IKJKEYWD macro instruction.

Table 22. The parameter control entry built by IKJKEYWD (continued)

Number of bytes	Field name	Contents or meaning
Variable		This field contains the default value supplied on the IKJKEYWD macro instruction.

Using IKJNAME to list keyword or reserved word operand names

Use the IKJNAME macro instruction to do the following:

- Define keyword operand names. In this case, use the IKJNAME macro instruction with the IKJKEYWD macro instruction.
- Define reserved word operand names. In this case, use the IKJNAME macro instruction with the IKJRSVWD macro instruction.

Defining keyword operand names

Use a series of IKJNAME macro instructions to indicate the possible names for a keyword operand. One IKJNAME macro instruction is needed for each possible keyword name. Code the IKJNAME macro instructions immediately following the IKJKEYWD macro instruction to which they pertain.

Figure 27 on page 90 shows the format of the IKJNAME macro instruction. Each of the operands is explained following the figure.

```
IKJNAME     'keyword-name'[,SUBFLD=subfield-name]
            [,INSERT='keyword-string']
            [,ALIAS=('name','name',...)]
```

Figure 27. The IKJNAME macro instruction (when used with the IKJKEYWD macro instruction)

keyword-name
One of the valid keyword operands for the IKJKEYWD macro instruction that precedes this IKJNAME macro instruction.

SUBFLD=*subfield-name*
This option indicates that this keyword name has other operands associated with it. Use the subfield-name as the label field of the IKJSUBF macro instruction that begins the description of the possible operands in the subfield. See “Using IKJSUBF to describe a keyword subfield” on page 92 for a description of the IKJSUBF macro instruction.

INSERT=‘*keyword-string*’
The use of some keyword operands implies that other keyword operands are required. The Parse Service Routine inserts the keyword string specified into the command string just as if it had been entered as part of the original command string. The command buffer is not altered.

ALIAS=(‘*name*’,‘*name*’,...)
specifies up to 32 alias names for a keyword. Each name represents a valid abbreviation or alternate name and must be enclosed in quotes. All abbreviations or names must be enclosed in a single set of parentheses.

Parse automatically accepts a keyword abbreviation if the abbreviation is distinguishable from the other keywords of the command or subcommand. Therefore, parse does not require you to code unambiguous abbreviations as alias names.

Defining reserved word operand names

Use a series of IKJNAME macro instructions to indicate the possible names for reserved words. One IKJNAME macro instruction is needed for each possible reserved word name. Code the IKJNAME macro instructions immediately following the IKJRSVWD macro instruction to which they apply.

Figure 28 on page 91 shows the format of the IKJNAME macro instruction. Each of the operands is explained following the figure.

```
IKJNAME     'reserved-word name'
```

Figure 28. The IKJNAME macro instruction (when used with the IKJRSVWD macro instruction)

reserved-word name
One of the valid reserved word operands for the IKJRSVWD macro instruction that precedes the IKJNAME macro instructions.

Note: The IKJNAME macro instruction has two uses when coded with the IKJRSVWD macro instruction. The reserved-words identified on the IKJNAME macros can be figurative constants when the IKJRSVWD macro is chained from an IKJTERM macro, or operators in an expression when the IKJRSVWD macro is chained from the IKJOPER macro. See "Using IKJRSVWD to describe a delimiter-dependent positional parameter" on page 81 for more information on using the IKJRSVWD macro instruction.

The parameter control entry built by IKJNAME

The IKJNAME macro instruction generates the variable-length parameter control entry (PCE) shown in Table 23 on page 91.

Note: Only the first four fields are valid when the IKJNAME macro instruction is coded with the IKJRSVWD macro instruction.

Table 23. The parameter control entry built by IKJNAME

Number of bytes	Field name	Contents or meaning
2		Flags. These flags are set to indicate which options were coded in the IKJNAME macro instruction. Byte 1: **011. ....** This is an IKJNAME PCE. **...0 0...** Reserved. **.... .1..** SUBFLD **.... ..00** Reserved. Byte 2: **000. ....** Reserved. **...1 ....** INSERT **.... ..1.** ALIAS **.... 00.0** Reserved.
2		Length of the parameter control entry. This field contains a hexadecimal number representing the number of bytes in this IKJNAME PCE.
1		This field contains the length minus one of the keyword or reserved word names specified on the IKJNAME macro instruction.
Variable		This field contains the keyword or reserved word name specified on the IKJNAME macro instruction.
2		This field contains a hexadecimal offset, plus one, from the beginning of the parameter control list to the beginning of a subfield PCE. This field is present only if the SUBFLD operand was specified in the IKJNAME macro instruction.

Table 23. The parameter control entry built by IKJNAME (continued)

Number of bytes	Field name	Contents or meaning
1		This field contains the length minus one of the keyword string included as the INSERT operand in the IKJNAME macro instruction. This field and the next are not present if INSERT was not specified.
Variable		This field contains the keyword string specified as the INSERT operand of the IKJNAME macro instruction.
1		The total number of aliases.
1		The length of first alias.
Variable		The first alias.
1		The length of second alias.
Variable		The second alias.

Using IKJSUBF to describe a keyword subfield

Keyword operands can have subfields associated with them. A subfield consists of a parenthesized list of operands (either positional or keyword types) which directly follows the keyword.

Use the IKJSUBF macro instruction to indicate the beginning of a subfield description. The IKJSUBF macro instruction ends the main part of the parameter control list or the previous subfield description, and begins a new subfield description. All subfield descriptions must occur after the main part of the parameter control list.

The IKJSUBF macro instruction is used only to begin the subfield description; the subfield is described using the IKJPOSIT, IKJIDENT, IKJUNFLD, and IKJKEYWD macro instructions, depending upon the type of operands within the subfield.

The label of this macro instruction must be the same name as the SUBFLD operand of the IKJNAME macro instruction that you coded to describe the keyword name.

Figure 29 on page 92 shows the format of the IKJSUBF macro instruction.

```
label     IKJSUBF
```

Figure 29. The IKJSUBF macro instruction

label
The name you supply as the label of this macro instruction must be the same name you have coded as the SUBFLD operand of the IKJNAME macro instruction describing the keyword name that takes this subfield.

The parameter control entry built by IKJSUBF

The IKJSUBF macro instruction generates the parameter control entry (PCE) shown in Table 24 on page 93.

Table 24. The parameter control entry built by IKJSUBF

Number of bytes	Field name	Contents or meaning
1		Flags. These flags indicate which type of PCE this is. **000. ....** This PCE indicates an end-of-field. These end-of-field indicators are present in IKJSUBF and IKJENDP PCEs; they indicate the end of a previous subfield or of the PCL itself. **...0 0000** Reserved.
2		This field contains a hexadecimal number representing the offset within the PCL to the first IKJKEYWD PCE or to the next end-of-field indicator if there are no keywords in this subfield.

Using IKJUNFLD to describe unidentified keyword operands

Use the IKJUNFLD macro instruction to indicate that the Parse Service Routine should do the following:

- Accept an unidentified keyword operand that it encounters in the command buffer. An unidentified keyword operand is an operand that is not defined in the parameter control list (PCL).
- Pass control to an indicated verify exit routine to perform checking on the unidentified keyword operand.

Note:

1. When unidentified keyword operands are present in the command buffer, the Parse Service Routine uses only the first specification of the IKJUNFLD macro instruction. Similarly, when an unidentified keyword is present within a subfield, parse uses only the first specification of the IKJUNFLD macro instruction within a subfield specification.
2. Parse processes unidentified operands within a subfield in the same way as keyword operands. The exception is when the unidentified operand is the subfield of an unidentified operand. In this case, unidentified operands in a subfield can be up to 31 characters in length and can contain any character other than a blank, comma, tab, or semicolon. An unknown subfield of an unknown keyword allows the specification of any character type.

   An example of an unknown keyword and unknown subfield is:

   ```
   ABC(9AB6)
   ```

   Parse interprets left parentheses within a subfield to indicate the start of a sublist.
3. The extended format of the IKJUNFLD macro instruction allows unidentified operands within a subfield to be up to 250 characters. If the subfield string is not in quotes, it can contain any character other than a blank, comma, tab, or semicolon. If the string is in quotes, parse recognizes any character, including a blank.

   Note that if you issue a command in ISPF or program control facility (PCF), the extended format of IKJUNFLD does not immediately receive control, the character that is used as the ISPF or PCF command delimiter still functions as a delimiter, and might cause a syntax error. The default command delimiter character for each is the semicolon (;) unless respecified by your installation.

Figure 30 on page 93 shows the format of the IKJUNFLD macro instruction.

```
IKJUNFLD     VERIFCK=symbolic-address
             [,SUBFLD=subfield-name]
             [,EXT]
```

Figure 30. The IKJUNFLD macro instruction

VERIFCK=*symbolic-address*
Supply the symbolic address of a verify exit routine that checks unidentified keyword operands. The Parse Service Routine passes control to this routine when it encounters an unidentified keyword operand. “Using verify exit routines” on page 102 describes what you must do to provide a verify exit routine.

SUBFLD=*subfield-name*
This option indicates that the unidentified keyword has other operands that are associated with it. Use the subfield-name as the label field of the IKJSUBF macro instruction that begins the description of the possible operands in the subfield. See “Using IKJSUBF to describe a keyword subfield” on page 92 for a description of the IKJSUBF macro instruction.

EXT
This option specifies extended parsing. The extended format allows up to 250 characters in the subfield. It also recognizes quoted strings, which might contain blanks. If a quoted string is processed with extended parsing, the quotes are stripped and the PPEEXTQS flag is set in the parse parameter element (PPE).

The parameter control entry built by IKJUNFLD

The IKJUNFLD macro instruction generates the parameter control entry (PCE) shown in Table 25 on page 94.

Table 25. The parameter control entry built by IKJUNFLD

Number of bytes	Field name	Contents or meaning
2		Flags. Byte 1: **0101 ....** This is an IKJUNFLD PCE. **.... 1...** This is an EXT-type IKJUNFLD PCE. **.... .000** Reserved. Byte 2: **0000 0000** Reserved.
2		Length of the parameter control entry. This field contains a hexadecimal number representing the number of bytes in this IKJUNFLD PCE.
4		This field contains the address of a verify exit routine.
2		Flags. These flags are set to indicate that this field begins an IKJKEYWD sub-PCE. This 6-byte sub-PCE is generated by the IKJUNFLD macro instruction to describe a keyword operand. Byte 1: **010. ....** This is an IKJKEYWD sub-PCE. **...0 0000** Reserved. Byte 2: **0000 0000** Reserved.
2		Length of this sub-PCE. This field contains the hexadecimal number X'6', which is the number of bytes in this IKJKEYWD sub-PCE.
2		This field contains a hexadecimal offset from the beginning of the parameter descriptor list to the related parameter descriptor entry built by the Parse Service Routine.

Table 25. The parameter control entry built by IKJUNFLD (continued)

Number of bytes	Field name	Contents or meaning
2		Flags. These flags are set to indicate that this field begins an IKJNAME sub-PCE. This sub-PCE is generated by the IKJUNFLD macro instruction to describe a keyword operand. Byte 1: **011. ....** This is an IKJNAME sub-PCE. **...0 0...** Reserved. **.... .1..** SUBFLD **.... ..00** Reserved. Byte 2: **0000 0000** Reserved.
2		Length of this sub-PCE. This field contains a hexadecimal value representing the number of bytes in this IKJNAME sub-PCE. If a subfield is *not* specified, this field contains the value X'24' for a standard-type IKJUNFLD or X'FF' for an EXT-type IKJUNFLD. If a subfield has been specified, this field contains the value X'26' for a standard-type IKJUNFLD or X'101' for an EXT-type IKJUNFLD.
1		This field contains the length minus one of the next field. The value is either X'1E' or X'F9'.
Variable		This field contains a dummy name. For a standard-type IKJUNFLD, it contains 31 blanks. For a EXT-type IKJUNFLD, it contains 250 blanks.
2		This field contains a hexadecimal offset, plus one, from the beginning of the parameter control list to the beginning of a subfield PCE. This field is present only if the SUBFLD operand was specified on the IKJUNFLD macro instruction.

Using IKJENDP to end the parameter control list

Use the IKJENDP macro instruction to inform the parse service routine that it has reached the end of the parameter control list built for this command.

Figure 31 on page 95 shows the format of the IKJENDP macro instruction.

```
IKJENDP
```

Figure 31. The IKJENDP macro instruction

The parameter control entry built by IKJENDP

The IKJENDP macro instruction generates the parameter control entry (PCE) shown in Table 26 on page 95. It is merely an end-of-field indicator.

Table 26. The parameter control entry built by IKJENDP

Number of bytes	Field name	Contents or meaning
1		Flags. These flags are set to indicate end-of-field. **000. ....** End-of-field indicator. Indicates the end of the PCL. **...0 0000** Reserved.

Using IKJRLSA to release virtual storage allocated by parse

Use the IKJRLSA macro instruction to release virtual storage allocated by the Parse Service Routine and not previously released by the Parse Service Routine. This storage consists of the parameter descriptor list (PDL) returned by the Parse Service Routine and any virtual storage obtained for new data received by parse as a result of a prompt.

If the return code from the Parse Service Routine is non-zero, parse has freed all virtual storage that it has allocated. In this case, you do not need to issue this macro instruction, but it will not cause an error if you do issue it.

Figure 32 on page 96 shows the format of the IKJRLSA macro instruction. Each of the operands is explained following the figure.

```
label     IKJRLSA     Address of the answer place
                              (1-12)
```

Figure 32. The IKJRLSA macro instruction

address of the answer place
The address of the word in which the Parse Service Routine placed a pointer to the parameter descriptor list (PDL), when control was returned to the Command Processor. Your Command Processor can load this address into one of the general registers 1 through 12, and right adjust it with the unused high-order bits set to zero. See “Passing control to the Parse Service Routine” on page 105 for a description of the parse parameter list.

Examples using the parse macro instructions

Example 1: Describing a PROCESS command syntax

This example shows how the parse macro instructions could be used within a Command Processor to describe the syntax of a PROCESS command to the Parse Service Routine. A sample Command Processor that includes the parse macros used in this example is shown in *z/OS TSO/E Programming Guide*.

The sample PROCESS command we are describing to the parse service routine has the following format:

```
PROCESS      dsname  [ ACTION   ]
                     [ NOACTION ]
```

Figure 33 on page 97 shows the sequence of parse macro instructions that describe the syntax of this PROCESS command to the Parse Service Routine. The parse macro instructions used in this example perform the following functions:

- The IKJPARM macro instruction indicates the beginning of the parameter control list and creates the PRDSECT DSECT that you use to map the parameter descriptor list returned by the Parse Service Routine.
- The IKJPOSIT macro instruction describes the data set name, which is a positional operand. The address of a validity checking routine, POSITCHK, is specified.
- The IKJKEYWD and IKJNAME macro instructions indicate the possible names for keyword operands.
- The IKJENDP macro instruction indicates the end of the parameter control list.

```
PCLDEFS  IKJPARM  DSECT=PRDSECT
DSNPCE   IKJPOSIT DSNAME,                                              X
               PROMPT='THE NAME OF THE DATA SET YOU WANT TO PROCESS.    X
               ENTER ''?'' FOR HELP',                                   X
               HELP=('A DATA SET NAME WHICH HAS A FIRST-LEVEL QUALIFIER  X
                OTHER THAN ''SYS1''.'),                                 X
               VALIDCK=POSITCHK
ACTPCE   IKJKEYWD DEFAULT='NOACTION'
         IKJNAME  'ACTION'
         IKJNAME  'NOACTION'
         IKJENDP
```

Figure 33. Example 1 - using parse macros to describe command operand syntax

Example 2: Describing an EDIT command syntax

This example shows how the parse macro instructions could be used within a Command Processor to describe the syntax of an EDIT command to the Parse Service Routine.

The sample EDIT command we are describing to the parse service routine has the following format:

```
  EDIT       dsname
             [ PLI [([number  [number]]   [CHAR60 )]]]
             [     [ [  2     [  72  ]]   [CHAR48  ]]]
             [ FORT                                  ]
             [ ASM                                   ]
             [ TEXT                                  ]
             [ DATA                                  ]

             [ SCAN  ]
             [ NOSCAN]

             [ NUM  ]
             [ NONUM]

             [ BLOCK(number)  ]
             [ BLKSIZE(number)]

              LINE(number)
```

Figure 34 on page 98 shows the sequence of parse macro instructions that describe the syntax of this EDIT command to the Parse Service Routine. The parse macro instructions used in this example perform the following functions:

- The IKJPARM macro instruction indicates the beginning of the parameter control list and creates the DSECT that you use to map the parameter descriptor list returned by the Parse Service Routine. The name of the DSECT is defaulted to IKJPARMD in this example.
- The IKJPOSIT macro instruction describes the data set name, which is a positional operand.
- The IKJKEYWD and IKJNAME macro instructions indicate the possible names for keyword operands.
- The IKJSUBF macro instruction indicates the beginning of subfield descriptions for keyword operands. Within these subfields, IKJIDENT and IKJKEYWD macro instructions describe the positional and keyword operands.
- The IKJENDP macro instruction indicates the end of the parameter control list.

```
PARMTAB  IKJPARM
DSNAME   IKJPOSIT   DSNAME,PROMPT='DATA SET NAME'
TYPE     IKJKEYWD
         IKJNAME    'PL1',SUBFLD=PL1FLD
         IKJNAME    'FORT'
         IKJNAME    'ASM'
         IKJNAME    'TEXT'
         IKJNAME    'DATA'
SCAN     IKJKEYWD   DEFAULT='NOSCAN'
         IKJNAME    'SCAN'
         IKJNAME    'NOSCAN'
NUM      IKJKEYWD   DEFAULT='NUM'
         IKJNAME    'NUM'
         IKJNAME    'NONUM'
BLOCK    IKJKEYWD
         IKJNAME    'BLOCK',SUBFLD=BLOCKSUB,ALIAS='BLKSIZE'
LINE     IKJKEYWD
         IKJNAME    'LINE',SUBFLD=LINESIZE
PL1FLD   IKJSUBF
PL1COL1  IKJIDENT   'NUMBER',FIRST=NUMERIC,OTHER=NUMERIC,DEFAULT='2'
PL1COL2  IKJIDENT   'NUMBER',FIRST=NUMERIC,OTHER=NUMERIC,DEFAULT='72'
PL1TYPE  IKJKEYWD   DEFAULT='CHAR60'
         IKJNAME    'CHAR60'
         IKJNAME    'CHAR48'
BLOCKSUB IKJSUBF
BLKNUM   IKJIDENT   'NUMBER',FIRST=NUMERIC,OTHER=NUMERIC,              X
               PROMPT='BLOCKSIZE',MAXLNTH=8
LINESIZE IKJSUBF
LINNUM   IKJIDENT   'NUMBER',FIRST=NUMERIC,OTHER=NUMERIC,              X
               PROMPT='LINESIZE'
         IKJENDP
```

Figure 34. Example 2 - using parse macros to describe command operand syntax

Example 3: Describing an AT command syntax

This example shows how the parse macro instructions could be used to describe the syntax of a sample AT command that has the following syntax:

```
          [stmt               ]
 AT       [(stmt-1,stmt-2,...)] (cmd chain) COUNT(integer)
          [stmt-3:stmt-4      ]
```

Figure 35 on page 99 shows the sequence of parse macro instructions that describe this sample AT command to the Parse Service Routine. The parse macro instructions used in this example perform the following functions:

- The IKJPARM macro instruction indicates the beginning of the parameter control list and creates the PARSEAT DSECT that you use to map the parameter descriptor list returned by the Parse Service Routine.
- The IKJTERM macro instruction indicates that the terminal user can enter the statement number as a single value or as a list or range of values.
- The IKJPOSIT macro instruction indicates that the user must enter the subcommand-chain as a parenthesized string.
- The IKJKEYWD and IKJNAME macro instructions indicate the name of the keyword operand COUNT.
- The IKJSUBF macro instruction indicates the beginning of a subfield description for the keyword operand. Within this subfield, an IKJIDENT macro instruction describes the positional operand.
- The IKJENDP macro instruction indicates the end of the parameter control list.

```
EXAM2    IKJPARM    DSECT=PARSEAT
STMTPCE  IKJTERM    'STATEMENT NUMBER',UPPERCASE,LIST,RANGE,TYPE=STMT, X
               VALIDCK=CHKSTMT
POSITPCE IKJPOSIT   PSTRING,HELP='CHAIN OF COMMANDS',VALIDCK=CHKCMD
KEYPCE   IKJKEYWD
NAMEPCE  IKJNAME    'COUNT',SUBFLD=COUNTSUB
COUNTSUB IKJSUBF
IDENTPCE IKJIDENT   'COUNT',FIRST=NUMERIC,OTHER=NUMERIC,               X
               VALIDCK=CHKCOUNT
         IKJENDP
```

Figure 35. Example 3 - using parse macros to describe command operand syntax

Example 4: Describing a LIST command syntax

This example shows how the parse macro instructions could be used to describe the syntax of a sample LIST command that has the following syntax:

```
LIST     symbol PRINT(symbol)
```

Figure 36 on page 99 shows the sequence of parse macro instructions that describe this sample LIST command to the Parse Service Routine. The parse macro instructions used in this example perform the following functions:

- The IKJPARM macro instruction indicates the beginning of the parameter control list and creates the PARSELST DSECT that you use to map the parameter descriptor list returned by the Parse Service Routine.
- The IKJTERM macro instruction describes a subscripted variable, such as,

  ```
  a of b in c(1)
  ```

  that the terminal user must specify.
- The IKJKEYWD and IKJNAME macro instructions indicate the name of the keyword operand PRINT.
- The IKJSUBF macro instruction indicates the beginning of a subfield description for the keyword operand. Within this subfield, an IKJTERM macro instruction describes the positional operand.
- The IKJENDP macro instruction indicates the end of the parameter control list.

```
EXAM3    IKJPARM    DSECT=PARSELST
VARPCE   IKJTERM    'SYMBOL',UPPERCASE,PROMPT='SYMBOL',TYPE=VAR,       X
               VALIDCK=CHECK,SBSCRPT=SUBPCE
SUBPCE   IKJTERM    'SUBSCRIPT',SBSCRPT,TYPE=CNST,PROMPT='SUBSCRIPT'
KEYPCE   IKJKEYWD
NAMEPCE  IKJNAME    'PRINT',SUBFLD=PRINTSUB
PRINTSUB IKJSUBF
         IKJTERM    'SYMBOL-2',UPPERCASE,PROMPT='SYMBOL-2',TYPE=VAR
IKJENDP
```

Figure 36. Example 4 - using parse macros to describe command operand syntax

Example 5: Describing a WHEN command syntax

This example shows how the parse macro instructions could be used to describe the syntax of a sample WHEN command that has the following syntax:

```
WHEN     [addr      ] (subcommand chain)
         [expression]
```

Figure 37 on page 100 shows the sequence of parse macro instructions that describe this sample WHEN command to the Parse Service Routine. The parse macro instructions used in this example perform the following functions:

- The IKJPARM macro instruction indicates the beginning of the parameter control list and creates the PARSEWHN DSECT that you use to map the parameter descriptor list returned by the Parse Service Routine.
- The IKJOPER macro instruction describes an operand that can be entered as either an expression or a variable.
- The IKJTERM macro instructions that are labeled "SYMBOL" and "SYMBOL2" describe the operands that are part of the expression.
- The IKJRSVWD and IKJNAME macro instructions define possible reserved words that can be operators in the expression.
- The IKJTERM macro instruction that is labeled "ADDR1" describes the variable that can be specified as the first positional operand.
- The IKJPOSIT macro instruction describes a parenthesized string.
- The IKJENDP macro instruction indicates the end of the parameter control list.

```
EXAM4    IKJPARM    DSECT=PARSEWHN
OPER     IKJOPER    'EXPRESSION',OPERND1=SYMBOL1,OPERND2=SYMBOL2,      X
                RSVWD=OPERATOR,CHAIN=ADDR1,PROMPT='TERM',VALICHK=CHECK
SYMBOL1  IKJTERM    'SYMBOL1',UPPERCASE,TYPE=VAR,PROMPT='SYMBOL2'
OPERATOR IKJRSVWD   'OPERATOR',PROMPT='OPERATOR'
         IKJNAME    'EQ'
         IKJNAME    'NEQ'
SYMBOL2  IKJTERM    'SYMBOL2',TYPE=VAR
ADDR1    IKJTERM    'ADDRESS',TYPE=VAR,VALIDCK=CHECK1
LASTONE  IKJPOSIT   PSTRING,VALIDCK=CHECK2
         IKJENDP
```

Figure 37. Example 5 - using parse macros to describe command operand syntax

Using validity checking routines

Your Command Processor can provide a validity checking routine to do additional checking on a positional operand. Each positional operand can have a unique validity checking routine. Indicate the presence of a validity checking routine by coding the entry point address of the routine as the VALIDCK= operand in the IKJPOSIT, IKJTERM, IKJOPER, or IKJIDENT macro instructions. This address must be within the program that invokes the Parse Service Routine.

The Parse Service Routine can call validity checking routines for the following types of positional parameters:

- HEX
- VALUE
- ADDRESS
- QSTRING
- USERID
- DSNAME
- DSTHING
- CONSTANT
- VARIABLE
- STATEMENT NUMBER
- EXPRESSION

- JOBNAME
- INTEG
- Any non-delimiter-dependent parameters.

Parse passes control to the validity checking routine after it has determined that the operand is non-null and syntactically correct. If a DSNAME or USERID operand is entered with a password, parse passes control to the validity checking routine after first parsing both the userid or dsname and the password. If the terminal user enters a list, the validity checking routine is called as each element in the list is parsed. If a range is entered, the Parse Service Routine calls the validity checking routine only after both items of the range are parsed.

Passing control to validity checking routines

Parse invokes all validity checking routines in the same addressing mode in which parse is invoked. Note that if a SYNCH macro is used to invoke parse, the addressing mode of the caller can be different from that in which parse is invoked.

When the Parse Service Routine passes control to a validity checking routine, parse uses standard linkage conventions. The validity checking routine must save parse's registers and restore them before returning control to the Parse Service Routine.

The validity check parameter list

The Parse Service Routine builds a three-word parameter list and places the address of this list into register 1 before branching to a validity checking routine. This three-word parameter list has the format shown in Table 27 on page 101.

Table 27. Format of the validity check parameter list

Offset dec(Hex)	Number of bytes	Field name	Contents or meaning
0(0)	4	PDEADR	The address of the parameter descriptor entry (PDE) built by parse for this syntactically correct operand.
4(4)	4	USERWORD	The address of the user work area. This is the same address you supplied to the Parse Service Routine in the PPLUWA field in the parse parameter list.
8(8)	4	VALMSG	Initialized to X'00000000' by parse. Your validity checking routine can place the address of a second-level message in this field when it sets a return code of 4.

Return codes from validity checking routines

Your validity checking routines must return a code in general register 15 to the Parse Service Routine. These codes inform the Parse Service Routine of the results of the validity check and determine the action that parse should take. Table 28 on page 101 shows the return codes, their meaning, and the action taken by the Parse Service Routine.

Table 28. Return codes from a validity checking routine

Return code dec(Hex)	Meaning	Action taken by parse
0(0)	The operand is valid.	No additional processing is performed on this operand by the Parse Service Routine.
4(4)	The operand is not valid.	The Parse Service Routine writes an error message to the terminal and prompts for a valid operand.

Table 28. Return codes from a validity checking routine (continued)

Return code dec(Hex)	Meaning	Action taken by parse
8(8)	The operand is not valid.	The validity checking routine has issued an error message; parse prompts for a valid operand.
12(C)	The operand is not valid; syntax checking cannot continue.	The Parse Service Routine stops all further syntax checking, sets a return code of 20, and returns to the calling routine.

If the Parse Service Routine receives a return code of 4 or 8, it processes new data entered in response to the prompt as though it were the original data, and passes control again to the validity checking routine. This cycle continues until a valid operand is obtained.

Prior to issuing a return code of 12, your validity checking routine should issue a message indicating that it has requested that parse terminate.

Using verify exit routines

Your Command Processor can provide verify exit routines to perform checking when the Parse Service Routine encounters either of the following in the command buffer:

- Unidentified keyword operands
- Unidentified keyword operands within a subfield.

To indicate the presence of a verify exit routine, specify the entry point address of the routine on the VERIFCK= operand in the IKJUNFLD macro instruction. This address must be within the program that invokes the Parse Service Routine.

When the Parse Service Routine encounters a keyword operand or subfield operand in the command buffer that is not specifically defined in the PCL, it determines whether a PCE has been created by the IKJUNFLD macro instruction. If parse encounters such a PCE, it gives control to the verify exit routine; if it does not, the operand is treated as not valid. When unidentified keyword operands are present in the command buffer, the Parse Service Routine uses only the first specification of the IKJUNFLD macro instruction. Similarly, when an unidentified keyword is present within a subfield, parse uses only the first specification of the IKJUNFLD macro instruction within a subfield specification.

Passing control to verify exit routines

Parse invokes all verify exit routines in the same addressing mode in which parse is invoked. If a SYNCH macro is used to invoke parse, the addressing mode of the caller can be different from that in which parse is invoked.

When the Parse Service Routine passes control to a verify exit routine, parse uses standard linkage conventions. The verify exit routine must save parse's registers and restore them before returning control to the Parse Service Routine.

The verify exit parameter list

The Parse Service Routine builds an eight-word parameter list and places the address of this list into register 1 before branching to a verify exit routine. This eight-word parameter list, the verify exit parameter list (VEPL), has the format shown in Table 29 on page 102.

Table 29. The verify exit parameter list

Offset dec(Hex)	Number of bytes	Field name	Contents or meaning
0(0)	4	VEPLID	Parse sets this field to the value of C'VEPL'.

Table 29. The verify exit parameter list (continued)

Offset dec(Hex)	Number of bytes	Field name	Contents or meaning
4(4)	2	VEPLVERS	Parse sets this field to the version number of the VEPL.
6(6)	2	VEPLLEN	Parse sets this field to the length of the VEPL.
8(8)	4	VEPLPPE	Parse sets this field to the address of the parse parameter element (PPE) that describes the operand.
12(C)	4	VEPLWRKA	The address of the user work area. This field is set by the Parse Service Routine to the value you supplied to parse in the PPLVEWA field in the parse parameter list (PPL).
16(10)	4	VEPLMSG1	The address of an insert for a first-level message to be issued by parse when the verify exit routine indicates that the keyword operand is not valid. This field should be set by the verify exit routine when its return code is 4 or 12.
20(14)	2	VEPLM1LN	The length of the message insert whose address is contained in VEPLMSG1. This field should be set by the verify exit routine when its return code is 4 or 12.
22(16)	2	VEPLRSV1	Reserved.
24(18)	4	VEPLMSG2	The address of a second-level message to be issued by parse when the verify exit routine indicates that the keyword operand is not valid. This field should be set by the verify exit routine when its return code is 4 or 12.
28(1C)	2	VEPLM2LN	The length of the message whose address is contained in VEPLMSG2. This field should be set by the verify exit routine when its return code is 4 or 12.
30(1E)	2	VEPLRSV2	Reserved.

The parse parameter element

The Parse Service Routine builds a five-word parse parameter element (PPE) that describes the operand or subfield operand currently being processed. Your verify exit routine uses the information contained in the VEPL and the PPE to refer to the operand that was entered by the terminal user. Use the VEPLPPE field in the verify exit parameter list to obtain the address of the PPE. The PPE has the format shown in Table 30 on page 103.

Table 30. The parse parameter element

Offset dec(Hex)	Number of bytes	Field name	Contents or meaning
0(0)	4	PPEID	The value of C'PPE'.
4(4)	2	PPEVERS	The version number of the PPE.
6(6)	2	PPELEN	The length of the PPE.
8(8)	4	PPEOPER	The address of the unidentified operand or subfield operand being processed.
12(C)	4	PPEVEXIT	The address of the verify exit routine that will receive control to process the unidentified operand or subfield operand.
16(10)	2	PPEOPLEN	The length of the unidentified operand or subfield operand currently being processed.

Table 30. The parse parameter element (continued)

Offset dec(Hex)	Number of bytes	Field name	Contents or meaning
18(12)	1	PPEFLAGS	These flags are set by the Parse Service Routine to indicate the following: **Setting** **Meaning** **PPELST(X'80')** The current operand is in a list of operands, or is in a list of operands within a subfield. **PPENDLST(X'40')** The previous operand is the last in a list of operands, or is the last in a list of operands within a subfield. Because this flag only indicates that a list is complete, no additional data is passed to the verify exit routine when it is set. **PPENDOP(X'20')** The previous operand is the last unidentified operand or the last unidentified operand within a subfield that occurs in the command buffer. The verify exit should perform clean-up processing, if necessary. **PPENWLST(X'10')** The current operand is the first in a list of operands, or is the first in a list of operands within a subfield. **PPEEXTQS(X'08')** The current operand was originally a quoted string and the quotes have been stripped by the Parse Service Routine.
19(13)	1	PPERSVD2	Reserved.

Return codes from verify exit routines

Your verify exit routines must return a code in general register 15 to the Parse Service Routine. This code informs the Parse Service Routine of the results of the check and determines the action that parse should take. Table 31 on page 104 shows the return codes, their meaning, and the action taken by the Parse Service Routine.

Table 31. Return codes from a verify exit routine

Return code dec(Hex)	Meaning
0(0)	The operand is valid. No additional processing is performed on this operand by the Parse Service Routine.
4(4)	The operand is not valid. Parse prompts the user to reenter the operand and takes the insert for the first-level message from the VEPLMSG1 field in the VEPL. Parse takes the second-level message from the VEPLMSG2 field in the VEPL.
8(8)	The operand is not valid. The verify exit routine has issued a message indicating that the operand is not valid; parse prompts the user to reenter the operand. This return code is not valid for cleanup calls.
12(C)	The operand is not valid. Parse performs normal processing for a not valid keyword operand by either: • Issuing a message indicating that a not valid keyword operand has been entered and prompting the user to reenter the operand. In this case, parse takes the inserts for the first-level message from the VEPLMSG1 field in the VEPL. It takes the second-level message from the VEPLMSG2 field in the VEPL. • Issuing a message indicating that extraneous information has been entered. In this case, parse does not prompt the user. This return code is not valid for cleanup calls.

Table 31. Return codes from a verify exit routine (continued)

Return code dec(Hex)	Meaning
16(10)	The operand is not valid, and the verify exit routine requests that parse terminate. Parse does not issue a message or prompt the user; it sets a return code of 32, and returns to its caller. This return code is not valid for cleanup calls.
20(14)	The operand is not valid. Parse issues a message indicating that the operand is extraneous and is ignored. If parse is currently processing a subfield, it skips to the end of the subfield and continues processing. This return code is not valid for cleanup calls.

If the Parse Service Routine receives a return code of 4 or 8, it processes new data entered in response to the prompt as though it were the original data, and passes control again to the verify exit routine. This cycle continues until a valid operand is obtained. After an operand is successfully processed, parse again calls the verify exit for notification of cleanup. Return codes other than 0 or 4 are ignored by parse for cleanup processing.

Prior to issuing a return code of 16, your verify exit routine should issue a message indicating that it has requested that parse terminate.

Passing control to the Parse Service Routine

Your Command Processor can invoke the Parse Service Routine by using either the CALLTSSR or LINK macro instructions, specifying IKJPARS as the entry point name. However, you must first create the parse parameter list (PPL) and place its address into register 1. This PPL must remain intact until the Parse Service Routine returns control to the calling routine. The PPL is described in “The parse parameter list” on page 105.

The Parse Service Routine can be invoked in either 24- or 31-bit addressing mode. IKJPARS accepts input above or below 16 MB in virtual storage. The caller's parameters must be in the primary address space.

Figure 38 on page 107 shows the flow of control between a Command Processor and the Parse Service Routine.

The parse parameter list

The parse parameter list (PPL) is an eight-word parameter list containing addresses required by the Parse Service Routine.

You can use the IKJPPL DSECT, which is provided in SYS1.MACLIB, to map the fields in the PPL. Table 32 on page 105 shows the format of the parse parameter list.

Note: The Parameter Control List (PCL) has a size limit of 65,535 bytes. If the PCL is bigger than the limit, then the parsing routine does not process the command correctly. The PCL starts with the IKJPARM macro and ends with the IKJENDP macro.

Table 32. The parse parameter list

Offset dec(Hex)	Number of bytes	Field name	Contents or meaning
0(0)	4	PPLUPT	The address of the user profile table.
4(4)	4	PPLECT	The address of the environment control table.
8(8)	4	PPLECB	The address of the command processor's event control block. The ECB is one word of storage, which must be declared and initialized to zero by your Command Processor.
12(C)	4	PPLPCL	The address of the parameter control list (PCL) created by your Command Processor using the parse macro instructions. Use the label on the IKJPARM macro instruction as the symbolic address of the PCL.

Table 32. The parse parameter list (continued)

Offset dec(Hex)	Number of bytes	Field name	Contents or meaning
16(10)	4	PPLANS	The address of a fullword of virtual storage, supplied by the calling routine, in which the Parse Service Routine places a pointer to the parameter descriptor list (PDL). If the parse of the command buffer is unsuccessful, parse sets the pointer to the PDL to X'FF000000'.
20(14)	4	PPLCBUF	The address of the command buffer.
24(18)	4	PPLUWA	A user supplied work area that parse passes to validity checking routines. This field can contain anything that your Command Processor needs to pass to a validity checking routine.
28(1C)	4	PPLVEWA	A user supplied work area that parse passes to verify exit routines. This field can contain anything that your Command Processor needs to pass to a verify exit routine.

Checking return codes from the Parse Service Routine

When the Parse Service Routine returns control to its caller, general register 15 contains one of the following return codes:

Table 33. Return codes from the Parse Service Routine

Return code dec(Hex)	Meaning
0(0)	Parse completed successfully.
4(4)	The command operands were incomplete and parse was unable to prompt.
8(8)	Parse did not complete because an attention interruption occurred during parse processing.
12(C)	Parse did not complete; the parse parameter list contains not valid information.
16(10)	Parse did not complete; no space was available.
20(14)	Parse did not complete; a validity checking routine requested termination by returning to parse with a return code of 12.
24(18)	Parse did not complete; conflicting operands were found on the IKJTERM, IKJOPER, or IKJRSVWD macro instruction.
28(1C)	Parse did not complete; the user's terminal has been disconnected or a serious error has occurred in z/OSMF ISPF.
32(20)	Parse did not complete; a verify exit routine requested termination by returning to parse with a return code of 16.
36(24)	Parse did not complete; an out-of-range DBCS character was found.
40(28)	Parse did not complete; an odd number of bytes was found in a DBCS character string.
44(2C)	Parse did not complete; a shift-out character was found with no corresponding shift-in character.
48(30)	Parse did not complete; a nested shift-out character was found.

If the Parse Service Routine returns to your Command Processor with a return code of zero, indicating that it has completed successfully, the PPLANS field in the parse parameter list contains the address of a fullword containing a pointer to the parameter descriptor list (PDL). See “Examining the PDL returned

by the Parse Service Routine" on page 108 for information on how to use the PDL to examine the results from the Parse Service Routine.

If the Parse Service Routine does not complete successfully, your Command Processor should issue a message except when the return code from parse is 4, 20 or 32. When the return code is 4, parse has already issued a message. When the return code is either 20 or 32, the validity checking routine or verify exit routine, respectively, has issued a message before it requested that parse terminate.

Your Command Processor can invoke the GNRLFAIL routine to issue meaningful error messages for the other parse return codes. See Chapter 21, "Analyzing error conditions with GNRLFAIL/VSAMFAIL," on page 357.

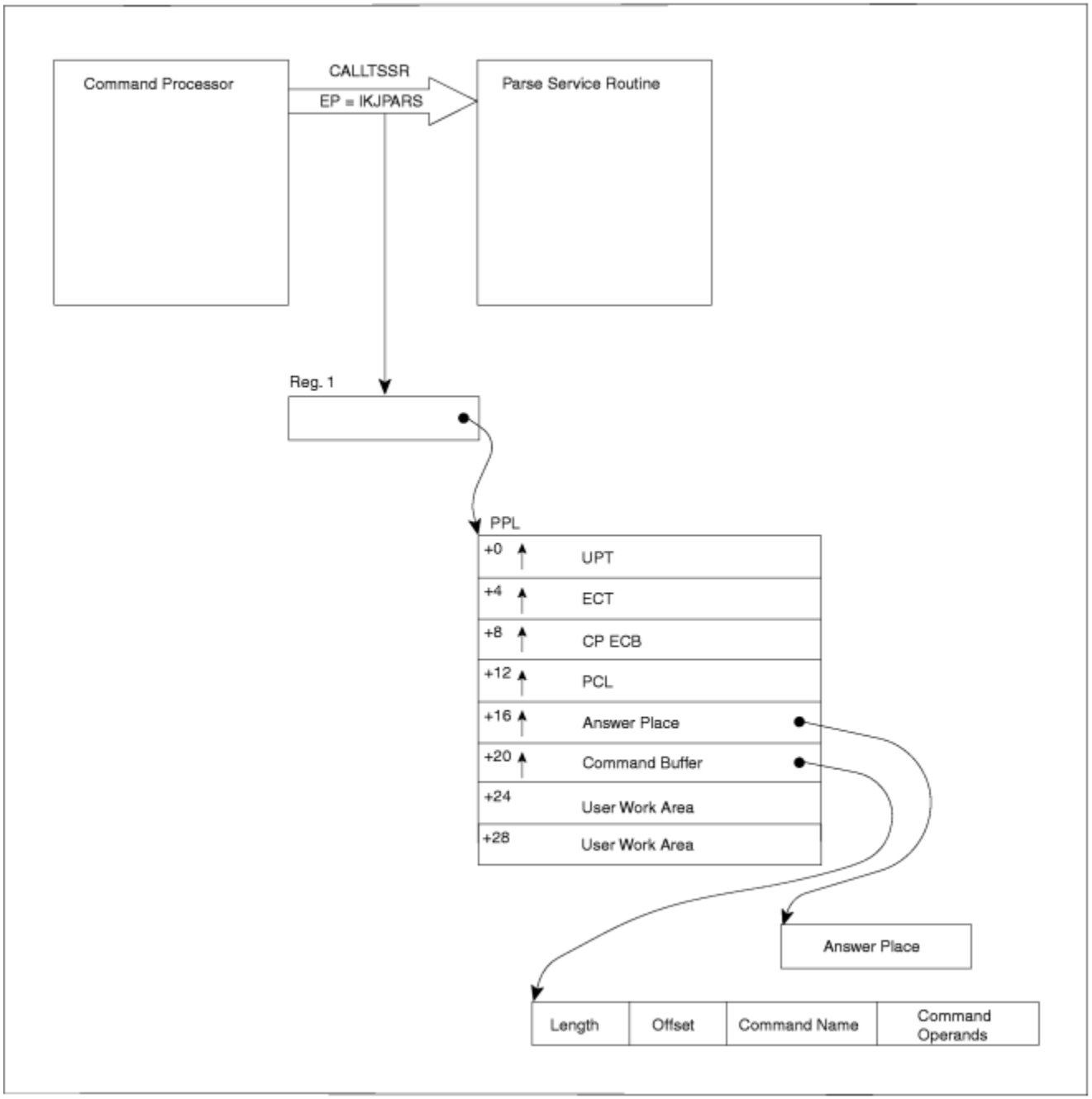


Figure 38. Control flow between Command Processor and the Parse Service Routine

Examining the PDL returned by the Parse Service Routine

The Parse Service Routine returns the results of the scan of the command buffer to the Command Processor in a parameter descriptor list (PDL). The PDL, built by parse, consists of the parameter descriptor entries (PDE), which contain pointers to the operands, indicators of the options specified, and pointers to the subfield operands entered with the command operands.

Use the name that you specified as the DSECT= operand on the IKJPARM macro instruction as the name of the DSECT that maps the PDL. The default name for this DSECT is IKJPARMD. Base this DSECT on the PDL address returned by the parse service routine. The PPLANS field of the parse parameter list points to a fullword of storage that contains the address of the PDL.

The format of the PDE depends upon the type of operand parsed. For a discussion of operand types, see the topic “Defining command operand syntax” on page 51. The following description of the possible PDEs shows each of the PDE formats and the type of operands they describe.

The PDL header

The PDL begins with a two-word header. The DSECT= operand of the IKJPARM macro instruction provides a name for the DSECT created to map the PDL. Use this name as the symbolic address of the beginning of the PDL header.

Offset decimal	Meaning
0	A pointer to the next block of virtual storage
4	Subpool number
5	Reserved
6	Length

Pointer to the next block of virtual storage:
The Parse Service Routine gets virtual storage for the PDL and for any data received as the result of a prompt. Each block of virtual storage obtained begins with another PDL header. The blocks of virtual storage are forward-chained by this field. A forward-chain pointer of X'FF000000' in this field indicates that this is the last storage element obtained.

Subpool number:
This field will always indicate subpool 1. All virtual storage allocated by the Parse Service Routine for the PDL and for data received from a prompt is allocated from subpool 1.

Length:
This field contains a hexadecimal number indicating the length of this block of real storage (this PDL). The length includes the header.

PDEs created for positional operands described by IKJPOSIT

The labels you use to name the macro instructions provide access to the corresponding PDEs. The positional operands described by the IKJPOSIT macro instruction have the following PDE formats.

SPACE, DELIMITER

The Parse Service Routine does not build a PDE for either a SPACE or a DELIMITER operand.

STRING, PSTRING, and QSTRING

The Parse Service Routine builds a two-word PDE to describe a STRING, PSTRING, or a QSTRING operand; the PDE has the following format:

Offset decimal	Meaning
0	A pointer to the character string
4	Length

Offset decimal	Meaning
6	Flags
7	Reserved

Pointer to the character string:
contains a pointer to the beginning of the character string, or a zero if the operand was omitted.

Length:
contains the length of the string. Any punctuation around the character string is not included in this length figure. The length is zero if the string is omitted or if the string is null.

Flags:

Setting	Meaning
`0... ....`	The operand is not present.
`1... ....`	The operand is present.
`.xxx xxxx`	Reserved bits.

Note: If the string is null, the pointer is set, the length is zero, and the flag bit is 1.

VALUE

The Parse Service Routine builds a two-word PDE to describe a VALUE operand; the PDE has the following format:

Offset decimal	Meaning
0	A pointer to the character string
4	Length
6	Flags
7	Type-character.

Pointer to the character string:
contains a pointer to the beginning of the character string; that is, the first character after the quote. Contains a zero if the VALUE operand is not present.

Length:
contains the length of the character string excluding the quotes.

Flags:

Setting	Meaning
`0... ....`	The operand is not present.
`1... ....`	The operand is present.
`.xxx xxxx`	Reserved bits.

Type-character:
contains the letter that precedes the quoted string.

DSNAME, DSTHING

The Parse Service Routine builds a six-word PDE to describe a DSNAME or a DSTHING operand. The PDE has the following format:

Offset decimal	Meaning
0	A pointer to the dsname
4	Length1
6	Flags1

Offset decimal	Meaning
7	Reserved
8	A pointer to the member name
12	Length2
14	Flags2
15	Reserved
16	A pointer to the password
20	Length3
22	Flag3
23	Reserved

Pointer to the dsname:
contains a pointer to the first character of the data set name. Contains zero if the data set name was omitted. Contains a pointer to the USID if it is prefixed.

Length1:
contains the length of the data set name. If the data set name is contained in quotes, this length figure does not include the quotes. When the USID is prefixed, this field will contain the total length of the data set name and the USID.

Flags1:

Setting	Meaning
`0... ....`	The data set name is not present.
`1... ....`	The data set name is present.
`.0.. ....`	The data set name is not contained within quotes.
`.1.. ....`	The data set name is contained within quotes.
`..xx xxxx`	Reserved bits.

Pointer to the member name:
contains a pointer to the beginning of the member name. Contains zero if the member name was omitted.

Length2:
contains the length of the member name. This length value does not include the parentheses around the member name.

Flags2:

Setting	Meaning
`0... ....`	The member name is not present.
`1... ....`	The member name is present.
`.xxx xxxx`	Reserved bits.

Pointer to the password:
contains a pointer to the beginning of the password. Contains zero if the password was omitted.

Length3:
contains the length of the password.

Flags3:

Setting	Meaning
`0... ....`	The password is not present.
`1... ....`	The password is present.
`.xxx xxxx`	Reserved bits.

JOBNAME

The Parse Service Routine builds a four-word PDE to describe a JOBNAME operand. The PDE has the following format:

Offset decimal	Meaning
0	A pointer to the jobname
4	Length1
6	Flags1
7	Reserved
8	A pointer to the jobid name
12	Length2
14	Flags2
15	Reserved

Pointer to the jobname:
contains a pointer to the beginning of the jobname. Contains zero if the jobname was omitted.

Length1:
contains the length of the jobname. The jobname cannot be entered in quotes.

Flags1:

Setting	Meaning
`0... ....`	The jobname is not present.
`1... ....`	The jobname is present.
`.xxx xxxx`	Reserved bits.

Pointer to the jobid:
contains a pointer to the beginning of the jobid. Contains zero if the jobid was omitted.

Length2:
contains the length of the jobid. This length figure does not include the parentheses around the jobid.

Flags2:

Setting	Meaning
`0... ....`	The jobid is not present.
`1... ....`	The jobid is present.
`.xxx xxxx`	Reserved bits.

ADDRESS

The Parse Service Routine builds a nine-word PDE to describe an ADDRESS operand. The PDE has the following format:

Offset decimal	Meaning
0	A pointer to the load name
4	Length1
6	Flags1
7	Reserved
8	A pointer to the entry name
12	Length2
14	Flags2
15	Flags3

Offset decimal	Meaning
16	A pointer to the address string
20	Length3
22	Flags4
23	Reserved
24	Flags5
25	Sign
26	Indirect count
28	A pointer to the first expression value PDE
32	Reserved for use by user validity check routine

Pointer to the load name:
contains a pointer to the beginning of the load module name. Contains zero if no load module name was specified.

Length1:
contains the length of the load module name, excluding the period.

Flags1:

Setting	Meaning
`0... ....`	The load module name is not present.
`1... ....`	The load module name is present.
`.xxx xxxx`	Reserved bits.

Pointer to the entry name:
contains a pointer to the name of the CSECT; zero if the CSECT name is not specified.

Length2:
contains the length of the entry name, excluding the period.

Flags2:

Setting	Meaning
`0... ....`	The entry name is not present.
`1... ....`	The entry name is present.
`.xxx xxxx`	Reserved bits.

Flags3:

Setting	Meaning
`1000 00..`	A single vector register element is present.
`0100 00..`	A vector register pair element is present.
`0010 00..`	A complete single vector register is present.
`0001 00..`	A complete vector register pair is present.
`0000 10..`	A vector mask register is present.
`0000 01..`	An access register is present.
`.... ..xx`	Reserved bits.

Pointer to the address string:
contains a pointer to the address string portion of a qualified address. Contains a zero if the address string was not specified.

Length3:
contains the length of the address string portion of a qualified address. This length count excludes the following characters for the following address types:

Type	Data excluded
Relative address	The plus sign.
Register address Vector mask register address Access register address	Letters.
Absolute address	The period.
Vector address	The right parenthesis.

Flags4:
The bits set in this one-byte flag field indicate whether the address string is present and what type of indirect address is represented.

Setting	Meaning
`0... ....`	The address string is not present.
`1... ....`	The address string is present.
`.0.. ....`	A 24-bit indirect address is represented.
`.1.. ....`	a 31-bit indirect address is represented.
`..xx xxxx`	Reserved bits.

Note: Bit 1 of Flags4 has no meaning if the indirect count is zero. This bit can be on only when the EXTENDED, VECTOR or AR keyword of IKJPOSIT has been specified.

Offset 23:
This byte is reserved for use by a validity checking routine.

Flags5:
The bits set in this one-byte flag field indicate the type of address found by the Parse Service Routine.

Bit setting	Hex	Meaning
`0000 0000`	00	Absolute address.
`1000 0000`	80	Symbolic address.
`0100 0000`	40	Relative address.
`0010 0000`	20	General register.
`0001 0000`	10	Double precision floating-point register.
`0000 1000`	08	Single precision floating-point register.
`0000 0100`	04	Non-qualified entry name (optionally preceded by a load name).
`0000 0010`	02	This one-byte flag field is not used to indicate the type of address.

Sign:
contains the arithmetic sign character used before the expression value defined by the first expression value PDE. If the sign field is zero and the pointer to the first expression value PDE is non-zero, then the first expression value PDE was created due to a switch in indirection symbols (?% or %?). If there are no address expression PDEs, then this field is zero.

Indirect count:
contains a number representing the number of levels of indirect addressing.

Pointer to the first expression value PDE:
This is a pointer to the first expression value PDE. Contains X'FF000000' if there are no expression value PDEs.

User word for validity checking routine:
A word provided for use by a validity checking routine.

Expression value

If the Parse Service Routine finds an ADDRESS operand to be in the form of an address expression, parse builds an expression value PDE for each expression value in the address expression.

If the EXTENDED, VECTOR, or AR keyword is specified on the IKJPOSIT macro, and parse encounters an alternating sequence of indirection symbols, (%? or ?%), parse completes the current PDE and generates a new expression value PDE.

These expression value PDEs are chained together, beginning at the eighth word of the address PDE built by the Parse Service Routine to describe the address operand. The last expression value PDE is indicated by X'FF000000' in its fourth word, the forward chaining field.

The Parse Service Routine builds a four-word PDE to describe an expression value; it has the following format:

Offset decimal	Meaning
0	A pointer to the address string
4	Length3
6	Flags6
7	Reserved
8	Flags7
9	Sign
10	Indirect count
12	A pointer to the next expression value PDE

Pointer to the address string:
contains a pointer to the expression value address string. Contains zero if this PDE was created due to a switch in indirection symbols.

Length3:
contains the length of the expression value address string. The N is not included in this length value.

Flags6:
The Parse Service Routine sets bit 1 to indicate the type of indirect addressing. Bit 1 has no meaning if the indirection count is 0.

Setting	Meaning
`x... ....`	Reserved bit.
`.0.. ....`	A 24-bit indirect address is represented.
`.1.. ....`	A 31-bit indirect address is represented.
`..xx xxxx`	Reserved bits.

Flags7:
The Parse Service Routine sets these flags to indicate the type of expression value. X'00' indicates that this PDE was not created for an expression value.

Bit setting	Hex	Meaning
`0000 0000`	00	This PDE was created due to a change in indirection symbols.
`0000 0100`	04	This is a decimal expression value.

Bit setting	Hex	Meaning
0000 0010	02	This is a hexadecimal expression value.

Sign:
contains the arithmetic sign character used before the expression value defined by the next expression value PDE. If the sign field is zero and the pointer to the next expression value PDE is non-zero, then the next expression value PDE was created due to a switch in indirection symbols (?% or %?). If there are no more PDEs, then this field is zero.

Indirect count:
contains a value representing the number of levels of indirect addressing within this particular address expression.

Pointer to the next expression value PDE:
contains a pointer to the next expression value PDE if one is present; contains X'FF000000' if this is the last expression value PDE.

Each time parse encounters a *%?* sequence, the current PDE is completed with the 31-bit indirection bit off and the count of 24-bit indirection symbols placed in that PDE. A new expression value PDE is generated in which the 31-bit indirection bit is on and the address string address, the decimal value bit, and the hexadecimal value bit are all zero. The number of consecutive 31-bit indirection symbols is placed in the latter PDE.

Each time parse encounters a *?%* sequence, a complementary process takes place. Specifically, the 31-bit indirection bit is on and the count of 31-bit indirection symbols is placed in the PDE. A new expression value PDE is generated in which the 31-bit indirection bit is off and the address string address, the decimal value bit, and the hexadecimal value bit are all zero. The number of consecutive 24-bit indirection symbols is placed in the latter PDE.

Figure 39 on page 116 illustrates the series of PDEs generated by parse when parse finds an address expression containing a mixed sequence of 31-bit and 24-bit indirection symbols. The following series of PDEs are generated for *12R%??%+16n?*, an address expression with mixed indirection symbols:

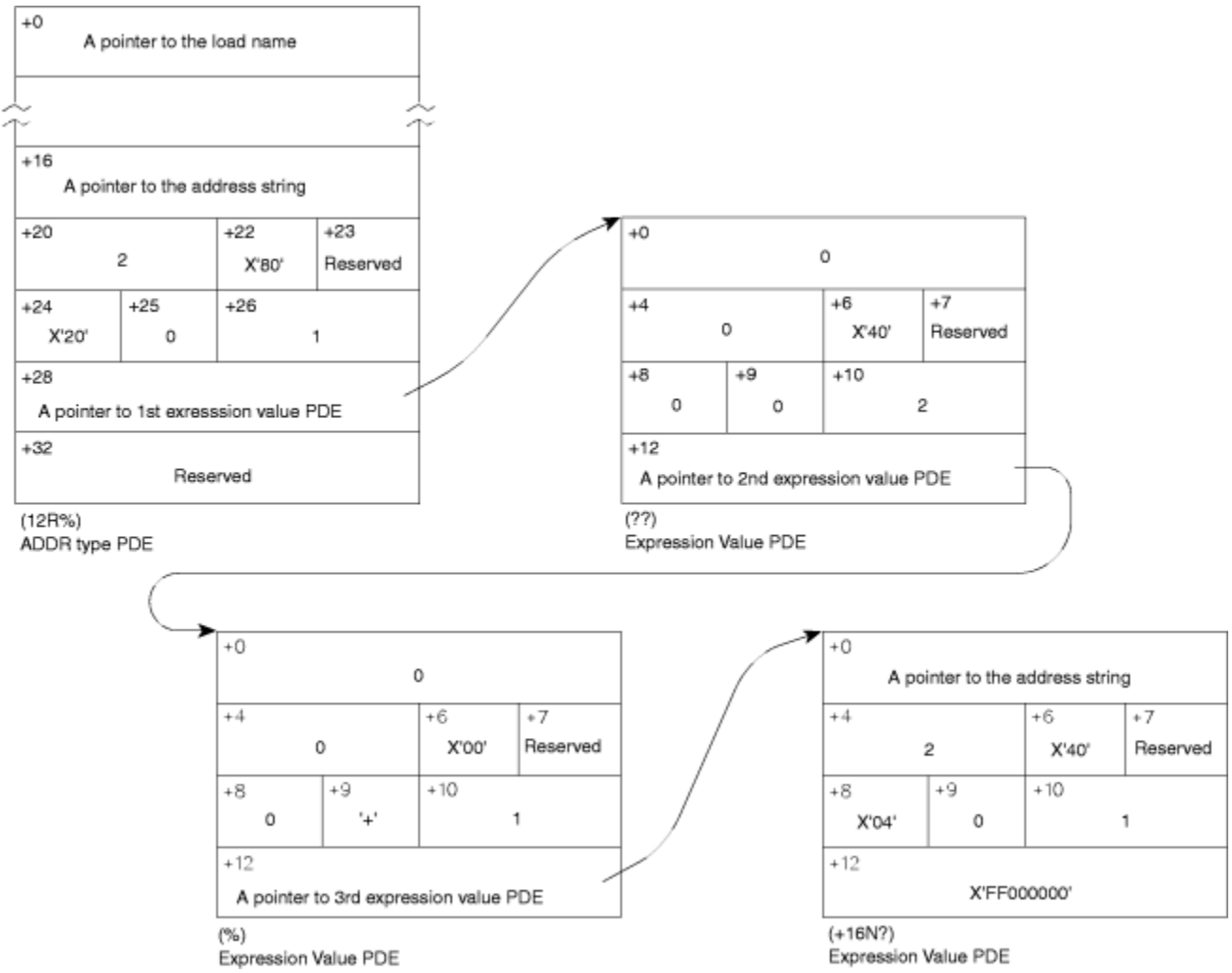


Figure 39. Series of PDEs Created for Mixed Sequence of Indirection Symbols

USERID

The Parse Service Routine builds a four-word PDE to describe a USERID operand; it has the following format:

Offset decimal	Meaning
0	A pointer to the user ID
4	Length1
6	Flags1
7	Reserved
8	A pointer to the password
12	Length2
14	Flags2
15	Reserved

Pointer to the user ID:
contains a pointer to the beginning of the user ID. Contains zero if the user ID was omitted.

Length1:
contains the length of the user ID.

Flags1:

Setting	Meaning
`0... ....`	The user ID is not present.
`1... ....`	The user ID is present.
`.xxx xxxx`	Reserved bits.

Pointer to the password:
contains a pointer to the beginning of the password. Contains zero if the password is omitted.

Length2:
contains the length of the password, excluding the slash.

Flags2:

Setting	Meaning
`0... ....`	The password is not present.
`1... ....`	The password is present.
`.xxx xxxx`	Reserved bits.

USERID8

The Parse Service Routine builds a four-word PDE to describe a USERID8 operand; it has the following format:

Offset decimal	Meaning
0	A pointer to the user ID
4	Length1
6	Flags1
7	Reserved
8	A pointer to the password
12	Length2
14	Flags2
15	Reserved

Pointer to the user ID:
contains a pointer to the beginning of the user ID. Contains zero if the user ID was omitted.

Length1:
contains the length of the user ID.

Flags1:

Setting	Meaning
`0... ....`	The user ID is not present.
`1... ....`	The user ID is present.
`.xxx xxxx`	Reserved bits.

Pointer to the password:
contains a pointer to the beginning of the password. Contains zero if the password is omitted.

Length2:
contains the length of the password, excluding the slash.

Flags2:

Setting	Meaning
`0... ....`	The password is not present.
`1... ....`	The password is present.
`.xxx xxxx`	Reserved bits.

UID2PSWD

The Parse Service Routine builds a six-word PDE to describe a UID2PSWD operand. It has the following format:

Offset decimal	Meaning
0	A pointer to the user ID
4	Length1
6	Flags1
7	Reserved
8	A pointer to password1
12	Length2
14	Flags2
15	Reserved
16	A pointer to password2
20	Length3
22	Flag3
23	Reserved

Pointer to the user ID:
contains a pointer to the beginning of the user ID. It contains zero if the user ID was omitted.

Length1:
contains the length of the user ID.

Flags1:

Setting	Meaning
`0... ....`	The user ID is not present.
`1... ....`	The user ID is present.
`.xxx xxxx`	Reserved bits.

Pointer to password1:
contains a pointer to the beginning of password1. It contains zero if the password1 is omitted.

Length2:
contains the length of password1, excluding the slash.

Flags2:

Setting	Meaning
`0... ....`	Password1 is not present.
`1... ....`	Password1 is present.
`.xxx xxxx`	Reserved bits.

Pointer to password2:
contains a pointer to the beginning of password2. It contains zero if password2 is omitted.

Length3:
contains the length of password2, excluding the slash.

Flags3:

Setting	Meaning
`0... ....`	Password2 is not present.
`1... ....`	Password2 is present.
`.xxx xxxx`	Reserved bits.

UID82PWD

The Parse Service Routine builds a six-word PDE to describe a UID82PWD operand. It has the following format:

Offset decimal	Meaning
0	A pointer to the user ID
4	Length1
6	Flags1
7	Reserved
8	A pointer to password1
12	Length2
14	Flags2
15	Reserved
16	A pointer to password2
20	Length3
22	Flag3
23	Reserved

Pointer to the user ID:
contains a pointer to the beginning of the user ID. It contains zero if the user ID was omitted.

Length1:
contains the length of the user ID.

Flags1:

Setting	Meaning
`0... ....`	The user ID is not present.
`1... ....`	The user ID is present.
`.xxx xxxx`	Reserved bits.

Pointer to password1:
contains a pointer to the beginning of password1. It contains zero if the password1 is omitted.

Length2:
contains the length of password1, excluding the slash.

Flags2:

Setting	Meaning
`0... ....`	Password1 is not present.
`1... ....`	Password1 is present.
`.xxx xxxx`	Reserved bits.

Pointer to password2:
contains a pointer to the beginning of password2. It contains zero if password2 is omitted.

Length3:
contains the length of password2, excluding the slash.

Flags3:

Setting	Meaning
0...	Password2 is not present.
1...	Password2 is present.
.xxx xxxx	Reserved bits.

PDEs created for positional operands described by IKJTERM

CONSTANT

The Parse Service Routine builds a five-word PDE to describe a CONSTANT operand. The PDE has the following format:

Offset decimal	Meaning
0	Length1
1	Length2
2	Reserved
4	Reserved Word Number
6	Flags
8	A pointer to the string of digits
12	A pointer to the exponent
16	A pointer to the decimal point

Length1:
contains the length of the term entered, depending on the type of operand entered as follows:

- For a fixed-point numeric literal, the length includes the digits but not the sign or decimal point.
- For a floating-point numeric literal, the length includes the mantissa (string of digits preceding the letter E) but not the sign or decimal point.
- For a non-numeric literal, the length includes the string of characters but not the apostrophes.

Length2:
For a floating-point numeric literal, length2 contains the length of the string of digits following the letter E but not the sign.

Reserved Word Number:
The reserved word number contains the number of the IKJNAME macro that corresponds to the entered name.

Note: The possible names of reserved words are given by coding a list of IKJNAME macros following an IKJRSVWD macro. One IKJNAME macro is needed for each possible name. If the name entered does not correspond to one of the names in the IKJNAME macro list then parse sets this field to zero.

Flags:
Byte 1:

Setting	Meaning
0...	The operand is missing.
1...	The operand is present.
.1..	Constant.

Setting	Meaning
..1.	Variable.
...1	Statement number.
.... 1...	Fixed-point numeric literal.
.... .1..	Non-numeric literal.
.... ..1.	Figurative constant.
....1	Floating-point numeric literal.

Byte 2:

Setting	Meaning
0...	Sign on constant is either plus or omitted.
1...	Sign on constant is minus.
.0..	Sign on exponent of floating-point numeric literal is either plus or omitted.
.1..	Sign on exponent of floating-point numeric literal is minus.
..1.	Decimal point is present.
...x xxxx	Reserved bits.

Pointer to the string of digits:
contains a pointer to the string of digits, not including the sign if entered. Contains zero if a constant type of operand is not entered.

Pointer to the exponent:
contains a pointer to the string of digits in a floating-point numeric literal following the letter E, not including the sign if entered.

Pointer to the decimal point:
contains a pointer to the decimal point in a fixed-point or floating-point numeric literal. If a decimal point is not entered, this field is zero.

STATEMENT NUMBER

The Parse Service Routine builds a five-word PDE to describe a STATEMENT NUMBER operand. The PDE has the following format:

Offset decimal	Meaning
0	Length1
1	Length2
2	Length3
3	Reserved
4	Reserved
6	Flags
8	A pointer to the program-id
12	A pointer to the line number
16	A pointer to the verb number

Length1:
contains the length of the program-id specified but does not include the following period. Contains zero if the program-id is not present.

Length2:
contains the length of the line number entered but does not include the delimiting periods. Contains zero if the line number is not present.

Length3:
contains the length of the verb number entered but does not include the preceding period. Contains zero if the verb number is not present.

Flags:
Byte 1:

Setting	Meaning
0...	The operand is missing.
1...	The operand is present.
.1..	Constant.
..1.	Variable.
...1	Statement number.
.... xxxx	Reserved.

Byte 2:

- Reserved.

Pointer to the program-id:
contains a pointer to the program-id, if entered. Contains zero if not present.

Pointer to the line number:
contains a pointer to the line number, if entered. Contains zero if not present.

Pointer to the verb number:
contains a pointer to the verb number, if entered. Contains zero if not present.

VARIABLE

The Parse Service Routine builds a five-word PDE to describe a VARIABLE operand. The PDE has the following format:

Offset decimal	Meaning
0	A pointer to the data-name
4	Length1
5	Reserved
6	Flags
7	Reserved
8	A pointer to the PDE for the first qualifier
12	A pointer to the program-id name
16	Length2
17	Number of Qualifiers
18	Number of Subscripts
19	Reserved

Pointer to the data-name:
contains a pointer to the data-name. If a program-id qualifier precedes the data-name, this pointer points to the first character after the period of the program-id qualifier.

Length1:
contains the length of the data-name.

Flags:
Byte 1:

Setting	Meaning
`0... ....`	The operand is missing.
`1... ....`	The operand is present.
`.1.. ....`	Constant.
`..1. ....`	Variable.
`...1 ....`	Statement number.
`.... xxxx`	Reserved.

Pointer to the PDE for the first qualifier:
contains a pointer to the PDE describing the first qualifier of the data-name, if any. This field contains X'FF000000' if no qualifiers are entered.

Note: The format of the PDE for a data-name qualifier follows this description.

Pointer to the program-id name:
contains a pointer to the program-id name, if entered. This field contains zero if the optional program-id name is not present.

Length2:
contains the length of the program-id name, if entered. Contains zero if the optional program-id name is not present.

Number of Qualifiers:
contains the number of qualifiers entered for this data-name. (For example, if data-name A of B is entered, this field would contain 1.)

Number of Subscripts:
contains the number of subscripts entered for this data-name. (For example, if data-name A(1,2) is entered, this field would contain 2.)

The format of a data-name qualifier is:

Offset decimal	Meaning
0	A pointer to the data-name qualifier
4	Length
5	Reserved
6	Reserved
7	Reserved
8	A pointer to the PDE for the next qualifier

Pointer to the data-name qualifier:
contains a pointer to the data-name qualifier.

Length:
contains the length of the data-name qualifier.

Pointer to the PDE for the next qualifier:
contains a pointer to the PDE describing the next qualifier, if any. This field contains X'FF000000' for the last qualifier.

The PDE created for expression operands described by IKJOPER

The Parse Service Routine builds a two-word PDE to describe an EXPRESSION operand. The PDE has the following format:

Offset decimal	Meaning
0	Reserved
4	Reserved

Offset decimal	Meaning
6	Flags
7	Reserved

Flags:

Setting	Meaning
`0... ....`	The entire operand (expression) is missing.
`1... ....`	The entire operand (expression) is present.
`.xxx xxxx`	Reserved.

The PDE created for reserved word operands described by IKJRSVWD

The Parse Service Routine builds a two-word PDE to describe a RESERVED WORD operand. The PDE has the following format:

Offset decimal	Meaning
0	Reserved
2	Reserved-word number
4	Reserved
6	Flags
7	Reserved

Note: This PDE is not used when the IKJRSVWD macro instruction is chained from an IKJTERM macro instruction. In this case, the reserved-word number is returned in the CONSTANT parameter PDE built by the IKJTERM macro instruction.

Reserved-word number:
The reserved-word number contains the number of the IKJNAME macro instruction that corresponds to the entered name.

Note: You indicate the possible names of reserved words by coding a list of IKJNAME macros following an IKJRSVWD macro. One IKJNAME macro is needed for each possible name. If the name entered does not correspond to one of the names in the IKJNAME macro list, parse sets this field to zero.

Flags:
Byte1:

Setting	Meaning
`0... ....`	The operand is missing.
`1... ....`	The operand is present.
`.xxx xxxx`	Reserved.

The PDE created for positional operands described by IKJIDENT

The Parse Service Routine builds a two-word PDE to describe a non-delimiter-dependent positional operand; it has the following format:

Offset decimal	Meaning
0	A pointer to the positional operand
4	Length
6	Flags
7	Reserved

Pointer to the positional operand:
contains a pointer to the beginning of the positional operand. If INTEG was specified on the IKJIDENT macro instruction, this will contain a pointer to a fullword binary value.

Contains zero if the positional operand is omitted.

Length:
contains the length of the positional operand.

Flags:

Setting	Meaning
`0... ....`	The operand is not present.
`1... ....`	The operand is present.
`.xxx xxxx`	Reserved bits.

The PDE created for keyword operands described by IKJKEYWD

Parse builds a halfword (2-byte) PDE to describe a keyword operand; it has the following format:

Offset decimal	Meaning
0	Number

Number:
You describe the possible names for a keyword operand to the parse service routine by coding a list of IKJNAME macro instructions directly following the IKJKEYWD macro instruction. One IKJNAME macro instruction must be executed for each possible name.

The Parse Service Routine places into the PDE a number that relates the keyword name entered to the position of the corresponding IKJNAME macro instruction in the list of IKJNAME macro instructions. For example, if two IKJNAME macro instructions follow the IKJKEYWD macro instruction, and the user has entered the second keyword operand, the Parse Service Routine places a 2 into the PDE.

If the keyword is not entered, and you did not specify a default in the IKJKEYWD macro instruction, the Parse Service Routine places a zero into the PDE.

The PDE created for keyword operands described by IKJUNFLD

The Parse Service Routine builds a halfword (2-byte) PDE for an unidentified keyword operand. Parse does not place a value into the PDE. Because all checking for an unidentified keyword operand is performed in the verify exit routine that is specified on the VERIFCK= operand in the IKJUNFLD macro instruction, all processing is complete by the time parse returns control to its caller.

How the list and range options affect PDE formats

Several factors affect the formats of the IKJPARMD mapping DSECT and the PDEs built by the Parse Service Routine:

- The options you specify in the parse macro instructions
- The type of operand that the user enters.

If you specify the LIST or the RANGE options in the parse macro instructions describing positional operands, the IKJPARMD DSECT and the PDEs returned by the Parse Service Routine are modified to reflect these options.

LIST

The LIST option can be used with the following positional operand types: USERID, DSNAME, DSTHING, ADDRESS, VALUE, CONSTANT, VARIABLE, STATEMENT NUMBER, HEX, INTEG, CHAR, and any non-delimiter-dependent positional operand.

If you specify the LIST option in the parse macro instructions describing the positional operand types listed above, the parse service routine allocates an additional word for the PDE created to describe the positional operand. This word is allocated even though the terminal user cannot actually enter a list. If a list is not entered, this word is set to X'FF000000'. If a list is entered, the additional word is used to chain the PDEs created for each element found in the list.

Each additional PDE has a format identical to the one described for that operand type within the IKJPARMD DSECT. Because the number of elements in a list is variable, the number of PDEs created by the Parse Service Routine is also variable. The chain word of the PDE created for the last element of the list is set to X'FF000000'.

Figure 40 on page 126 shows the PDL returned by the Parse Service Routine after three positional operands have been entered. In this case, the first two operands, a USERID and a STRING operand, had been defined as not accepting lists. The third operand, a VALUE operand, had the LIST option coded in the IKJPOSIT macro instruction that defined the operand syntax. The VALUE operand was entered as a two-element list.

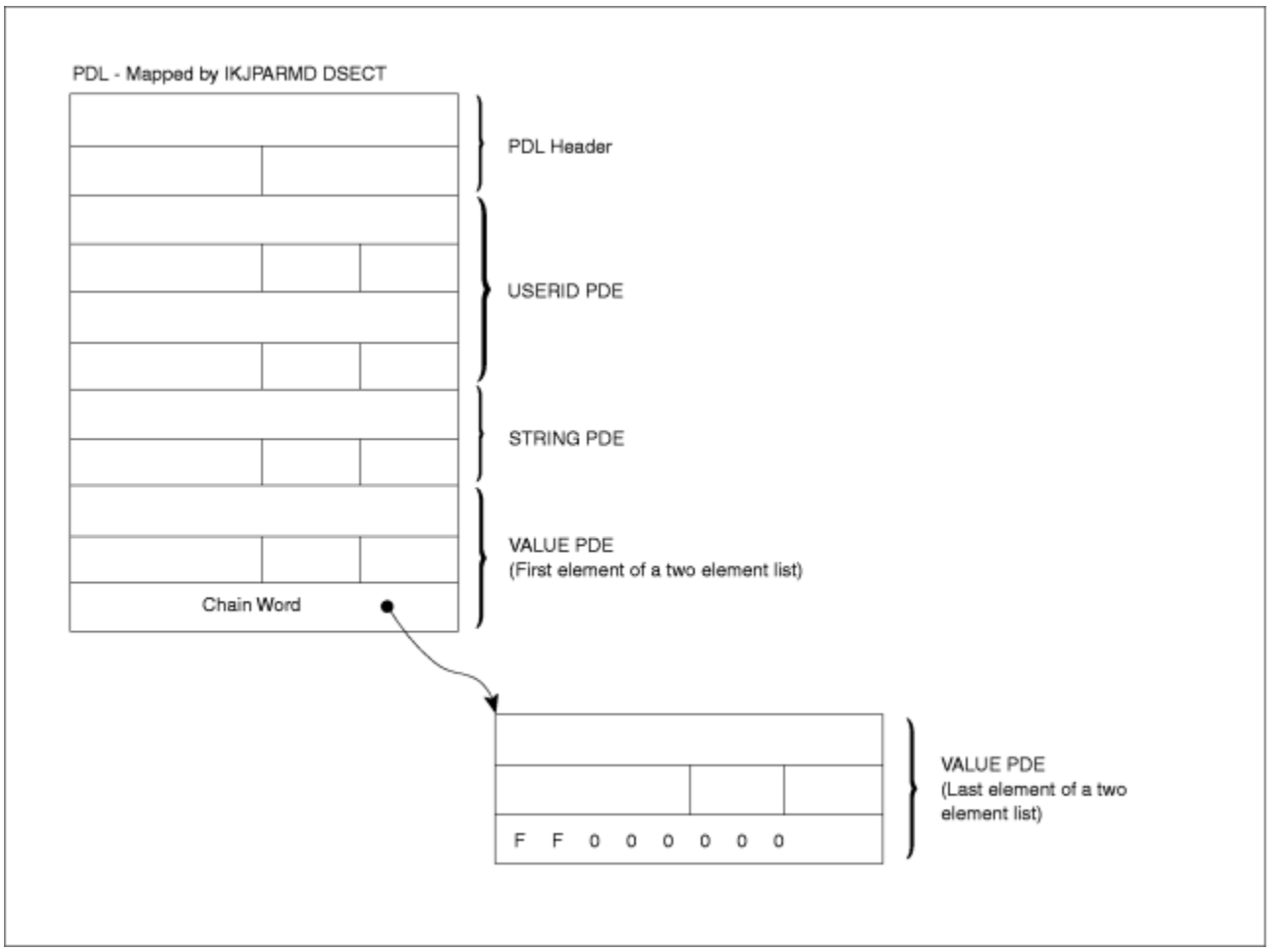


Figure 40. A PDL showing PDEs that describe a list

RANGE

The RANGE option can be used with the following positional operand types: HEX (X'' only), ADDRESS, VALUE, CONSTANT, VARIABLE, STATEMENT NUMBER, INTEG, and any non-delimiter-dependent positional operand.

If you specify the RANGE option in the parse macro instructions describing the positional operand types listed above, the parse service routine builds two identical, sequential PDEs within the PDL returned to the calling routine. Parse allocates space for the second PDE even though the terminal user cannot

actually specify a range. If a range is not supplied, the second PDE is set to zero. The flag bit which is normally set for a missing parameter will also be zero in the second PDE.

Figure 41 on page 127 shows the PDL returned by the Parse Service Routine after two positional operands have been entered. In this case, the first operand is a USERID operand and the second operand is a VALUE operand that had the RANGE option coded in the IKJPOSIT macro instruction that defined the operand syntax. For this example, the VALUE operand was not entered as a range, and, consequently, parse sets the second PDE to zero.

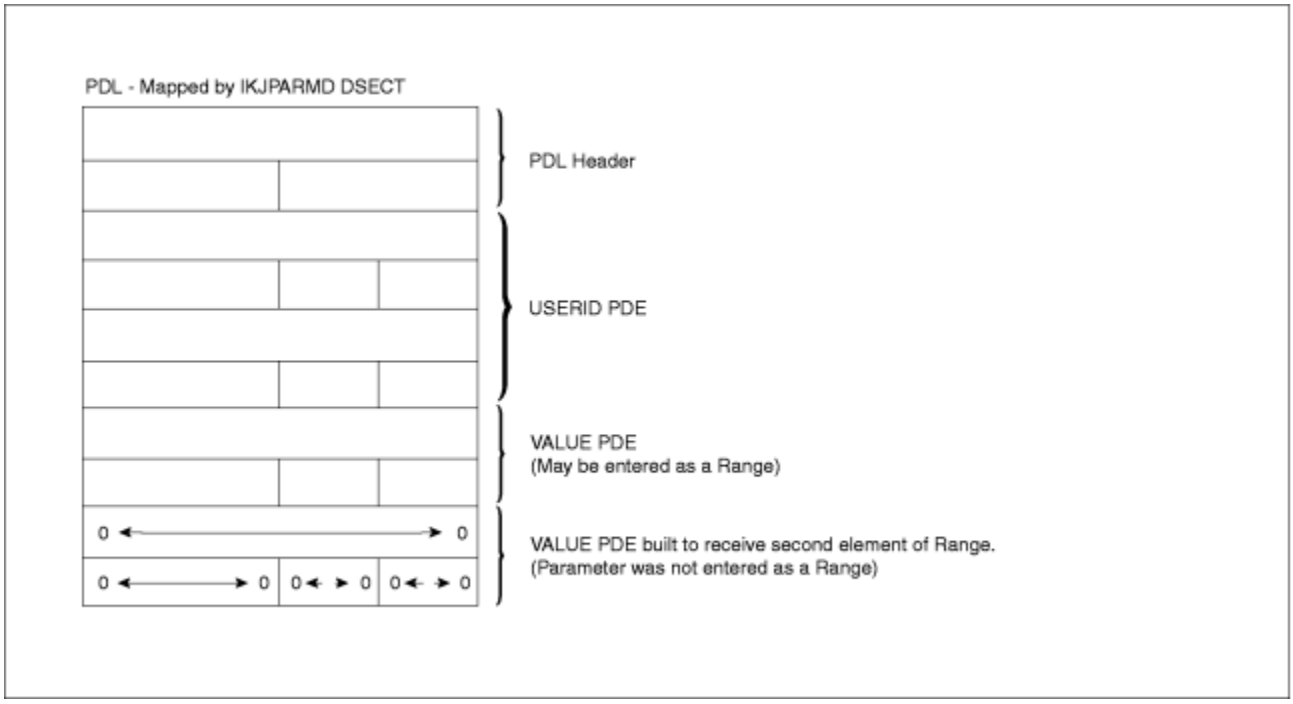


Figure 41. A PDL showing PDEs describing a range

How combining the LIST and RANGE options affects PDE formats

If you specify both the LIST and RANGE options in a parse macro instruction describing a positional operand, the Parse Service Routine builds two identical PDEs within the PDL returned to the calling routine. Both of these PDEs are formatted according to the type of positional operand described. These two PDEs describe the RANGE. Parse appends an additional word to the second PDE to chain any additional PDEs built to describe the LIST.

Figure 42 on page 128 shows this general format.

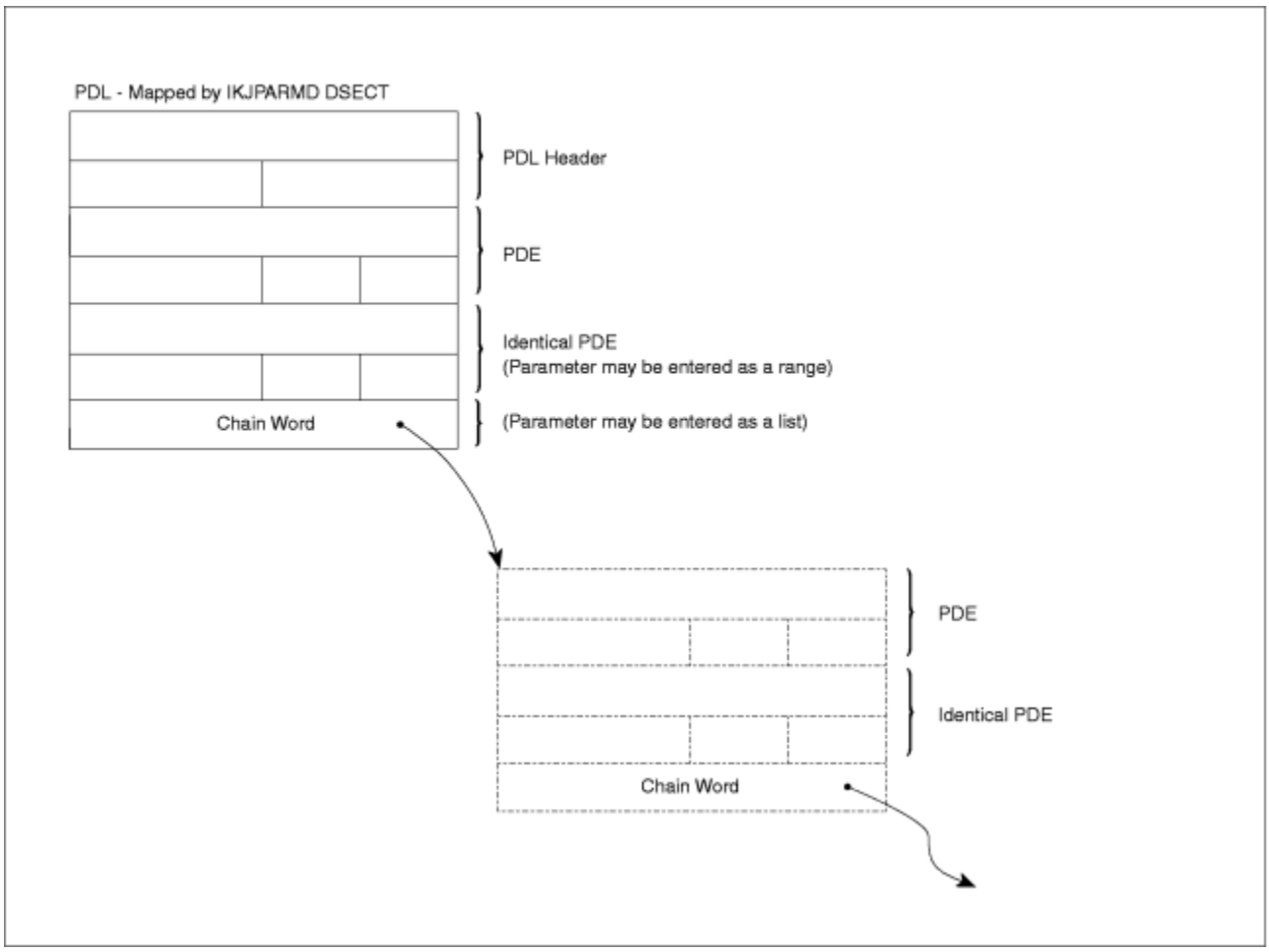


Figure 42. A PDL showing PDEs that describe LIST and RANGE options

If you have specified both the LIST and the RANGE options in the parse macro instruction describing a positional operand, the user at the terminal has the option of supplying a single operand, a single range, a list of operands, or a list of ranges. The construction of the PDL returned by the Parse Service Routine can reflect each of these conditions.

Figure 43 on page 129 shows the PDL returned by the Parse Service Routine if the user enters a single operand.

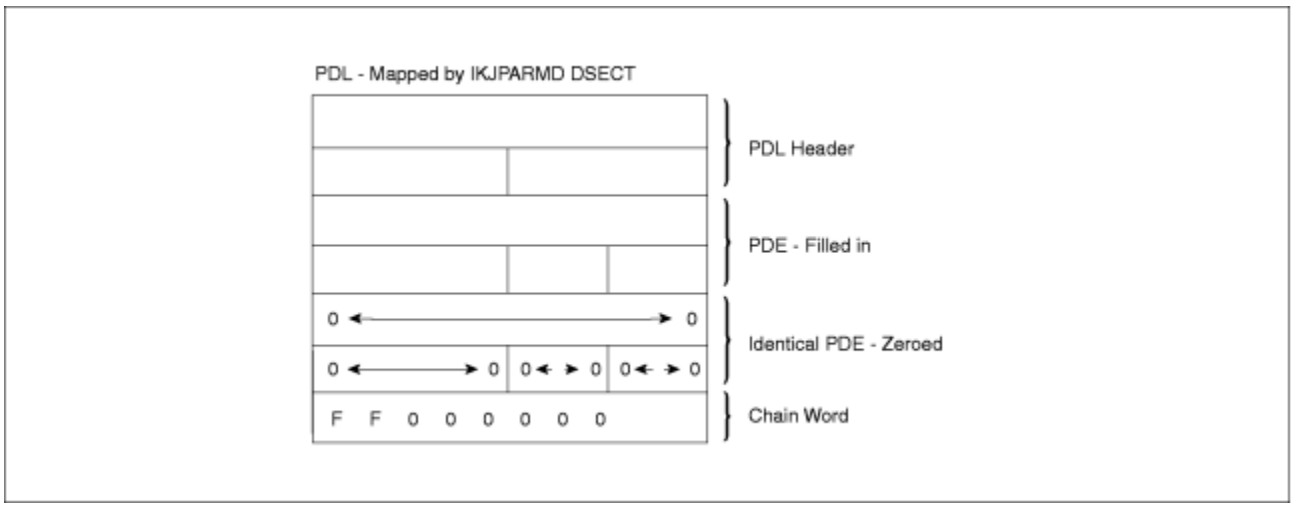


Figure 43. PDL - LIST and RANGE acceptable, single operand entered

As Figure 43 on page 129 shows, the Parse Service Routine sets both the second PDE and the chain word to zero when the LIST and RANGE options were coded in the macro instruction describing the operand, but the user entered a single operand.

Figure 44 on page 129 shows the PDL returned by the Parse Service Routine if the user enters a single range of the form:

```
operand:operand
```

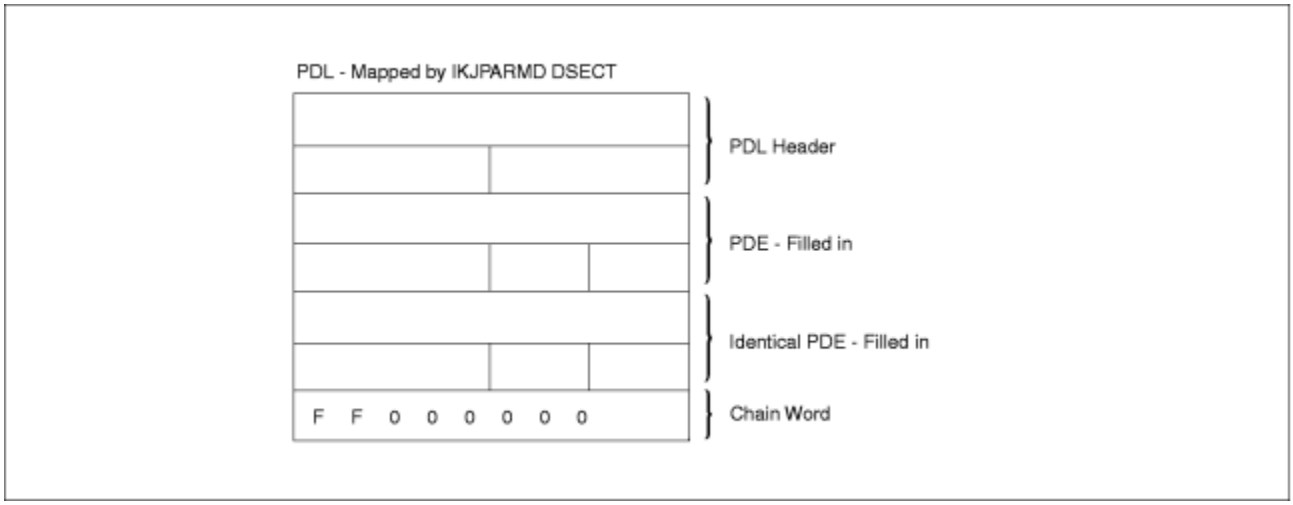


Figure 44. PDL - LIST and RANGE acceptable, single range entered

As Figure 44 on page 129 shows, the Parse Service Routine fills in both PDEs to describe the single RANGE operand entered by the user. The chain word is set to X'FF000000' to indicate that there are no elements chained to this one. (That is, the operand was not entered in the form of a list).

Figure 45 on page 130 shows the format of the PDL returned by the parse service routine if the user enters a list of operands in the form:

```
(operand,operand,…)
```

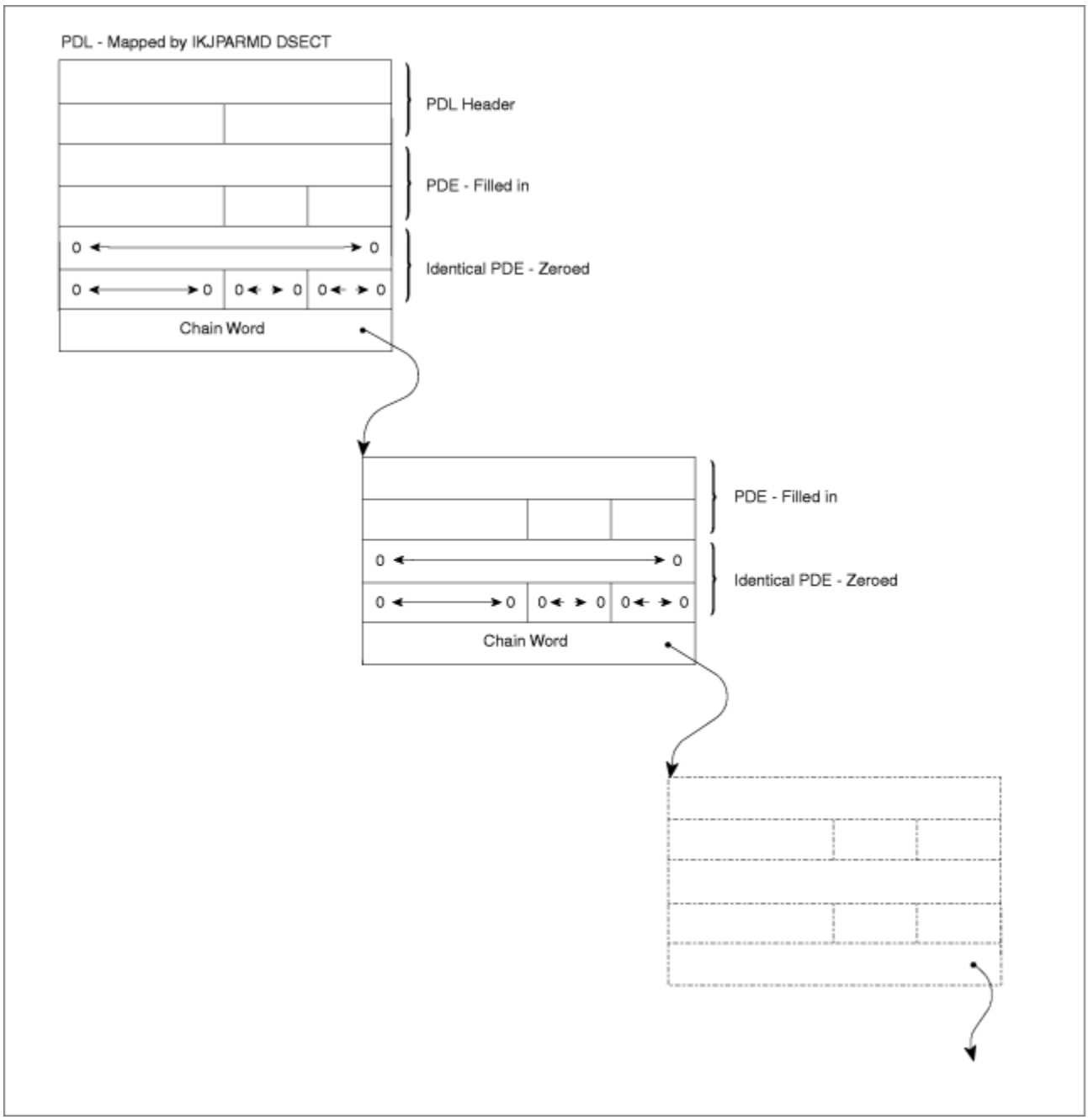


Figure 45. PDL - LIST and RANGE acceptable, LIST entered

As Figure 45 on page 130 shows, the Parse Service Routine fills in each of the first PDEs and the chain word pointers to describe the list of operands entered by the user. The second, identical PDEs are set to zero to indicate that the operand was not entered in the form of a range.

The last set of PDEs on the chain contain X'FF000000' in the chain word to indicate that there are no more PDEs on that particular chain.

The PDL created by the Parse Service Routine to describe an operand entered as a list of ranges is similar to the one created to describe a list. The difference is that the Parse Service Routine fills in the second, identical PDEs to describe the ranges entered.

Figure 46 on page 131 shows the format of the PDL returned by the parse service routine if the user enters a list of ranges in the form:

```
(operand:operand, operand:operand,…)
```

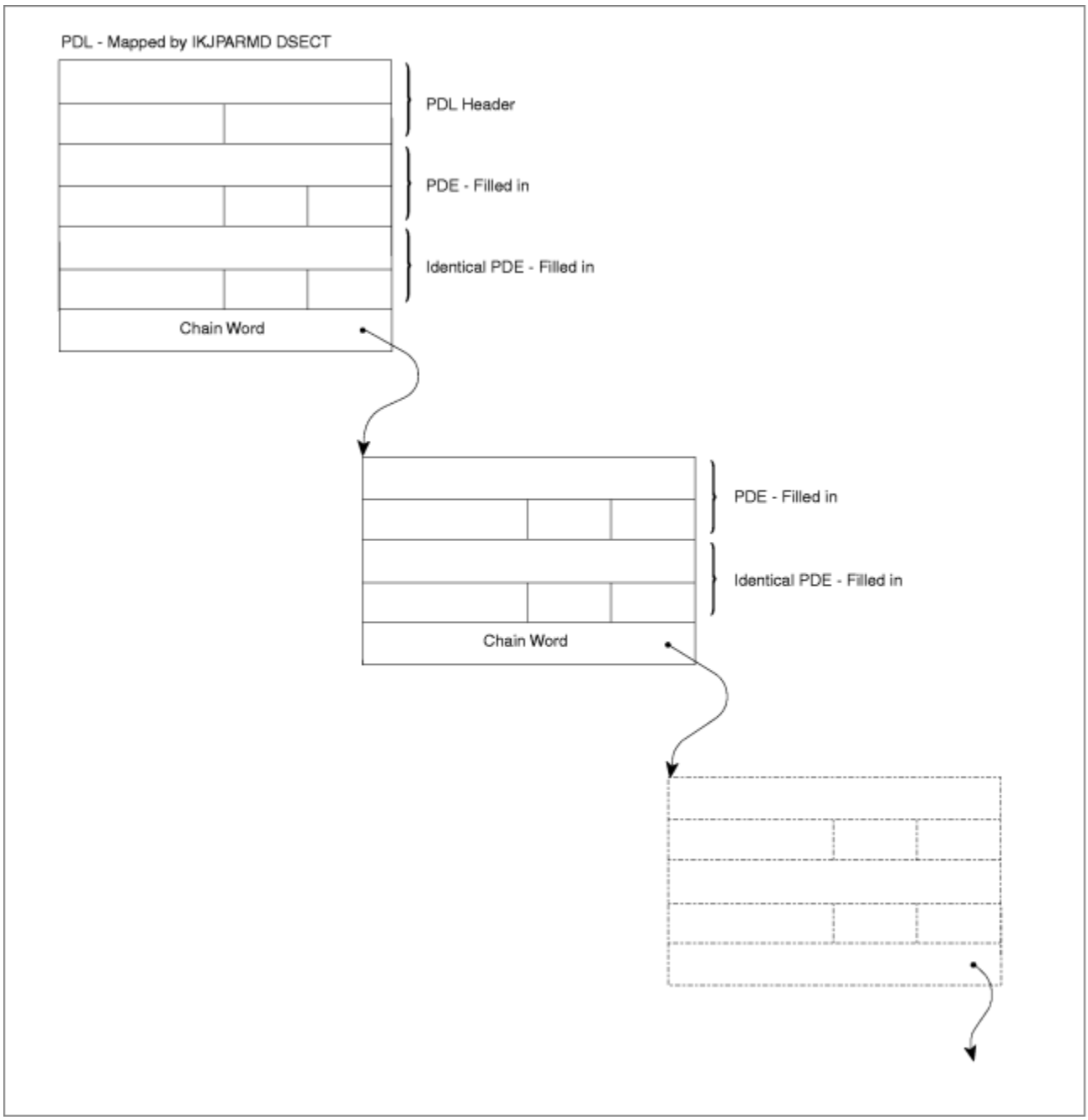


Figure 46. PDL - LIST and RANGE acceptable, list of ranges entered

As Figure 46 on page 131 shows, the Parse Service Routine fills in each of the second, identical PDEs to describe the ranges entered. The chain words are also filled in to point down through the list of parameters entered.

The last set of PDEs on the chain contain X'FF000000' in the chain word to indicate that there are no more PDEs on that particular chain.

Examples using the Parse Service Routine

Example 1: Describing a PROCESS command syntax

This example expands upon "Example 1: Describing a PROCESS command syntax" on page 96. This example shows how the parse macro instructions could be used within a Command Processor to describe the syntax of a PROCESS command to the Parse Service Routine. A sample Command Processor that includes the parse macros used in this example is shown in *z/OS TSO/E Programming Guide*.

The sample PROCESS command we are describing to the parse service routine has the following format:

```
PROCESS     dsname     [ ACTION   ]
                       [ NOACTION ]
```

Figure 47 on page 132 shows the sequence of parse macro instructions that describe the syntax of this PROCESS command to the Parse Service Routine. The parse macro instructions used in this example build the parameter control list (PCL) describing the syntax of the PROCESS command operands. The macro instructions also create the DSECT that you use to map the parameter descriptor list returned by the parse service routine. In this example, the name of the DSECT is PRDSECT.

```
PCLDEFS  IKJPARM  DSECT=PRDSECT
DSNPCE   IKJPOSIT DSNAME,                                              X
               PROMPT='THE NAME OF THE DATA SET YOU WANT TO PROCESS.    X
               ENTER ''?'' FOR HELP',                                   X
               HELP=('A DATA SET NAME WHICH HAS A FIRST-LEVEL QUALIFIER  X
                OTHER THAN ''SYS1''.'),                                 X
               VALIDCK=POSITCHK
ACTPCE   IKJKEYWD DEFAULT='NOACTION'
         IKJNAME  'ACTION'
         IKJNAME  'NOACTION'
         IKJENDP
```

Figure 47. Example 1 - using parse macros to describe command operand syntax

Figure 48 on page 132 shows the IKJPARMD DSECT created by the expansion of the parse macro instructions.

```
PRDSECT DSECT
         DS    2A
DSNPCE   DS    6A
ACTPCE   DS    H
```

Figure 48. Example 1 - The PRDSECT DSECT created by parse

If a terminal user entered the PROCESS command described in this example in the form:

```
process myid.data noation
```

the Parse Service Routine would prompt the terminal user with:

```
INVALID KEYWORD, NOATION
REENTER THIS OPERAND -
```

The user at the terminal might respond with:

```
NOACTION
```

The Parse Service Routine would then complete the scan of the command parameters, build a parameter descriptor list (PDL), place the address of the PDL into the fullword pointed to by PPLANS, and return to the calling program.

The calling routine uses the address of the PDL as a base address for the PRDSECT DSECT.

Figure 49 on page 133 shows the PDL returned by the parse service routine. The symbolic addresses within the PRDSECT DSECT are shown to the left of the PDL at the points within the PDL to which they apply, and the meanings of the fields within the PDL are explained to the right of the PDL.

Note: Only the macros IKJIDENT, IKJKEYWD, and IKJPOSIT return a label in the DSECT.

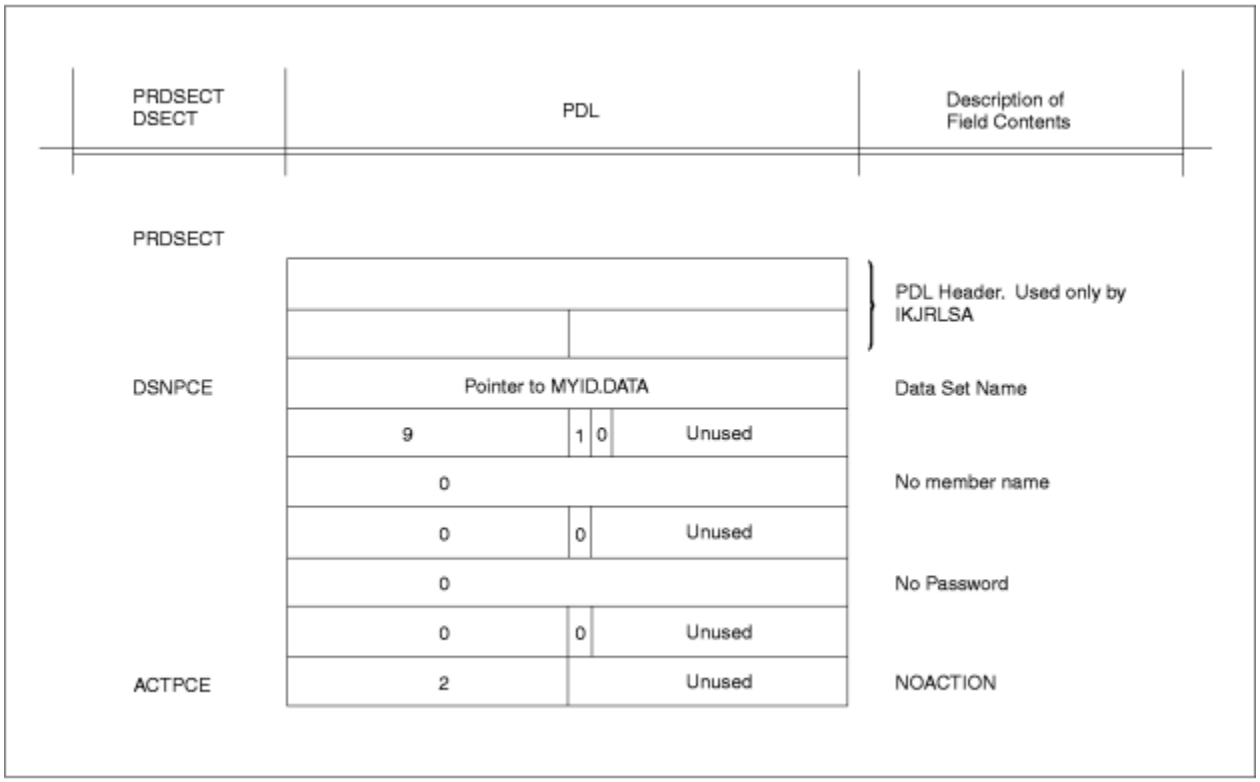


Figure 49. Example 1 - The PRDSECT DSECT and the PDL

Example 2: Describing an EDIT command syntax

This example expands upon “Example 2: Describing an EDIT command syntax” on page 97. This example shows how the parse macro instructions could be used within a Command Processor to describe the syntax of an EDIT command to the Parse Service Routine.

The sample EDIT command we are describing to the parse service routine has the following format:

```
EDIT       dsname
           [ PLI [([number  [number]]   [CHAR60 )]]]
           [     [ [  2     [  72  ]]   [CHAR48  ]]]
           [ FORT                                  ]
           [ ASM                                   ]
           [ TEXT                                  ]
           [ DATA                                  ]

           [ SCAN  ]
           [ NOSCAN]

           [ NUM  ]
           [ NONUM]

           [ BLOCK(number)  ]
           [ BLKSIZE(number)]

            LINE(number)
```

Figure 50 on page 134 shows the sequence of parse macro instructions that describe the syntax of this EDIT command to the Parse Service Routine. The parse macro instructions used in this example build the

parameter control list (PCL) describing the syntax of the EDIT command operands. The macro instructions also create the DSECT that you use to map the parameter descriptor list returned by the Parse Service Routine. In this example, the name of the DSECT defaults to IKJPARMD.

Note: Only the macros IKJIDENT, IKJKEYWD, and IKJPOSIT return a label in the DSECT.

```
PARMTAB  IKJPARM
DSNAME   IKJPOSIT   DSNAME,PROMPT='DATA SET NAME'
TYPE     IKJKEYWD
         IKJNAME    'PL1',SUBFLD=PL1FLD
         IKJNAME    'FORT'
         IKJNAME    'ASM'
         IKJNAME    'TEXT'
         IKJNAME    'DATA'
SCAN     IKJKEYWD   DEFAULT='NOSCAN'
         IKJNAME    'SCAN'
         IKJNAME    'NOSCAN'
NUM      IKJKEYWD   DEFAULT='NUM'
         IKJNAME    'NUM'
         IKJNAME    'NONUM'
BLOCK    IKJKEYWD
         IKJNAME    'BLOCK',SUBFLD=BLOCKSUB,ALIAS='BLKSIZE'
LINE     IKJKEYWD
         IKJNAME    'LINE',SUBFLD=LINESIZE
PL1FLD   IKJSUBF
PL1COL1  IKJIDENT   'NUMBER',FIRST=NUMERIC,OTHER=NUMERIC,DEFAULT='2'
PL1COL2  IKJIDENT   'NUMBER',FIRST=NUMERIC,OTHER=NUMERIC,DEFAULT='72'
PL1TYPE  IKJKEYWD   DEFAULT='CHAR60'
         IKJNAME    'CHAR60'
         IKJNAME    'CHAR48'
BLOCKSUB IKJSUBF
BLKNUM   IKJIDENT   'NUMBER',FIRST=NUMERIC,OTHER=NUMERIC,              X
               PROMPT='BLOCKSIZE',MAXLNTH=8
LINESIZE IKJSUBF
LINNUM   IKJIDENT   'NUMBER',FIRST=NUMERIC,OTHER=NUMERIC,              X
               PROMPT='LINESIZE'
         IKJENDP
```

Figure 50. Example 2 - using parse macros to describe command operand syntax

Figure 51 on page 134 shows the IKJPARMD DSECT created by the expansion of the parse macro instructions.

Note: Only the macros IKJIDENT, IKJKEYWD, and IKJPOSIT return a label in the DSECT.

```
IKJPARMD DSECT
         DS    2A
DSNAME   DS    6A
TYPE     DS    H
SCAN     DS    H
NUM      DS    H
BLOCK    DS    H
BLKSIZE  DS    0H
LINE     DS    H
PL1COL1  DS    2A
PL1COL2  DS    2A
PL1TYPE  DS    H
BLKNUM   DS    2A
LINNUM   DS    2A
```

Figure 51. Example 2 - The IKJPARMD DSECT created by parse

If a terminal user entered the EDIT command described in this example in the form:

```
edit sysfile/x pl1(3) nonum block
```

the Parse Service Routine would prompt for the blocksize as follows:

```
ENTER BLOCKSIZE
```

The user at the terminal might respond with:

```
160
```

The Parse Service Routine would then complete the scan of the command parameters, build a parameter descriptor list (PDL), place the address of the PDL into the fullword pointed to by PPLANS, and return to the calling program.

The calling routine uses the address of the PDL as a base address for the IKJPARMD DSECT.

Figure 52 on page 136 shows the PDL returned by the Parse Service Routine. The symbolic addresses within the IKJPARMD DSECT are shown to the left of the PDL at the points within the PDL to which they apply, and the meanings of the fields within the PDL are explained to the right of the PDL.

Note: Only the macros IKJIDENT, IKJKEYWD, and IKJPOSIT return a label in the DSECT.

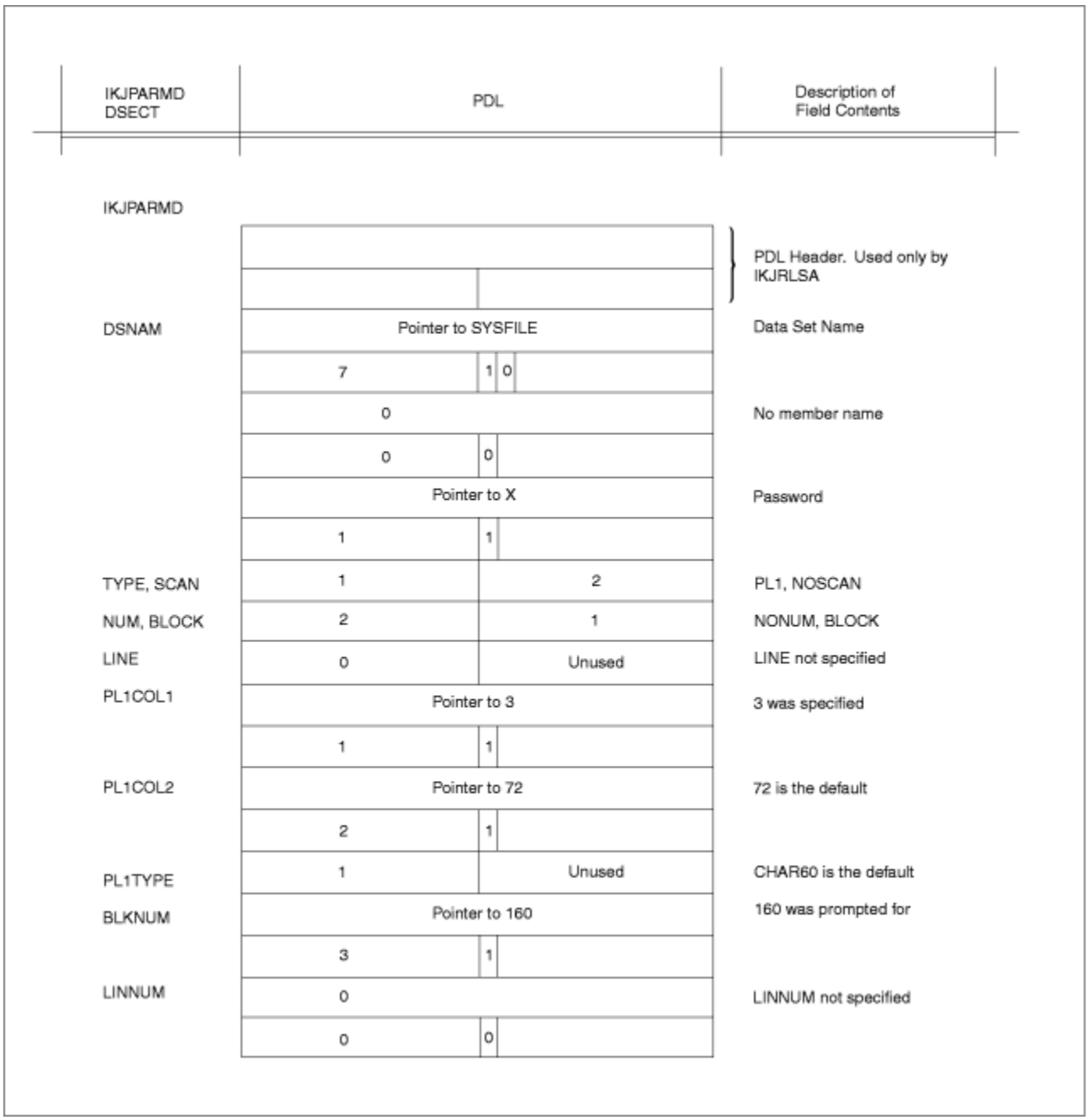


Figure 52. Example 2 - The IKJPARMD DSECT and the PDL

Example 3: Describing an AT command syntax

This example expands upon “Example 3: Describing an AT command syntax” on page 98. This example shows how the parse macro instructions could be used to describe the syntax of a sample AT command that has the following syntax:

```
      [stmt               ]
AT    [(stmt-1,stmt-2,...)] (cmd chain) COUNT(integer)
      [stmt-3:stmt-4      ]
```

Figure 53 on page 137 shows the sequence of parse macro instructions that describe this sample AT command to the Parse Service Routine. The parse macro instructions used in this example build the

parameter control list (PCL) describing the syntax of the AT command operands. The macro instructions also create the DSECT that you can use to map the parameter descriptor list returned by the parse service routine. In this example, the name of the DSECT is PARSEAT.

Note: Only the macros IKJIDENT, IKJKEYWD, and IKJPOSIT return a label in the DSECT.

```
EXAM2    IKJPARM    DSECT=PARSEAT
STMTPCE  IKJTERM    'STATEMENT NUMBER',UPPERCASE,LIST,RANGE,TYPE=STMT, X
               VALIDCK=CHKSTMT
POSITPCE IKJPOSIT   PSTRING,HELP='CHAIN OF COMMANDS',VALIDCK=CHKCMD
KEYPCE   IKJKEYWD
NAMEPCE  IKJNAME    'COUNT',SUBFLD=COUNTSUB
COUNTSUB IKJSUBF
IDENTPCE IKJIDENT   'COUNT',FIRST=NUMERIC,OTHER=NUMERIC,               X
               VALIDCK=CHKCOUNT
         IKJENDP
```

Figure 53. Example 3 - using parse macros to describe command operand syntax

Figure 54 on page 137 shows the PARSEAT DSECT created by the expansion of the parse macro instructions.

Note: Only the macros IKJIDENT, IKJKEYWD, and IKJPOSIT return a label in the DSECT.

```
PARSEAT  DSECT
         DS    2A
         DS    11A
POSITPCE DS    2A
KEYPCE   DS    H
IDENTPCE DS    2A
```

Figure 54. Example 3 - the PARSEAT DSECT created by parse

In this example, if the terminal user entered the AT command incorrectly as:

```
at 200/3 (list all) count(a)
```

the Parse Service Routine would prompt the terminal user with the message:

```
INVALID STATEMENT NUMBER, 200/3
REENTER
```

The user might respond with:

```
200.3
```

The Parse Service Routine would then prompt the user with:

```
INVALID COUNT, a
REENTER
```

The user might respond with:

```
3
```

This sequence resulted in the syntactically correct command of:

```
at 200.3 (list all) count(3)
```

The Parse Service Routine would then build a parameter descriptor list (PDL) and place the address of the PDL into PPLANS.

The Parse Service Routine then returns to the caller and the caller uses the address of the PDL as a base address for the PARSEAT DSECT.

Figure 55 on page 138 shows the PDL returned by the parse routine. The symbolic addresses of the PARSEAT DSECT are shown to the left of the PDL at the points within the PDL to which they apply. A description of the fields within the PDL is shown on the right.

Note: Only the macros IKJIDENT, IKJKEYWD, and IKJPOSIT return a label in the DSECT.

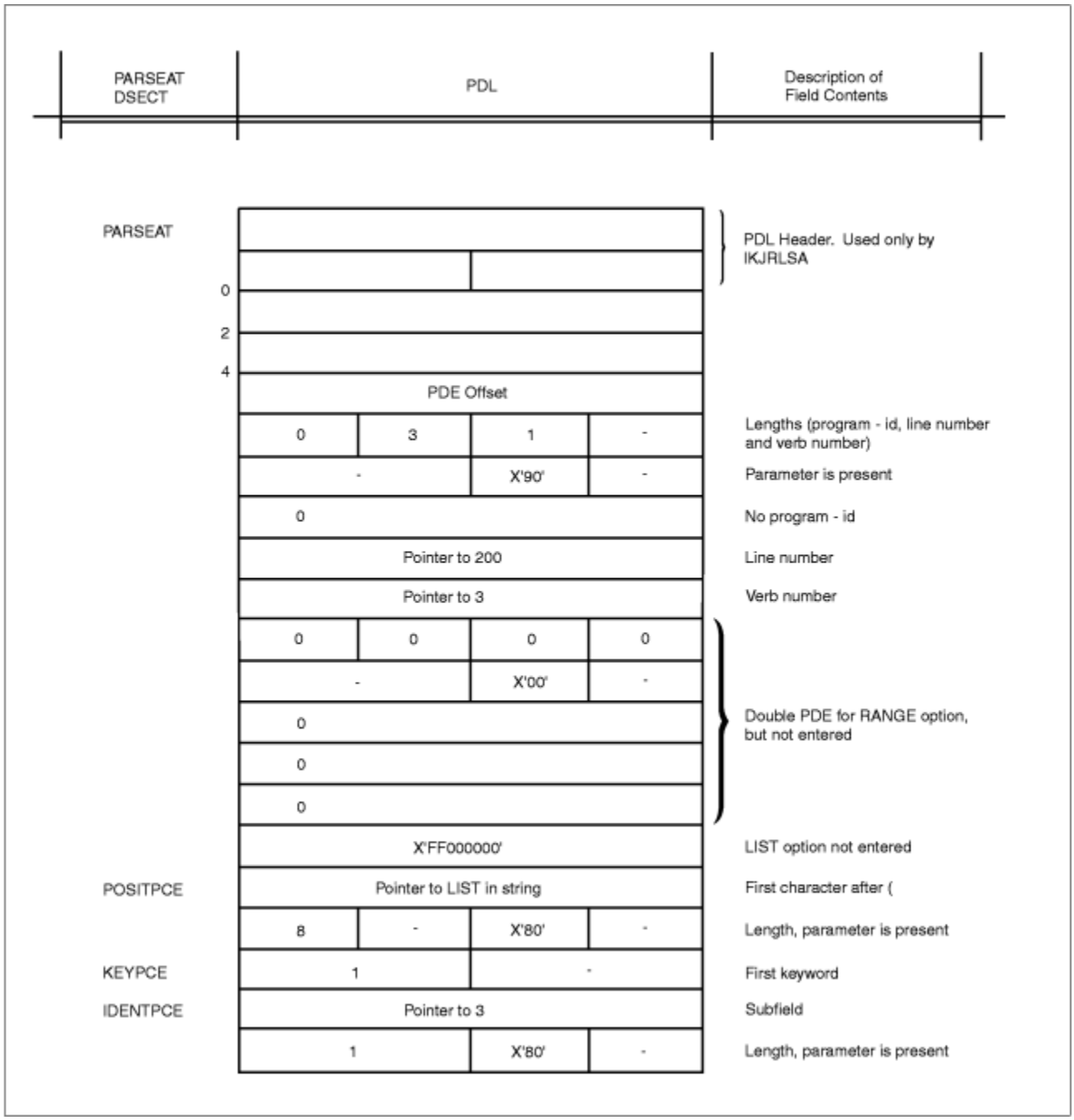


Figure 55. Example 3 - the PARSEAT DSECT and the PDL

Example 4: Describing a LIST command syntax

This example expands upon "Example 4: Describing a LIST command syntax" on page 99. This example shows how the parse macro instructions could be used to describe the syntax of a sample LIST command that has the following syntax:

```
LIST         symbol PRINT(symbol)
```

Figure 56 on page 139 shows the sequence of parse macro instructions that describe this sample LIST command to the Parse Service Routine. The parse macro instructions used in this example build the parameter control list (PCL) describing the syntax of the LIST command operands. The macro instructions also create the DSECT that you use to map the parameter descriptor list returned by the parse service routine. In this example, the name of the DSECT is PARSELST.

Note: Only the macros IKJIDENT, IKJKEYWD, and IKJPOSIT return a label in the DSECT.

```
EXAM3    IKJPARM    DSECT=PARSELST
VARPCE   IKJTERM    'SYMBOL',UPPERCASE,PROMPT='SYMBOL',TYPE=VAR,       X
               VALIDCK=CHECK,SBSCRPT=SUBPCE
SUBPCE   IKJTERM    'SUBSCRIPT',SBSCRPT,TYPE=CNST,PROMPT='SUBSCRIPT'
KEYPCE   IKJKEYWD
NAMEPCE  IKJNAME    'PRINT',SUBFLD=PRINTSUB
PRINTSUB IKJSUBF
         IKJTERM    'SYMBOL-2',UPPERCASE,PROMPT='SYMBOL-2',TYPE=VAR
IKJENDP
```

Figure 56. Example 4 - Using Parse Macros to Describe Command Operand Syntax

Figure 57 on page 139 shows the PARSELST DSECT created by the expansion of the parse macro instructions.

Note: Only the macros IKJIDENT, IKJKEYWD, and IKJPOSIT return a label in the DSECT.

```
PARSELST DSECT
         DS    2A
         DS    5A
         DS    15A
KEYPCE   DS    H
         DS    11A
```

Figure 57. Example 4 - The PARSELST DSECT

In this example, if the terminal user entered the LIST command incorrectly as:

```
list a of 1 in 3(1) print(d)
```

the Parse Service Routine would prompt the terminal user with:

```
INVALID SYMBOL, a...1 in 3(1)
REENTER
```

The user might respond with:

```
a of b in 3(1)
```

The Parse Service Routine would then prompt with:

```
INVALID SYMBOL, a...3(1)
REENTER
```

The user might respond with:

```
a of b in c(1)
```

This sequence resulted in the syntactically correct command of:

```
list a of b in c(1) print(d)
```

The Parse Service Routine would then build a parameter descriptor list (PDL) and place the address of the PDL into the fullword pointed to by PPLANS.

The Parse Service Routine then returns to the caller and the caller uses the address of the PDL as a base address for the PARSELST DSECT.

Figure 58 on page 141 shows the PDL returned by the Parse Service Routine. The symbolic addresses of the PARSELST DSECT are shown to the left of the PDL at the points within the PDL to which they apply. A description of the fields within the PDL is shown on the right.

Note: Only the macros IKJIDENT, IKJKEYWD, and IKJPOSIT return a label in the DSECT.

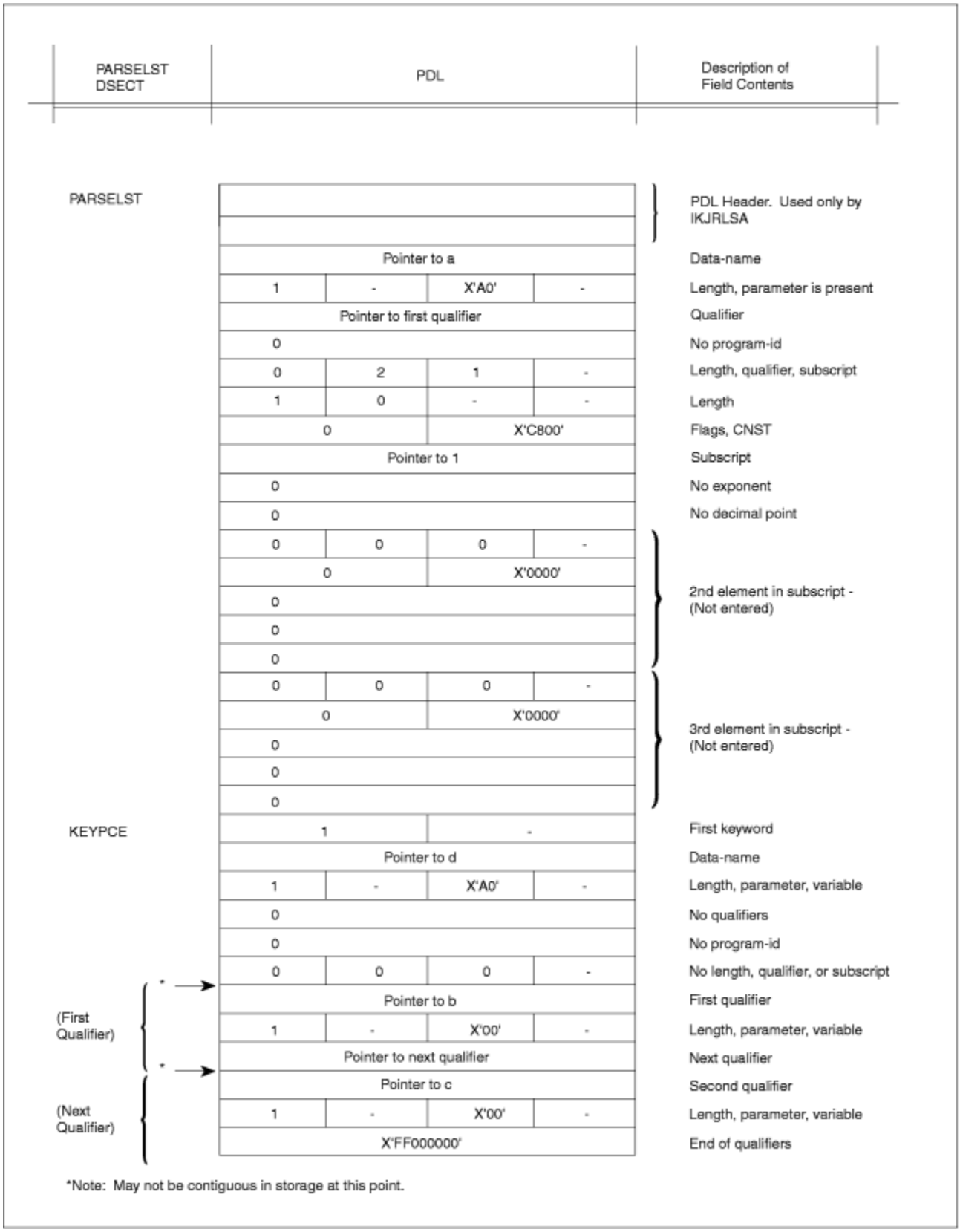


Figure 58. Example 4 - The PARSELST DSECT and the PDL

Example 5: Describing a WHEN command syntax

This example expands upon "Example 5: Describing a WHEN command syntax" on page 99. This example shows how the parse macro instructions could be used to describe the syntax of a sample WHEN command that has the following syntax:

```
WHEN     {addr      } (subcommand chain)
         {expression}
```

Figure 59 on page 142 shows the sequence of parse macro instructions that describe this sample WHEN command to the parse service routine. The parse macro instructions used in this example build the parameter control list (PCL) describing the syntax of the WHEN command operands. The macro instructions also create the DSECT that you use to map the parameter descriptor list returned by the parse service routine. In this example, the name of the DSECT is PARSEWHN.

Note: Only the macros IKJIDENT, IKJKEYWD, and IKJPOSIT return a label in the DSECT.

```
EXAM4    IKJPARM    DSECT=PARSEWHN
OPER     IKJOPER    'EXPRESSION',OPERND1=SYMBOL1,OPERND2=SYMBOL2,      X
                RSVWD=OPERATOR,CHAIN=ADDR1,PROMPT='TERM',VALICHK=CHECK
SYMBOL1  IKJTERM    'SYMBOL1',UPPERCASE,TYPE=VAR,PROMPT='SYMBOL2'
OPERATOR IKJRSVWD   'OPERATOR',PROMPT='OPERATOR'
         IKJNAME    'EQ'
         IKJNAME    'NEQ'
SYMBOL2  IKJTERM    'SYMBOL2',TYPE=VAR
ADDR1    IKJTERM    'ADDRESS',TYPE=VAR,VALIDCK=CHECK1
LASTONE  IKJPOSIT   PSTRING,VALIDCK=CHECK2
         IKJENDP
```

Figure 59. Example 5 - using parse macros to describe command operand syntax

Figure 60 on page 142 shows the PARSELST DSECT created by the expansion of the parse macro instructions.

Note: Only the macros IKJIDENT, IKJKEYWD, and IKJPOSIT return a label in the DSECT.

```
PARSEWHN DSECT
         DS    2A
         DS    2A
         DS    5A
         DS    2A
         DS    5A
         DS    5A
LASTONE  DS    2A
```

Figure 60. Example 5 - the PARSEWHN DSECT

In this example, if the terminal user entered the WHEN command incorrectly as:

```
when (a) (list b)
```

the Parse Service Routine would prompt the terminal user with:

```
ENTER OPERATOR
```

The user might then respond:

```
eq
```

The Parse Service Routine would then prompt with:

```
INVALID EXPRESSION, (a eq)
REENTER
```

The user might respond then with:

```
(a eq b)
```

This sequence resulted in a syntactically correct command of:

```
when (a eq b) (list b)
```

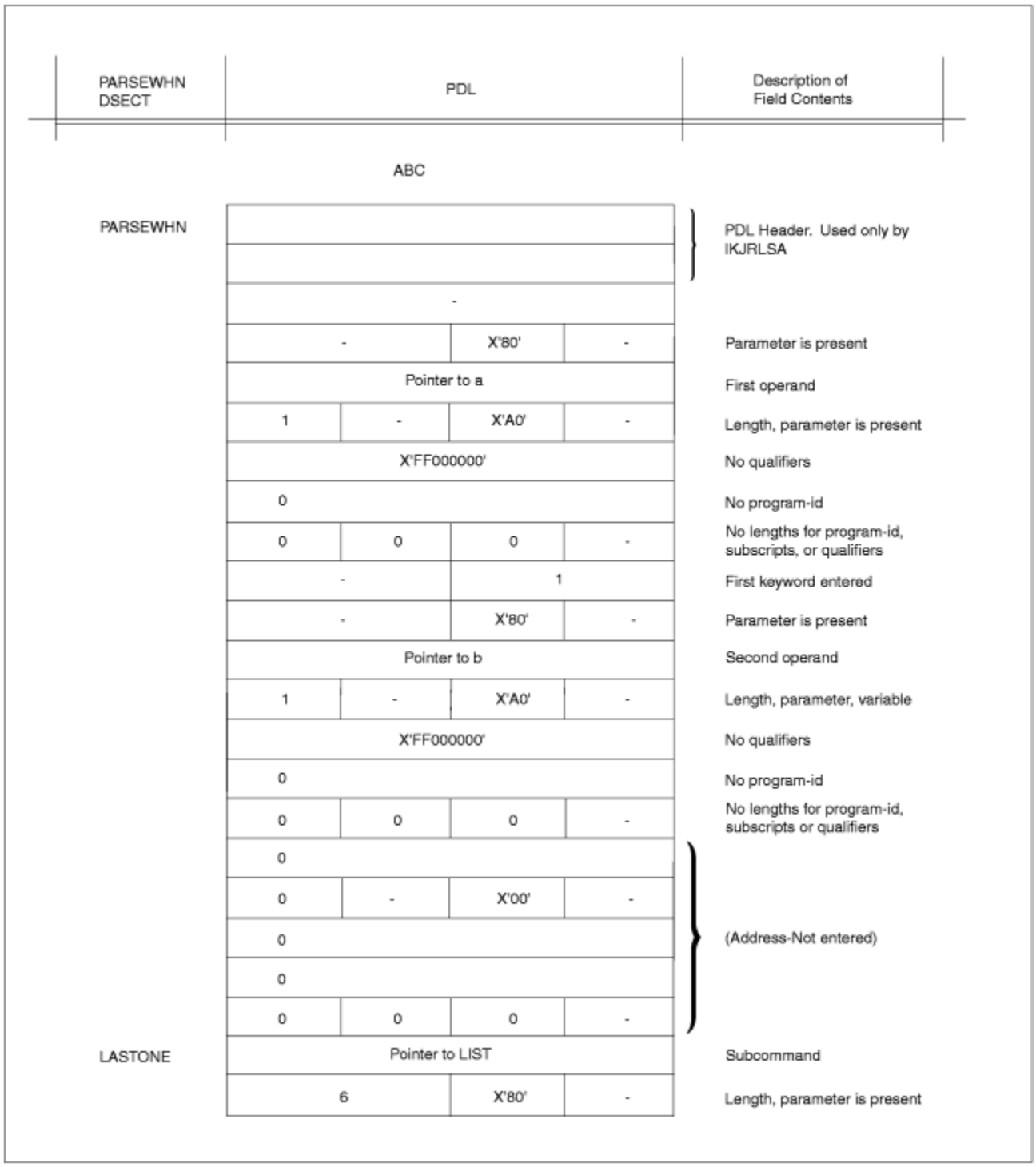
The Parse Service Routine would then build a parameter descriptor list (PDL) and place the address of the PDL into the fullword pointed to by PPLANS.

The Parse Service Routine then returns to the caller and the caller uses the address of the PDL as a base address for the PARSEWHN DSECT.

Note: Only the macros IKJIDENT, IKJKEYWD, and IKJPOSIT return a label in the DSECT.

Figure 61 on page 144 shows the PDL returned by the Parse Service Routine. The symbolic addresses of the PARSEWHN DSECT are shown to the left of the PDL at the points within the PDL to which they apply. A description of the fields within the PDL is shown on the right.

Note: Only the macros IKJIDENT, IKJKEYWD, and IKJPOSIT return a label in the DSECT.



PARSEWHN DSECT	PDL				Description of Field Contents
	ABC				
PARSEWHN					PDL Header. Used only by IKJRLSA
	-				
	-		X'80'	-	Parameter is present
	Pointer to a				First operand
	1	-	X'A0'	-	Length, parameter is present
	X'FF000000'				No qualifiers
	0				No program-id
	0	0	0	-	No lengths for program-id, subscripts, or qualifiers
	-		1		First keyword entered
	-		X'80'	-	Parameter is present
	Pointer to b				Second operand
	1	-	X'A0'	-	Length, parameter, variable
	X'FF000000'				No qualifiers
	0				No program-id
	0	0	0	-	No lengths for program-id, subscripts or qualifiers
	0				
	0	-	X'00'	-	
	0				(Address-Not entered)
	0				
	0	0	0	-	
LASTONE	Pointer to LIST				Subcommand
	6		X'80'	-	Length, parameter is present

Figure 61. Example 5 - the PARSEWHN DSECT and PDL

Chapter 7. Using the terminal control macro instructions

Functions of the terminal control macro instructions

Use the following macro instructions in your Command Processor to control terminal functions and attributes.

macro instruction	Function	Valid under z/OSMF ISPF	Issue in 24-bit addressing mode
GTDEVSIZ	Get device size	x	x
GTSIZE	Get terminal line size	x	
GTTERM	Get terminal attributes		x
RTAUTOPT	Restart automatic line numbering or character prompting		
SPAUTOPT	Stop automatic line numbering or character prompting		
STATTN	Set Attention Simulation		x
STAUTOCP	Start automatic character prompting		
STAUTOLN	Set automatic line numbering		x
STBREAK	Set Break		x
STCC	Specify Terminal Control Characters		x
STCLEAR	Set Display Clear Character String		x
STCOM	Set Inter-Terminal Communication		x
STFSMODE	Set full-screen mode		x
STLINENO	Set line number		x
STSIZE	Set terminal line size		x
STTIMEOU	Set Time Out Feature		x
STTMPMD	Set terminal display manager options		x
STTRAN	Set Character Translation		x
TCLEARQ	Clear buffers		x

Except for the GTDEVSIZE, GTSIZE, and GTTERM macros, all terminal control macros will be ignored under z/OSMF ISPF. GTTERM fails with RC 4 to indicate the terminal does not support full screen TPUT. GTDEVSIZE and GTSIZE return the screen size for ISPF.

GTDEVSIZ - Get Device Size

Use the GTDEVSIZ macro instruction to determine the current logical line size and the number of lines of a user's terminal. This macro returns both values regardless of whether the terminal type is display or non-display. See the description of the GTSIZE macro, which you can use to obtain the screen length for display stations only.

When GTDEVSIZ is issued in a time-sharing environment, the logical line size of the user's terminal, which is the maximum number of characters per line, is returned in register 1. The logical screen length, which is the number of lines per display, is returned in register 0. If there is no maximum number of lines, register 0 contains all zeros.

The GTDEVSIZ macro is applicable only in a VTAM time-sharing environment. It is ignored if VTAM is not active when the macro instruction is issued.

Figure 62 on page 146 shows the format of the GTDEVSIZ macro instruction.

```
[symbol]      GTDEVSIZ
```

Figure 62. The GTDEVSIZ macro instruction

When control is returned to the user, register 15 contains one of the following return codes:

Table 34. Return codes from GTDEVSIZ

Return code dec(Hex)	Meaning
0(0)	Successful. The contents of registers 0 and 1 are described above.
4(4)	A parameter was specified. No parameter should be specified.

GTSIZE - Get Terminal Line Size

Use the GTSIZE macro instruction to determine the current logical line size of the user's terminal. When GTSIZE is executed in the background, it returns a line size of 132 characters and a screen size of 0 lines. If the terminal is a display station, use the GTSIZE macro instruction to determine the size of the display screen. See the description of the GTDEVSIZ macro, which you can use to obtain the screen length for both display and non-display stations.

When the GTSIZE macro instruction is issued in a time sharing environment, the logical line size of the user's terminal, which is the maximum number of characters per line, is returned in register 1. If the terminal is a display station, the line size is returned in register 1 and the screen length, which is the maximum number of lines per display, is returned in register 0. If the terminal is an LU1 device type, register 0 contains all zeros. The GTSIZE macro instruction is ignored if TSO/E is not active when the macro instruction is issued.

Figure 63 on page 146 shows the format of the GTSIZE macro instruction.

```
[symbol]      GTSIZE
```

Figure 63. The GTSIZE macro instruction

When control is returned to the user, register 15 contains one of the following return codes:

Table 35. Return codes from GTSIZE	
Return code dec(Hex)	**Meaning**
0(0)	Successful. The contents of registers 0 and 1 are described above.
4(4)	A parameter was specified. No parameter should be specified.

GTTERM - Get Terminal Attributes

Use the GTTERM macro instruction to determine the primary (default) and the alternate screen sizes for a 3270 display terminal. Use the ERASE/WRITE command (X'F5') to erase the screen, to set the screen size mode to primary mode, and optionally to write data to the screen. Use the ERASE/WRITE ALTERNATE command (X'7E') to erase the screen, to set the screen size mode to the alternate mode, and optionally to write data to the screen. Figure 64 on page 147 shows the format of the GTTERM macro instruction.

```
symbol      GTTERM     PRMSZE=addr  [,ALTSZE=addr] [,MF= {L           }]
                                                   [     {(E,ctraddr) }]
                       [,ATTRIB=addr] [,TERMID=addr]
```

Figure 64. The GTTERM macro instruction

PRMSZE=*addr*
specifies the address of a 2-byte area into which GTTERM returns the primary row value in the high-order byte and the primary column value in the low-order byte.

ALTSZE=*addr*
specifies the address of a 2-byte area into which GTTERM returns the alternate row value in the high-order byte and the alternate column value in the low-order byte.

ATTRIB=*addr*
specifies the address of a 1-word field into which GTTERM returns terminal attributes. The contents of this field are described below:

Byte	Setting	Meaning
0	xxxx xxxx	Reserved.
1	0...	The terminal does not support double-byte character set (DBCS).
1	1...	The terminal supports DBCS.
1	.000 0000	American English (default).
1	.000 0001	American English.
1	.001 0001	Katakana.
2	xxxx	Reserved.
2	.... 00..	The ASCII-7 device code identifier.
2	.... 01..	The ASCII-8 device code identifier.
2	xx	Reserved.
3	1...	This is a VTAM TSB[1].
3	.1..	Break features are not allowed[1].
3	..1.	The translate table is in use[1].
3	...1	The default translate table is in use[1].

Byte	Setting	Meaning
3	.... 1...	Display in full-screen mode[1].
3	.... .x..	Reserved.
3	.... ..0.	The device supports EBCDIC code.
3	.... ..1.	The device supports ASCII code.
3	0	The Read Partition (Query) is not supported.
3	1	The Read Partition (Query) is supported.
[1] These bits are returned only for VTAM applications.		

MF=L | (E,*ctrl addr*)
indicates the form of the GTTERM macro instruction.

L
specifies the list form.

(E,*ctrl addr*)
specifies the execute form and the address of the list form.

TERMID=*addr*
addr specifies one of the following:

- 16-byte area into which GTTERM returns the terminal name in the first eight bytes and the network ID in the second eight bytes.
- 39-byte area with CODEPG in the first six bytes, where GTTERM will return the terminal name, the network ID, the IP address (IPv4 or IPv6), port number, and code page (CGCSGID) information.
- 52-byte area with IPADD6 in the first six bytes, where GTTERM will return the IP address (IPv4 or IPv6), port number, and zone identifier if requested.
- 310-byte area with DOMIP6 in the first six bytes, where GTTERM will return the domain name, terminal name, IP address, port number, and zone identifier if requested.

Tip: If CODEPAGE=YES is coded in the TSOKEY00 member of SYS1.PARMLIB, TSO/VTAM will query the terminal during the logon process to obtain the code page information. This enables the Display Code Page function for the client.

GTTERM returns the following when addr is a 52-byte area with IPADD6 specified in the first 6 bytes:

Offset Dec (Hex)	Type	Length in bytes	Description
0(0)	Character	8	Terminal name
8(8)	Character	8	Network ID
16(10)	Character	16 4	IPv6 address IPv4 address
32(20)	Character	2	Port number in hex
34(22)	Character	1 1... .... .1.. .... ..1. ....1 1111	Flag byte On - indicates IPv6 address at offset 16 On - indicates IPv4 address at offset 16 On - zone id truncated Not used
35(23)	Byte	1	Length of zone id at offset 36

Offset Dec (Hex)	Type	Length in bytes	Description
36(24)	Character	16	Zone id

This is true for all TSO with Telnet sessions. GTTERM clears the IP address and port number area if it was not a Telnet session.

The user program can request the return of domain name, IP address (IPv4 or IPv6) and port number for Telnet sessions on GTTERM macro by specifying the keyword DOMIP6 in the first six bytes of the terminal ID area.

IPADDR and DOMAIN in the terminal id are deprecated (they are still valid but not suggested). If you are using IPADDR in the terminal id, and the IP address for the tn3270 client is an IPv6 address, you will receive X'FFFFFFFF' in the IP address field that is returned.

In this case, *addr* specifies the address of a field at least 310 bytes in length. GTTERM returns the information as follows:

Offset dec (Hex)	Type	Length in bytes	Description
0(0)	Character	8	Terminal name
8(8)	Character	8	Network ID
16(10)	Character	16 4	IPv6 address IPv4 address
32(20)	Character	2	Port number in hex
34(22)	Character	1 1... .... 0... .... .1.. .... ..1. ....1 1111	Flag byte 1 On - truncated domain name Off - complete domain name On - Indicates IPv6 address at offset 16 On - Indicates IPv4 address at offset 16 On - Indicates zone id truncated Not used
35(23)	Character	1	Flag byte 2 (not used)
36(24)	Character	2	Length of domain name
38(26)	Character	255	Domain name
293(126)	Integer	1	Length of zone id
294(127)	Character	16	Zone id

Note: The length of the zone id will be zero if the zone id is not available.

This is true for all TSO with Telnet sessions. GTTERM clears the IP address, port number, and domain name area if it was not a Telnet session.

GTTERM returns the following when addr is a 39 byte area with CODEPG specified in the first 6 bytes.

Offset Dec (Hex)	Type	Length in Bytes	Description
0 (0)	Character	8	Terminal name

Offset Dec (Hex)	Type	Length in Bytes	Description
8 (8)	Character	8	Network ID
16 (10)	Character	16 4	IPv6 address IPv4 address
32(20)	Character	2	Port number in hex
34(22)	Character	1	Flag byte
		1...	On - indicates IPv6 address at offset 16
		.1..	On - indicates IPv4 address at offset 16
		..11 1111	Not used
35 (23)	Character	2	Character Set
37 (25)	Character	2	Code Page

Note: The terminal or emulator sets the CGCSGID values. Consult the vendor of the terminal or emulator regarding CGCSGID value documentation. Also see chapter 5 of the *3174 Character Set Reference* (GA27-3831) for common CGCSGID values.

If you use the list form of the GTTERM macro, the coded parameters expand into the parameter list shown in Table 36 on page 150.

Table 36. Parameter list expansion for the list form of GTTERM

Offset dec (Hex)	Number of bytes	Meaning
0(0)	4	Address of halfword to receive primary screen size.
4(4)	4	Address of halfword to receive alternate screen size.
8(8)	4	Address of word to receive Device Query supported flag.
12(C)	4	Address of one of following: • 16-byte field to receive terminal name • 52-byte field to receive terminal name, IP address, port number, and zone identifier is requested • 39-byte field to receive the terminal name, network ID, IP address (IPv4 or IPv6), port number, and code page (CGCSGID) information. • 310-byte field to receive terminal name, IP address, port number, domain name, and zone identifier is requested

When control is returned to the user, register 15 contains one of the following return codes:

Table 37. Return codes from GTTERM

Return code dec (Hex)	Meaning
0(0)	Successful.
4(4)	The terminal in use does not support full screen TPUT.

Table 37. Return codes from GTTERM (continued)	
Return code dec (Hex)	**Meaning**
8(8)	The terminal in use is not a display terminal.
12(C)	The PRMSZE parameter, which is required, was not specified.

RTAUTOPT - Restart Automatic Line Numbering or Character Prompting

Use the RTAUTOPT macro instruction to restart either the automatic line numbering feature or the automatic character prompting feature. These features are suspended when the terminal user causes an attention interruption or enters a null line of input. Because only one of these features can be used at a time, the restarted feature is the one that was suspended. See "STAUTOLN - Start Automatic Line Numbering" on page 153. for a description of the automatic line numbering feature and "STAUTOCP - Start Automatic Character Prompting" on page 152 for a description of the automatic character prompting feature.

When this macro instruction is used to restart automatic line numbering, the first line number assigned after line numbering is restarted is the same line number that would have been assigned to the next line of terminal input if automatic line numbering had not been suspended.

If your application program is creating a line numbered data set, use the STAUTOLN macro to specify the starting numbers when restarting automatic line numbering. This will ensure that the application's numbers are still in synchronization with the system's.

The RTAUTOPT macro instruction can be used only in a time sharing environment. If you issue this macro when TSO/E is not active or when your program is running under Session Manager, it is ignored.

Figure 65 on page 151 shows the format of the RTAUTOPT macro instruction.

```
[symbol]      RTAUTOPT
```

Figure 65. The RTAUTOPT macro instruction

When control is returned to the user, register 15 contains one of the following return codes:

Table 38. Return codes from RTAUTOPT	
Return code dec(Hex)	**Meaning**
0(0)	Successful. Either automatic line numbering or automatic character prompting has been restarted.
4(4)	A parameter was specified. No parameter should be specified.
8(8)	The request is not valid because one of the following has occurred: • Automatic line numbering or automatic character prompting was never started or never suspended. • An SPAUTOPT macro instruction has been issued to stop automatic line numbering or automatic character prompting.

SPAUTOPT - Stop Automatic Line Numbering or Character Prompting

Use the SPAUTOPT macro instruction to stop either the automatic line numbering feature or the automatic character prompting feature. Because only one of these features can be used at a time, the active feature is the feature that is stopped. See “STAUTOLN - Start Automatic Line Numbering” on page 153 for a description of the automatic line numbering feature, and “STAUTOCP - Start Automatic Character Prompting” on page 152 for a description of the automatic character prompting feature.

The system can suspend automatic prompting when the terminal user causes an attention interruption or enters a null line of input. Your application program should then issue this macro instruction in its attention exit, or when it receives a zero length input line from a TGET macro instruction. When the SPAUTOPT macro is used to stop prompting, you cannot use the RTAUTOPT macro to restart it. You must restart prompting by issuing either the STAUTOLN or STAUTOCP macro instruction.

The SPAUTOPT macro instruction can be used only in a time sharing environment. If you issue this macro when TSO/E is not active or when your program is running under Session Manager, it is ignored.

Figure 66 on page 152 shows the format of the SPAUTOPT macro instruction.

```
[symbol]      SPAUTOPT
```

Figure 66. The SPAUTOPT macro instruction

When control is returned to the user, register 15 contains one of the following return codes:

Table 39. Return codes from SPAUTOPT

Return code dec(Hex)	Meaning
0(0)	Successful. Either automatic line numbering or automatic character prompting has been stopped.
4(4)	A parameter was specified. No parameter should be specified.
8(8)	The request is not valid. Either automatic line numbering or automatic character prompting was never started.

STAUTOCP - Start Automatic Character Prompting

Use the STAUTOCP macro instruction to start automatic character prompting. Automatic character prompting signals the terminal user when the system is ready to accept input from the terminal. This signal consists of displaying at the terminal either an underscore and a backspace or a period and a carriage return, depending on the type of terminal being used. The STAUTOCP macro has no effect with a display station, because the terminal user is always prompted for input by the start-of-message symbol.

This macro instruction can be used to cause the system to automatically prompt the user for input.

Once started, automatic prompting is handled as follows: When the system has received a line of input, it immediately sends back to the terminal the next character prompt. If the program should send output while automatic prompting is in effect, the prompt will be repeated after all output has been sent to the terminal. For example:

```
line of input
OUTPUT MSG FROM PROGRAM
```

Automatic prompting is designed to be used by a program operating in input mode (that is, issuing successive TGET macros).

The system suspends automatic prompting when the terminal user causes an attention interruption or enters a null (nonprinting) line of input. The application program then takes appropriate action in an attention exit routine, or after receiving a zero length input from the TGET macro instruction. The application program can stop the prompting or line numbering function by using SPAUTOPT, or can restart the function via STAUTOCP.

The STAUTOCP macro instruction can be used only in a time sharing environment. It is ignored if issued by a batch task or if the program is running under Session Manager.

Figure 67 on page 153 shows the format of the STAUTOCP macro instruction.

```
[symbol]      STAUTOCP
```

Figure 67. The STAUTOCP macro instruction

When control is returned to the user, register 15 contains one of the following return codes:

Table 40. Return codes from STAUTOCP

Return code dec(Hex)	Meaning
0(0)	Successful.
4(4)	A parameter was specified. No parameter should be specified.

STAUTOLN - Start Automatic Line Numbering

Use the STAUTOLN macro instruction to start automatic line numbering. Automatic line numbering displays a line number at the beginning of each line.

This macro instruction can be used to cause the system to automatically prompt the user for input.

Once started, automatic line numbering is handled as follows: when the system has received a line of input, it immediately sends back to the terminal the next line number. If the program should send output while automatic line numbering is in effect, the line number will be repeated after all output has been sent to the terminal. For example:

```
00030 line of input
00040 OUTPUT MSG FROM PROGRAM
00040
```

Automatic line numbering is designed to be used by a program operating in input mode (that is, issuing successive TGET macros).

The system displays a new line number for each line of input received. The current line number maintained by the system is decreased appropriately whenever the input queue is cleared by a TCLEARQ macro or as the result of an attention interruption. Your application program is responsible for numbering the lines independently if it is creating a line numbered data set. The system line number is not available to the application program.

The system suspends automatic line numbering when the terminal user causes an attention interruption enters a null (nonprinting) line of input. The application program can then take appropriate action in an attention exit routine, or after receiving a zero length input from the TGET macro instruction. The application program can stop the line numbering function by using the SPAUTOPT macro instruction, or can restart the function by using either STAUTOLN or RTAUTOPT. You should use STAUTOLN rather than RTAUTOPT to restart automatic line numbering if the application program is numbering the input lines it receives. This choice will insure that the program's numbers are still in synchronization with the system's numbers.

The STAUTOLN macro instruction can be used only in a time sharing environment. It is ignored if issued by a batch task or if your program is running under Session Manager.

Figure 68 on page 154 shows the format of the STAUTOLN macro instruction. Each of the operands is explained following the figure.

```
[symbol]     STAUTOLN       S=address, I=address
```

Figure 68. The STAUTOLN macro instruction

S=*address*
indicates the address of a fullword that contains the number to be assigned to the first line of terminal input. This number can be any integer from 0 to 99,999,999.

I=*address*
indicates the address of a fullword that contains the increment value to be used when assigning line numbers to lines of terminal input. This number can be any integer from 0 to 99,999,999.

When control is returned to the user, register 15 contains one of the following return codes:

Table 41. Return codes from STAUTOLN

Return code dec(Hex)	Meaning
0(0)	Successful. A line number will be printed at the beginning of each line of input.
4(4)	A parameter is not valid because the specified value is out of range.

STFSMODE - Set Full-Screen Mode

Use the STFSMODE macro instruction under VTAM to specify whether an IBM 3270 display terminal is to operate in full-screen mode. Operating in full-screen mode provides screen protection by preventing the screen from being overlaid by non-full-screen messages, and allowing the terminal user to read non-full-screen messages before they are overlaid by full-screen messages. If full-screen mode is set off, full-screen TPUT requests (that is, TPUT requests that specify the FULLSCR operand) can result in certain problems at the terminal. A message not expected by the terminal user or the Command Processor, such as a broadcast message or password request, might not be noticed by the terminal user and might be quickly overlaid by a full-screen display. An unexpected message might overlay part of a full-screen display, which could result in incorrect input to the Command Processor.

See *z/OS TSO/E Programming Guide* for complete information about writing a full-screen Command Processor and examples of using the STFSMODE macro.

The STFSMODE macro instruction can be used only in a VTAM time-sharing environment and is ignored if issued when VTAM is not active.

Figure 69 on page 154 shows the format of the STFSMODE macro instruction.

```
 [symbol]    STFSMODE   [ ON ] +
 [,INITIAL=YES]  [,NOEDIT=YES] [,RSHWKEY=n] [,PARTION=YES]
                        [ OFF]  [,INITIAL=NO ]  +
[,NOEDIT=NO ]              [,PARTION=NO ]
```

Figure 69. The STFSMODE macro instruction

ON | OFF

ON
indicates that full-screen mode is in operation. If neither ON nor OFF is specified, ON is assumed. When a terminal operating in full-screen mode is to receive a non-full-screen message (TPUT without FULLSCR), the display screen is cleared, the alarm is sounded (if the Audible Alarm special feature is installed), and the message is displayed on the screen. If several such messages occur one after the other, the screen is cleared once, the alarm is sounded, and the messages are displayed in sequence. When the next full-screen TPUT message (TPUT with FULLSCR) is issued by the application, the terminal user will be required to acknowledge the messages on the screen before the TPUT FULLSCR can be displayed. Three asterisks (***) displayed at the current line indicate that acknowledgment is required. To continue, the user must press the Enter key.

OFF
indicates that full-screen mode is not in operation. When a terminal that is not operating in full-screen mode receives a message, the RSHWKEY is reset to the default, and the message is sent to the terminal according to the options specified in the TPUT macro, possibly overlaying the current screen contents.

INITIAL=YES | NO

YES
indicates that this is the first time during the execution of a Command Processor that the Command Processor has entered full-screen mode. This operand prevents the first TPUT FULLSCR issued by the Command Processor from forcing a paging condition when the last transaction at the terminal was input. For example, after a user logs on and the READY message is displayed and the user types in the name of a Command Processor, a paging condition is not forced if INITIAL=YES was specified. INITIAL=YES is ignored if OFF is specified.

Note that the first TPUT FULLSCR issued by the Command Processor forces a normal paging condition if INITIAL=YES is specified when the last transaction at the terminal was non-full-screen output.

NO
indicates that forced paging is to occur normally whenever a TPUT with FULLSCR follows a TPUT without FULLSCR. If neither INITIAL=YES nor INITIAL=NO is specified, INITIAL=NO is assumed.

NOEDIT=YES | NO

YES
indicates that input from the terminal will be added to the input queue without being modified, regardless of the options specified on the TGET macro instruction.

TSO/VTAM supports 3270 extended data stream functions via TGET in unedited input mode and TPUT NOEDIT. For more information about TPUT NOEDIT, refer to "Using the TPUT macro instruction to Write a Line to the Terminal" on page 257.

NO
indicates that input from the terminal will be handled according to the options specified on the TGET macro instruction before it is added to the input queue. If neither NOEDIT=NO nor NOEDIT=YES is specified, NOEDIT=NO is assumed.

RSHWKEY
specifies as a decimal digit the program function (PF) key to be used as the reshow key. If RSHWKEY is not specified, the default value for the PA2 key (X'6E') is used.

PARTION=YES | NO

YES
indicates to TSO/VTAM that partitions are being used and the buffer address of the terminal screen is either a 14 or a 16-bit address.

NO
indicates to TSO/VTAM that partitions are not being used and the buffer address of the terminal screen is a 12-bit address.

When control is returned to the user, register 15 contains one of the following return codes:

Table 42. Return codes from STFSMODE	
Return code dec(Hex)	**Meaning**
0(0)	Successful.
4(4)	An incorrect parameter was specified.
8(8)	The terminal type is not valid. This macro instruction is valid only for IBM 3270 display terminals that use VTAM.

STLINENO - Set Line Number

Use the STLINENO macro instruction under VTAM to specify the number of the screen line on an IBM 3270 display terminal on which the next non-full-screen message should appear. (A non-full-screen message results from issuing a TPUT macro instruction without the FULLSCR operand.) The STLINENO macro instruction can also be used to specify whether the 3270 terminal is to operate in full-screen mode.

See *z/OS TSO/E Programming Guide* for complete information about writing a full-screen Command Processor and examples of using the STLINENO macro.

The STLINENO macro instruction can be used only in a VTAM time-sharing environment and is ignored if issued when VTAM is not active.

Figure 70 on page 156 shows the format of the STLINENO macro instruction.

```
[symbol]     STLINENO      {LINE=number     }[,MODE=ON  ][,CLEAR=YES]
                           {LINELOC=address }[,MODE=OFF ][,CLEAR=NO ]
```

Figure 70. The STLINENO macro instruction

LINE=*number*
specifies in decimal the line number on which the next non-full-screen message is to appear. The line number must be a value from 1 to n where n is the maximum number of lines allowed for the terminal in use. Either the actual line number or a register (2-12, enclosed in parentheses) containing the line number in the low-order byte can be specified.

Note: LINE=1 clears the screen with the next output and sets full-screen mode to off.

LINELOC=*address*
specifies the address of a fullword whose low-order byte contains the number of the screen line on which the next non-full-screen message is to appear. Either an actual address (RX-type) or a register (2–12, enclosed in parentheses) containing the address may be specified.

MODE=ON | OFF
specifies whether full-screen mode is to be set ON or OFF. If MODE is not specified, MODE=OFF is assumed.

CLEAR=YES | NO
specifies to clear the screen when line number is one while turning Fullscreen mode off. If CLEAR=YES is specified, the screen will be cleared even though Fullscreen TPUT was written over with TPUT EDIT. If CLEAR is not specified, CLEAR=NO is assumed. The screen will not be cleared when the Fullscreen TPUT was written over with TPUT EDIT.

When control is returned to the user, register 15 contains one of the following return codes:

Table 43. Return codes from STLINENO	
Return code dec(Hex)	**Meaning**
0(0)	Successful.
4(4)	An incorrect parameter was specified.
8(8)	The terminal type is not valid. This macro instruction is valid only for IBM 3270 display terminals that use VTAM.
12(C)	The line number specified was 0 or it was greater than the maximum number of lines allowed for the terminal in use.

STSIZE - Set Terminal Line Size

Use the STSIZE macro instruction to set the logical line size of the time sharing terminal.

If the terminal is a display station, the STSIZE macro instruction is used to set the screen size. The STSIZE macro changes only the logical screen size of a terminal. In non-full-screen processing, the logical and physical screen sizes are the same. However, in full-screen processing they are not necessarily the same and when they are not the same, this macro does not change the physical screen size of the terminal. Full-screen applications can change the physical screen size using the appropriate WRITE command.

The STSIZE macro instruction can be used only in a time sharing environment. If you issue this macro when TSO/E is not active or when your program is running under Session Manager, it is ignored.

Figure 71 on page 157 shows the format of the STSIZE macro instruction. Each of the operands is explained following the figure.

```
[symbol]     STSIZE      {SIZE=number     }[,LINE=number     ]
                         {SIZELOC=address }[,LINELOC=address ]
```

Figure 71. The STSIZE macro instruction

SIZE=*number*
Specify the logical line size of the terminal in characters. If the logical line size requested is greater than the physical line size of the terminal, the last character in the line may be repeatedly typed over. Specifying a size greater than 255 gives unpredictable results.

SIZELOC=*address*
Specify the address of a word containing the logical line size of the terminal in characters.

LINE=*number*
Specify the number of lines that can appear on the screen of a display station terminal.

LINELOC=*address*
Specify the address of a word containing the number of lines that can appear on the screen of a display station terminal.

Note: If the terminal is a display station, either the LINE or LINELOC operand must be specified. If the terminal is not a display station, neither operand should be specified.

Defaults by terminal type are as follows:

Terminal type	Line size, number of lines, or screen size
2741	120
1050	120
33/35 Teletype	72

Terminal type	Line size, number of lines, or screen size
2260, 2265	12x80, 12x40, 6x40, 15x64
3270	12x40 or 24x80
3267	132
3770	132

When control is returned to the user, register 15 contains one of the following return codes:

Table 44. Return codes from STSIZE

Return code dec(Hex)	Meaning
0(0)	Successful.
4(4)	An incorrect parameter was specified.
8(8)	The LINE, LINELOC, SIZE, or SIZELOC operands are not valid for one of the following reasons: • The LINE or LINELOC operand was specified for a terminal that is not a display station. (An operand value of zero is not an error, and has the same effect as omitting the operand.) • The LINE or LINELOC operand was omitted, or specified as zero, for a display station. • The SIZE or SIZELOC operand was omitted, or specified as zero, for any terminal type.
12(C)	The dimensions specified for a display station do not correspond to a known, existing screen size. Incorrect screen management can result.

STTMPMD - Set Terminal Display Manager Options

Use the STTMPMD macro instruction to specify whether a Display Terminal Manager is active or whether the PA1 and CLEAR key indications are to be passed through to the application program.

The STTMPMD macro instruction can be issued only in a time-sharing environment. It is ignored if issued for a non-TSO/E task. The STTMPMD macro is valid for display terminals operating in the VTAM environments.

See *z/OS TSO/E Programming Guide* for complete information about using the STTMPMD macro in a full-screen Command Processor.

Figure 72 on page 158 shows the format of the STTMPMD instruction. Each of the operands is explained following the figure.

```
[symbol]     STTMPMD      [ ON]  [,KEYS{=NO}  ]
                          [OFF]  [     {ALL}  ]
```

Figure 72. The STTMPMD macro instruction

ON | OFF

ON
indicates that a Display Terminal Manager is in control. If neither ON nor OFF is specified, ON is the default.

OFF
indicates that a Display Terminal Manager is not in control.

KEYS=NO | ALL

NO
indicates that the PA1 and CLEAR key indications are not to be returned to the application program. This is the default if the KEYS operand is omitted.

ALL
indicates that the PA1 and CLEAR key indications are to be returned to the application program.

When control is returned to the user, register 15 contains one of the following return codes:

Table 45. Return codes from STTMPMD

Return code dec(Hex)	Meaning
0(0)	Successful.
4(4)	An incorrect parameter was specified.
8(8)	The terminal type is not valid because it is not a display terminal.

TCLEARQ - Clear Buffers

TCLEARQ enables your program to throw away "typed ahead" input or unsent output. This clearing of the buffers lets the Command Processor resynchronize with the terminal user.

For example, when a Command Processor analyzes the specified operands in a line of input and discovers missing or incorrect parameters, it issues a TCLEARQ INPUT before sending a prompting message to the user. This ensures that the Command Processor will receive a line of input entered after the terminal user has seen the prompting message.

When the TCLEARQ macro instruction is issued to clear the input buffers, all the input that has been entered at the terminal, but has not yet been processed by the program, is purged.

When the TCLEARQ macro instruction is issued to clear the output buffers, all the output that has been processed by the program but not yet displayed at the terminal is purged.

The TCLEARQ macro instruction can be used only in a time sharing environment. It is ignored if TSO/E is not active when the macro instruction is issued.

Figure 73 on page 159 shows the format of the TCLEARQ macro instruction; each of the operands is described following the figure.

```
[symbol]    TCLEARQ       [INPUT ]
                          [OUTPUT]
```

Figure 73. The TCLEARQ macro instruction

INPUT
indicates that all input currently in the terminal's input buffer queue will be lost, including the input line currently being entered, if any. If neither INPUT nor OUTPUT is specified, INPUT is assumed.

OUTPUT
indicates that all the output for this terminal that is currently in the terminal's output buffer queue will be purged, except for output messages that have begun to appear at the terminal, or messages from other terminals or the system operator.

When control is returned to the user, register 15 contains one of the following return codes:

Table 46. Return codes from TCLEARQ	
Return code dec(Hex)	**Meaning**
0(0)	Successful.
4(4)	An incorrect parameter was specified.

The following terminal control macro instructions are intended for system use, and are not suggested for use in user-written command processors. Inappropriate use of these macro instructions can cause terminal errors.

macro instruction	Function	Issue in 24-bit addressing mode
STATTN	Set attention simulation	x
STBREAK	Set break	x
STCC	Specify line-deletion and character-deletion characters	x
STCLEAR	Set display clear character string	x
STCOM	Set interterminal communication	x
STTIMEOU	Set timeout feature	x
STTRAN	Set character translation	x

STATTN - Set Attention Simulation

Use the STATTN macro instruction to specify how a terminal user can interrupt the execution of the program without using an attention key.

When the STATTN macro instruction assigns a value to an operand, that value remains in effect until another STATTN macro instruction assigns a new value to the operand, or until the terminal user logs off. Issuing the STATTN macro instruction without specifying any operands results in a NOP instruction.

The STATTN macro instruction can be used only in a time sharing environment with terminals that use TSO/E. It is ignored if TSO/E is not active when the macro instruction is issued.

Figure 74 on page 160 shows the format of the STATTN macro instruction. Each of the operands is explained following the figure. If an operand is not specified, its current status is not changed.

```
[symbol]      STATTN     [ LINES= {integer}]  [ ,TENS={integer} ]
                         [        {  0    }]  [       {  0    } ]
                         [ ,INPUT={address}]
                         [        {   0   }]
```

Figure 74. The STATTN macro instruction

LINES=*integer* | 0
indicates the output line count (if any) that determines when a terminal user can interrupt the execution of his program.

integer
specifies an integer from 1 to 255. This integer indicates the number of consecutive lines of output that can be directed to the terminal before the keyboard will unlock to let the terminal user interrupt the execution of his program.

0
indicates that output line count will not be used to determine when the terminal user can interrupt the execution of his program.

The LINES operand applies only to terminals that are not display stations. However, the display user can cause a simulated attention interruption at the bottom of the screen (that is, after every 6, 12, or 15 lines of consecutive output, depending on screen size).

TENS=*integer* | 0
indicates whether locked keyboard time will be used to determine when a terminal user can interrupt the execution of his program.

integer
specifies an integer from 1 to 255. This integer indicates the tens of seconds (that is, from 10 to 2550 seconds) of locked keyboard time that can elapse before the keyboard will unlock to let the terminal user interrupt the execution of his program.

0
indicates that locked keyboard time will not be used to determine when the terminal user can interrupt the execution of his program.

INPUT=*address* | 0
indicates whether a character string will be used to determine when a terminal user can interrupt the execution of his program.

address
specifies the address of a character string from one to four EBCDIC characters long, left-justified and padded to the right with blanks if less than four characters long. When this character string is encountered as the only data in a line, input processing is interrupted to let the program take an attention exit. For information on attention exits routines, see Chapter 12, “Using the STAX service routine to handle attention interrupts,” on page 285. This string will not be recognized if it is preceded by any other character, including line-delete or character-delete control characters.

0
indicates that no character string will be used to determine when the terminal user can interrupt the execution of his program.

When control is returned to the user, register 15 contains one of the following return code:

Table 47. Return codes from STATTN

Return code dec(Hex)	Meaning
0(0)	Successful
8(8)	The terminal type is not valid. This macro instruction should not be issued for terminals that use VTAM.

STBREAK - Set Break

Use the STBREAK macro instruction to indicate whether the transmit interrupt feature on an IBM 1050, 2741, 3270, 3767, or 3770 terminal will be used or suppressed. The transmit interrupt feature lets terminal output processing interrupt terminal input processing.

The transmit interrupt feature is a special feature on 1050 and 2741 terminals; it is a standard feature on the 3767, 3770, and 3270 display terminals. Specifying STBREAK YES for a 1050 without the transmit interrupt feature could result in loss of output or a permanent error at the terminal.

When the transmit interrupt feature is being used by the system, the terminal user can enter the next line while the previous one is being processed. All 33/35 Teletypes and IBM 3270, 3767, and 3770 terminals are handled this way. 1050s and 2741s that have been defined in the message control program as having the transmit interrupt feature will be handled this way unless STBREAK NO is specified.

Note: For 2741s, 3767s, 3770s, TWX, and WTTY devices supported by VTAM, the keyboard will remain unlocked when STBREAK NO is specified.

When the feature is in use, terminal handling of input and output is as follows: if no output is available for the terminal, and if there are sufficient TSO/E terminal buffers available, the keyboard will be unlocked to allow the user to enter input. If the user's program generates output (TPUT) before he has started to enter data, the read operation is halted and the break (transmit interrupt) feature can be used to lock the keyboard and condition the communications line to transmit output. If the user has already started to type when the TPUT is issued, the output will not be sent until he has finished that line of input. If, however, the TPUT had specified the BREAKIN option, the output message would interrupt any input in progress. If the application does not issue a TCLEARQ macro to erase the contents the input buffer queue then,

- The interrupted input from a 1050 or a 2741 terminal will be printed out again after the output is sent, to let the user continue to type from the point where he had been interrupted.
- The interrupted input from a 3767, 3270, or a 3770 terminal is received by the application program but is not printed at the terminal.

When the transmit interrupt feature is not being used by the system, a 1050 or 2741 terminal keyboard is unlocked only after the user's program has issued a TGET request for input. (A 3270, 3767, or 3770 terminal keyboard's normal state is unlocked.) In this mode of operation, the terminal user cannot type ahead of his program. A TPUT with the BREAKIN option cannot interrupt input. The output will not be sent until the terminal user has completed entering his current input line. All display stations are handled in this way. All 1050s and 2741s that have been defined in the message control program as not having the transmit interrupt feature are handled this way.

The STBREAK macro instruction can be used only in a time sharing environment. It is ignored if TSO/E is not active when the macro instruction is issued.

Figure 75 on page 162 shows the format of the STBREAK macro instruction.

```
[symbol]    STBREAK         [YES]
                            [NO ]
```

Figure 75. The STBREAK macro instruction

YES | NO

YES
indicates that the transmit interrupt feature will be used. YES is the default.

NO
indicates that the transmit interrupt feature not be used.

When control is returned to the user, register 15 will contain one of the following return codes:

Table 48. Return codes from STBREAK

Return code dec(Hex)	Meaning
0(0)	Successful.
4(4)	An incorrect parameter was specified.
8(8)	The terminal type is not valid. This macro instruction should be issued only for the IBM 1050, 2741, 3270, 3767, or 3770 terminal.

STCC - Specify Terminal Control Characters

Use the STCC macro instruction to specify what control characters will be used to delete a character or a line of terminal input.

When the line-delete control character specified in the STCC macro instruction is encountered within a line of terminal input, the line control character and all the preceding characters in that line are deleted. When the character-delete control character specified in the STCC macro instruction is encountered within a line of terminal input, the character-delete control character and the character immediately preceding it are deleted from the line.

When the user is logging on, he can delete a line or character by using the system-supplied defaults. The defaults, according to the type of terminal, are as follows:

Type of terminal	Desired action	Key(s) to be pressed
1050 and 2741	Line deletion or character deletion	Attention key and backspace
33/35 Teletype	Line deletion or character deletion	CTRL and X key (X'18.'), back arrow (<-), or underscore (_), depending on keyboard. (Either key results in X'6D'.)
3767/3770	Line deletion or character deletion	Attention key and backspace

No defaults are defined for the display stations, because the terminal user can use cursor control keys more effectively to delete characters or lines before the input is transmitted to the system.

The STCC macro instruction is valid in a time sharing environment with terminals (other than LU_T1 devices) that use TSO/E with the exception that ATTN/NATN is not supported in a VTAM environment. STCC is ignored if TSO/E is not active when the macro instruction is issued.

Figure 76 on page 163 shows the format of the STCC macro instruction; each of the operands is explained following the figure.

```
  [symbol]     STCC       [ATTN] [,LD={X'n'}+
]  [CD={X'n'}]
                          [NATN] [    {C'c'}+
]  [   {C'c }]
```

Figure 76. The STCC macro instruction

ATTN | NATN

ATTN
When this operand is in effect, pressing the ATTENTION key after having typed data will only delete the current line. System response is `!D`. Automatic prompting is not turned off. The ATTENTION key can then be pressed again, without typing any input, to interrupt the program and turn off prompting. When this operand is *not* in effect, the attention key will both delete a line of terminal input and interrupt the execution of the user's program. System response is `!` or `!I`.

NATN
indicates that the attention key will not be used to delete a line of terminal input.

LD=
indicates what character will be used for the line delete control character:

X'n'
where X'n' is the hexadecimal representation of any EBCDIC character on the terminal keyboard, except the new line (NL) and carrier return (CR) control characters. If X'00' is specified, the previously used line-delete control character is retained. If X'FF' is specified, no character will be used for the line-delete control character. If an incorrect character is specified, that character is rejected and no character is used to delete a line of terminal input.

C‘c’
where *c* is the character representation of any EBCDIC character on the terminal keyboard.

CD=
indicates what character will be used for the character-delete control character:

X'n'
where X'n' is the hexadecimal representation of any EBCDIC character on the terminal keyboard except the new line (NL) and carrier return (CR) control characters. If X'00' is specified, the previously used character-delete control character is retained. If X'FF' is specified, no character will be used for the character- delete control character. If an incorrect character is specified, that character is rejected and no character is used to delete a character from a line of terminal input.

C‘c’
where *c* is the character representation of any EBCDIC character on the terminal keyboard.

When control is returned to the user, the low-order byte of register 0 contains the former line-delete control character. If X'FF' appears in the low-order byte of register 0, there is no former line-delete control character. If X'80' appears in the high-order byte of register 0, ATTN was in effect for line deletion prior to the issuance of the STCC macro.

The low-order byte of register 1 contains the former character-delete control character. If X'FF' appears in the low-order byte of register 1, there is no former character-delete control character.

Register 15 contains one of the following return codes:

Table 49. Return codes from STCC

Return code dec(Hex)	Meaning
0(0)	Successful.
4(4)	Incorrect parameters were specified to the SVC.
8(8)	The request is not valid because either the specified character does not appear on the terminal keyboard, or ATTN was specified for a terminal that does not have an attention key.
12(C)	The terminal type is not valid.

STCLEAR - Set Display Clear Character String

Use the STCLEAR macro instruction to specify the character string that will be used to request that a 2260 or 2265 display station screen be erased.

The STCLEAR macro instruction can be used only in a time sharing environment. It is ignored if TSO/E is not active when the macro instruction is issued.

Figure 77 on page 164 shows the format of the STCLEAR macro instruction. Each of the operands is explained following the figure.

```
[symbol]     STCLEAR       STRING={address}
                                  {   0   }
```

Figure 77. The STCLEAR macro instruction

STRING=*address* | 0
indicates the address of a one- to four-character string that will be used to request that the display station screen be erased. This character string must be left-justified and padded on the right with blanks, if necessary. If 0 is specified, no character string will be used to erase the screen.

When control is returned to the user, register 15 contains one of the following return codes:

Table 50. Return codes from STCLEAR	
Return code dec(Hex)	**Meaning**
0(0)	Successful.
4(4)	An incorrect parameter was specified.
8(8)	The terminal type is not valid because it is not a display station.

STCOM - Set Inter-Terminal Communication

Use the STCOM macro instruction to specify whether a terminal will accept messages from other terminals or low priority messages from the system operator. High priority operator messages are always sent to the terminal.

The STCOM macro instruction can be used only in a time sharing environment. It is ignored if TSO/E is not active when the macro instruction is issued.

Figure 78 on page 165 shows the format of the STCOM macro instruction.

```
[symbol]     STCOM       [YES]
                         [NO ]
```

Figure 78. The STCOM macro instruction

YES | NO

YES
indicates that the terminal will accept messages from other terminals. If neither YES nor NO is specified, YES is assumed.

NO
indicates that the terminal will not accept messages from other terminals.

When control is returned to the user, register 15 contains one of the following return codes:

Table 51. Return codes from STCOM	
Return code dec(Hex)	**Meaning**
0(0)	Successful.
4(4)	An incorrect parameter was specified.

STTIMEOU - Set Time Out Feature

Use the STTIMEOU macro instruction to specify whether the 1050 terminal has the optional text time out suppression feature. The macro instruction allows a 1050 terminal, with or without the feature, to call in using the same switched line, and to be handled initially as if it did not have the feature.

A 1050 without the text time out suppression feature operates as follows: When the PROCEED light is on and the keyboard is unlocked, the terminal will time out; that is, the keyboard will lock if the user does not type input for approximately 20 seconds. The system subsequently responds to the time out by restoring the keyboard so that the user may continue. The user can prevent the time out by periodically pressing the SHIFT key.

A 1050 with the text time out suppression feature operates as follows: The keyboard does not lock if the user does not type input within 20 seconds. The system can therefore use the read inhibit channel command, which does not time out within 28 seconds, in contrast to the read channel command that

does time out. (Note: If the system is directed to use the read inhibit channel command for a 1050 that does time out, the terminal may be locked out of the system.)

Until the STTIMEOU macro instruction is issued, 1050 terminals are handled according to the definition provided in the message control program. If the currently connected terminal has the text time out suppression feature, STTIMEOU NO can be issued to direct the system to use read inhibit rather than read channel commands. (STTIMEOU NO should not be issued for a 1050 that does not have the text time out suppression feature. This specification could cause the terminal to be locked out of the system.)

The STTIMEOU macro instruction should be issued only when an IBM 1050 terminal is being used. Terminals which are equivalent to the one explicitly supported may also function satisfactorily. The customer is responsible for establishing equivalency. IBM assumes no responsibility for the impact that any changes to the IBM-supplied products or programs may have on such terminals.

The STTIMEOU macro instruction can be used only in a time sharing environment. It is ignored if TSO/E is not active when the macro instruction is issued.

Figure 79 on page 166 shows the format of the STTIMEOU macro instruction.

```
[symbol]     STTIMEOU        [YES]
                             [NO ]
```

Figure 79. The STTIMEOU macro instruction

YES | NO

YES
indicates that IBM 1050 terminal does time out. It does not have the text time out suppression feature. If the operand is omitted, the default is YES.

NO
indicates that the IBM 1050 terminal does not time out. The 1050 does have the text time out suppression feature.

When control is returned to the user, register 15 contains one of the following return codes:

Table 52. Return codes from STTIMEOU

Return code dec(Hex)	Meaning
0(0)	Successful.
4(4)	An incorrect parameter was specified.
8(8)	The terminal type is not valid. This macro instruction applies to the IBM 1050 terminal only.

STTRAN - Set Character Translation

Use the STTRAN macro instruction to initiate the use of user-specified translation tables, to modify specific character translations in active translation tables, to remove character modifications made to user-specified translation tables, and to terminate the use of user-specified translation tables. Translation tables allow characters entered at the terminal to be interpreted as other characters when they are received by TSO/E, and characters sent by TSO/E to be interpreted as other characters when they are received at the terminal.

Translation tables are built and used in pairs: one for input and one for output. Each pair is a control section consisting of a fullword containing the address of the output table, followed by a 256-byte EBCDIC table for translating the inbound characters, followed by a 256-byte EBCDIC table for translating the outbound characters. Each character in an input table must have a counterpart in its companion

output table, and the characters must have the same relative position in both tables. See *z/OS TSO/E Customization* for instructions on building translation tables.

A translation table translates inbound data after the system translates the line code to EBCDIC characters. A translation table translates outbound data before the system translates EBCDIC characters to line code.

The STTRAN macro instruction can be used only in a VTAM time-sharing environment. It is ignored if VTAM is not active when the macro instruction is issued.

Figure 80 on page 167 shows the format of the STTRAN macro instruction. Each of the operands is explained following the figure.

```
[symbol]     STTRAN     [ { {TABLE=address,NAME=address }} ]
                        [ { {NOTRAN                     }} ]
                        [ {                              } ]
                        [ { {TCHAR=address,SCHAR=address}} ]
                        [ { {NOCHAR,NAME=address        }} ]

                            [MF={L             } ]
                            [   {(E,ctrl addr)} ]
```

Figure 80. The STTRAN macro instruction

TABLE=*address*
specifies the address of a pair of user-written translation tables.

NAME=*address*
specifies the address of an 8-byte area containing an EBCDIC character string. (The string is left-justified and padded to the right with blanks if it is less than eight characters long.) The character string consists of the name of a member in a load module that contains user-written translation tables.

When NAME is used with NOCHAR, the STTRAN macro instruction causes the Command Processor to store the member name in the 8-byte area.

NOTRAN
specifies that the use of user-written translation tables be discontinued.

TCHAR=*address*
specifies the address of a 1-byte area containing the EBCDIC representation of a character as it appears at the terminal.

SCHAR=*address*
specifies the address of a 1-byte area containing the EBCDIC representation of a character as it appears to the system.

NOCHAR
specifies that current TCHAR and SCHAR values are no longer in effect.

MF=L | (E,*ctrl addr*)
indicates the form of the STTRAN macro instruction.

L
specifies the list form.

(E,*ctrl addr*)
specifies the execute form and the address of the list form.

When control is returned to the user, register 15 contains one of the following return codes:

Table 53. Return codes from STTRAN

Return code dec(Hex)	Meaning
0(0)	Successful.
4(4)	NOTRAN or NOCHAR was specified but translation was not in effect.
8(8)	TABLE or NOCHAR was specified but the NAME operand did not specify an address.
12(C)	An internal error occurred - an unidentifiable flag was set in input register 0.

Chapter 8. Using BSAM or QSAM for terminal I/O

This chapter describes how to use the basic sequential access method (BSAM) and the queued sequential access method (QSAM) to provide terminal I/O support for programs that run under TSO/E. For a complete discussion of the use of BSAM and QSAM, see *z/OS DFSMS Using Data Sets*.

The major benefit of using BSAM or QSAM to process terminal I/O under TSO/E is that programs using these access methods do not become TSO/E dependent or device dependent and might execute either under TSO/E or in the batch environment. Therefore, your existing programs that use BSAM or QSAM for I/O might be used under TSO/E without modification or recompilation. However, there are some restrictions as noted in this chapter.

Overview of the BSAM and QSAM macro instructions

Some of the BSAM and QSAM access method routines have been modified to provide special services under TSO/E; others provide the same function that is provided in a batch environment. Those BSAM and QSAM macro instructions that are not relevant to terminal I/O act as no-ops. All of the BSAM and QSAM macro instructions, when executed in the batch environment, provide the non-terminal functions as explained in *z/OS DFSMS Macro Instructions for Data Sets*.

You can issue the BSAM and QSAM macro instructions in 24-bit or 31-bit addressing mode but with these restrictions that do not apply to other environments:

- The EODAD specification on the DCBE macro has no effect, so the end-of-data routine must be specified in the DCB macro and reside below the 16 MB line.
- Certain ABEND issuances do not result in an IEC message or in calling the DCB ABEND exit routine.
- The RMODE31=BUF operand in the DCBE macro has no effect, so the QSAM buffer pool, if any, is below the 16 MB line. Therefore, OPEN does not turn on the DCBEMD31 bit.
- The large block interface, LBI, is not available.

Table 54 on page 169 shows the functions performed by the BSAM and QSAM macro instructions when used for terminal I/O. Following the table are more detailed explanations of the GET, PUT, PUTX, READ, WRITE, and CHECK macro instructions.

Table 54. BSAM and QSAM macro functions under TSO/E

SAM macro instruction	BSAM	QSAM	Terminal interpretation
BSP	X	X	NOP
BUILD	X	X	As in batch processing, the BUILD macro instruction causes a buffer pool to be constructed in a user-provided storage area.
CHECK	X		Takes an EODAD exit after a READ EOF. NOP after a WRITE.
CLOSE	X	X	The CLOSE macro instruction frees the control blocks built to handle I/O and deletes the loaded SAM terminal routines.
CNTRL	X	X	NOP
DCBE	X	X	Currently has no effect.
FEOV	X	X	NOP
FREEBUF	X		As in batch processing, the FREEBUF macro instruction causes the control program to return a buffer to the buffer pool assigned to the specified data control block.

Table 54. BSAM and QSAM macro functions under TSO/E (continued)

SAM macro instruction	BSAM	QSAM	Terminal interpretation
FREEPOOL	X	X	As in batch processing, the FREEPOOL macro instruction causes an area of virtual storage, previously assigned as a buffer pool for a specified data control block, to be released.
GET		X	The GET macro instruction obtains data from the terminal.
GETBUF	X		As in batch processing, the GETBUF macro instruction causes the control program to obtain a buffer from the buffer pool assigned to the specified data control block, and to return the address of the buffer in a designated register.
GETPOOL	X	X	As in batch processing, the GETPOOL macro instruction causes a buffer pool to be constructed in a storage area provided by the control program.
NOTE	X		NOP
OPEN	X	X	The OPEN macro instruction loads the proper SAM terminal I/O routines and constructs the necessary control blocks.
POINT	X		NOP, but TYPE=RELNEXT is not supported.
PRTOV	X	X	NOP
PUT		X	The PUT macro instruction routes data to the terminal.
PUTX		X	The PUTX macro instruction routes data to the terminal.
READ	X		The READ macro instruction obtains data from the terminal.
RELSE		X	NOP
SETPRT	X	X	NOP
TRUNC		X	NOP, but BSAM is not supported.
WRITE	X		The WRITE macro instruction routes data to the terminal.

Note: If QSAM is used for terminal I/O and the data set is defined with BLKSIZE=80 and RECFM=U, which is not recommended, each line will be truncated by 1 character. This byte (the last byte) is reserved for an attribute character.

The SAM Terminal Routines

The GET, PUT, PUTX, READ, WRITE, and CHECK macro instructions perform differently in terminal I/O than they do in the batch environment. Descriptions of these differences are presented here, but for a detailed explanation of how to use the macro instructions, see *z/OS DFSMS Macro Instructions for Data Sets*.

GET

The GET macro instruction causes a record to be retrieved from the terminal and placed in either the first buffer of the buffer pool control block (locate mode) or in a user specified area (substitute or move mode). In either case, the address of the record is returned in register 1.

The input to the GET macro instruction consists of the DCB address and the user's area address, which is omitted for locate mode. The output is edited, which means that specially-indicated characters are deleted from the message. Also, lowercase characters are folded to uppercase characters.

When the terminal user types /*, end-of-file is indicated and control is passed to the problem program's EODAD routine. If no EODAD routine is specified, the job will ABEND with a system code of 337.

PUT and PUTX

Both the PUT and the PUTX macro instructions cause a record to be written to a terminal.

In locate mode, the first use of PUT or PUTX causes an address pointing to a buffer to be returned in register 1. The first record is placed in this buffer by the problem program and is written out when the next PUT or PUTX for the same data control block (DCB) is issued. Succeeding records are written in the same manner. The last record is written at CLOSE time.

In move or substitute mode, the PUT or PUTX macro instruction moves a record from the user-specified work area to the terminal. You must supply the work area address to the PUT macro instruction.

The input to the PUT and PUTX macro instruction consists of the DCB address and the user's area address, which is omitted for locate mode.

READ

The READ macro instruction causes a block of data to be retrieved from the terminal and placed in a user-designated area in storage. The data is folded to uppercase.

For a description of the input parameters for the READ macro instruction, see the discussion in *z/OS DFSMS Macro Instructions for Data Sets*.

WRITE

The WRITE macro instruction causes a block of data to be written from the user-specified area to the terminal.

For a description of the input parameters for the WRITE macro instruction, see the discussion in *z/OS DFSMS Macro Instructions for Data Sets*.

CHECK

The CHECK macro instruction, when used after a WRITE macro instruction, results in a NOP. When it is used after a READ macro instruction, it performs as a NOP unless an end of file (EOF) condition is encountered. The end of file signal from the terminal is /*. When end of file is encountered, CHECK takes the EODAD exit specified in the data control block. If no EODAD exit is specified, CHECK will cause the job to abend with a system code of 337.

The input to the CHECK macro instruction is the address of the problem program's data event control block (DECB).

Record Formats, Buffering Techniques, and Processing Modes

All record formats, fixed (F), variable (V), and undefined (U), are supported under TSO/E. Before passing the data to the problem program, TSO/E automatically generates the first four bytes of control information for V format records coming in from the terminal. When you send V format records to the terminal, TSO/E automatically removes the control information before writing the line.

Control characters (ANSI or machine) are not supported under TSO/E. On output, they are removed before the data is sent to the terminal. On input, they are ignored.

Table reference characters with OPTCD=J are treated as part of the data, and the access method does not remove them.

Both simple and exchange buffering techniques are supported, as are all four processing modes for the queued access method.

Specifying Terminal Line Size

If the LRECL and BLKSIZE fields are not specified in the DCB, the terminal line size default, or the line size the terminal user has specified using the TERMINAL command, is merged into the data control block fields as if it came from the label of the data set.

For BSAM, BLKSIZE is used by TSO/E to determine the length of the text line it is to process. For both BSAM and QSAM, if the text entered from the terminal is shorter than the value specified for LRECL, and if

F format is used, blanks are supplied on the right. For either access technique, if the text entered is longer than BLKSIZE or LRECL, the next GET or READ retrieves the remainder of the message. If the record generated by the problem program is longer than the specified line size, multiple lines are displayed at the terminal.

End-of-File (EOF) for Input Processing

The sequential access method GET and CHECK terminal routines recognize /* from the terminal as an end-of-file (EOF). The EODAD exit in the data control block is taken for the EOF condition. If no EODAD exit has been specified, and an EOF has been signaled from the terminal, the job abends with a system code of 337.

Modifying DD Statements for Batch or TSO/E Processing

TERM=TS, when added to a DD statement defining an input or an output data set, is ignored in the batch processing environment, but under TSO/E indicates to the system that the unit to which I/O is being addressed is a time sharing terminal. Therefore, if you want a job to run in either the foreground or the background, provide a DD statement as follows:

```
//DD1     DD     TERM=TS,SYSOUT=A
```

In this example the output device is defined as a terminal under TSO/E processing, and as the SYSOUT device during batch processing. For a complete description of the TERM=TS parameter, see *z/OS MVS JCL Reference*.

Chapter 9. Using the TSO/E I/O service routines for terminal I/O

This chapter describes how to use the TSO/E I/O service routines, STACK, GETLINE, PUTLINE, and PUTGET, to process terminal I/O.

Functions of the I/O Service Routines

If you write your own command processors, use the I/O service routines to process terminal I/O. Table 55 on page 173 describes the function of each of the I/O service routines.

Table 55. The TSO/E I/O service routines

Service routine	Function
STACK	Establishes and changes the source of input.
GETLINE	Obtains a line of input, other than commands, subcommands, and prompt message responses.
PUTLINE	Writes a line to the terminal.
PUTGET	Writes a message to the terminal and obtains a line of input in response.

The I/O service routines, STACK, GETLINE, PUTLINE, and PUTGET, offer the following features:

- They write to or obtain input from a terminal.
- They provide a method of selecting sources of input other than the terminal. Your Command Processor can direct requests for input to an in-storage list or data set as well as to the terminal.
- They provide a message formatting facility that allows you to insert text segments into a basic message format, and display or inhibit the displaying of message identifiers.
- They process requests for more information (question-mark processing), and they analyze processing conditions to determine if I/O requests should be disregarded or honored.

Passing Control to the I/O Service Routines

Your Command Processor can pass control to the I/O service routines in the following ways:

- By using the CALLTSSR macro instruction and specifying the entry point name of the I/O service routine. See Chapter 4, “Invoking TSO/E service routines with CALLTSSR,” on page 33. Use the following entry point names to invoke the I/O service routines:

  Service Routine
  Entry Point Name

  STACK
  IKJSTCK

  GETLINE
  IKJGETL

  PUTLINE
  IKJPUTL

  PUTGET
  IKJPTGT

  If you use the CALLTSSR macro instruction to invoke the I/O service routines, you must first create an input/output parameter list (IOPL) and place its address in general register 1. See “The Input/Output Parameter List” on page 174.

- By using the list and execute forms to the I/O service routine macro instructions. These macro instructions allow you to pass control to the I/O service routines and indicate the functions you want performed by coding the operands you require.

  Each of the I/O service routine macro instructions, STACK, GETLINE, PUTLINE, and PUTGET, has a list and an execute form. The list form of each service routine macro instruction initializes the parameter blocks according to the operands you code on the macro. The execute form is used to modify the parameter blocks and to provide linkage to the service routines, and can be used to set up the input/output parameter list. The input/output parameter list contains addresses required by the I/O service routines.

Addressing Mode Considerations

Your Command Processor can invoke the I/O service routines or issue the I/O service routine macro instructions in either 24-bit or 31-bit addressing mode. These routines return control to their caller in the same addressing mode with which they were invoked. The caller's parameters must be in the primary address space. The TSO/E I/O service routines must be invoked under a program status word (PSW) that is running key 8, problem program state.

Input can reside above or below 16 MB in virtual storage, except for the list storage descriptor (LSD), which is used by the STACK service routine. The LSD must reside below 16 MB in virtual storage.

The input/output parameter list (IOPL), which is needed by the I/O service routines, can reside above or below 16 MB in virtual storage. However, if the IOPL resides above 16 MB, then your Command Processor must execute in 31-bit addressing mode.

Service routines treat input addresses according to the addressing mode in which they are invoked. However, if you use the GETLINE macro, the addressing mode of the STACK macro is used rather than your program's addressing mode. Address values are treated as 24-bit or 31-bit addressing mode, depending on the addressing mode of the original issuer of the STACK macro for that element.

Considerations for Using I/O Service Routines by a Multitasking Application

An MVS application executing in a multitasking environment must be aware of the control blocks that TSO/E might be updating on the application's behalf. It is the application's responsibility to ensure that only one I/O service routine operates on a given I/O environment, specifically an ECT, at a single time. To support concurrent use of the I/O service routines, an MVS task can pass the address of a new ECT to the I/O service routines (STACK, GETLINE, PUTLINE, and PUTGET). To create a new ECT, the MVS task can use the ENVIRON=CREATE operand of the STACK service routine. Similarly, the MVS task can destroy an ECT using the ENVIRON=DESTROY operand of the STACK service routine. For more information about the STACK ENVIRON operand, see “Using STACK to Change the Source of Input” on page 176.

If an application task attempts to run an I/O service routine on a given ECT while another task is running an I/O service routine on the same ECT, the STACK service routine issues abend code X'66D'. Similarly, an application task should not attempt to destroy an I/O environment while another task is currently using the environment. If the application attempts to destroy the I/O environment while another task is using the I/O environment, the STACK service routine issues abend code X'66D'.

The Input/Output Parameter List

The I/O service routines use two of the pointers contained in the Command Processor parameter list (CPPL), which is described in “Interfacing with the TSO/E service routines” on page 13. These pointers are the pointer to the user profile table and the pointer to the environment control table. Your Command Processor must pass these addresses to the service routines in another parameter list, the input/output parameter list (IOPL).

Before executing any of the TSO/E I/O macro instructions, GETLINE, PUTLINE, PUTGET, or STACK, you must provide an IOPL and pass its address to the I/O service routine. There are two ways you can construct an IOPL:

- You can build and initialize the IOPL within your code and place a pointer to it in the execute form of the I/O macro instruction.
- You can provide space for an IOPL (4 fullwords), pass a pointer to it, together with the addresses required to fill it, to the execute form of the I/O macro instruction, and let the I/O macro instruction build the IOPL for you.

You can use the IKJIOPL DSECT, which is provided in SYS1.MACLIB to map the fields in the IOPL. Table 56 on page 175 describes the format of the IOPL.

Table 56. The Input/Output parameter list

Number of bytes	Field name	Contents or meaning
4	IOPLUPT	The address of the user profile table from the CPPLUPT field of the Command Processor parameter list.
4	IOPLECT	The address of the environment control table from the CPPLECT field of the CPPL.
4	IOPLECB	The address of the command processor's event control block (ECB). The ECB is one word of storage, declared and initialized to zero by the Command Processor. Command processors with attention exits can post this ECB after an attention interruption to cause active service routines to exit.
4	IOPLIOPB	The address of the parameter block created by the list form of the I/O macro instruction. There are four types of parameter blocks, one for each of the I/O service routines: • STACK parameter block (STPB) • GETLINE parameter block (GTPB) • PUTLINE parameter block (PTPB) • PUTGET parameter block (PGPB).

The parameter block pointed to by the fourth word of the I/O parameter list (IOPLIOPB) is created and initialized by the list form of the I/O macro instruction, and is modified by the execute form. Therefore, you can use the same parameter block to perform different functions. All you need to do is code different parameters in the execute forms of the macro instructions; these parameters provide those options not specified in the list form, and override those which were specified.

The STACK, GETLINE, PUTLINE, and PUTGET parameter blocks are described in the separate sections on each of the I/O macro instructions.

Using the I/O Service Routine macro instructions

You can use the I/O service routine macro instructions to pass control to the STACK, GETLINE, PUTLINE, and PUTGET service routines.

Each of the I/O macro instructions has a list and an execute form. The list form sets up the parameter block required by that I/O service routine; the execute form can be used to set up the input/output parameter list, and to modify the parameter block created by the list form of the macro instruction.

The parameter block required by each of the I/O service routines is different, and each one can be referenced through a DSECT which is provided in SYS1.MACLIB. The parameter blocks and the DSECTs used to reference them are:

Service routine	page #	DSECT name	Parameter block
STACK	"Using STACK to Change the Source of Input" on page 176	IKJSTPB	The STACK parameter block
GETLINE	"Using GETLINE to Get a Line of Input" on page 197	IKJGTPB	The GETLINE parameter block
PUTLINE	"Using PUTLINE to Put a Line Out to the Terminal" on page 209	IKJPTPB	The PUTLINE parameter block
PUTGET	"Using PUTGET to Put a Message Out to the Terminal and Obtain a Line of Input in Response" on page 232	IKJPGPB	The PUTGET parameter block

Each of these blocks is explained in the section describing the I/O macro instruction that builds it.

Using STACK to Change the Source of Input

Use the STACK macro instruction to establish and to change the source of input. The currently active input source is described by the top element of the input stack, an internal pushdown list, which is anchored in the environment control table (ECT) and maintained by the I/O service routines. The first element of the input stack is initialized to indicate that the terminal is the current input source, and cannot be changed or deleted afterward. The STACK service routine adds an element to the input stack or deletes one or more elements from it, and therefore changes the source of input for the other I/O service routines.

Your Command Processor can divide the input stack into substacks by creating barrier elements with the STACK macro instruction. A barrier element separates one group of stack elements, or substack, from another group of stack elements. Each substack can then be treated as a separate input stack. Use the barrier function of the STACK macro with the PUTGET or GETLINE SUBSTACK=YES services to determine when a barrier element is reached on the input stack.

Your program cannot build an input stack directly; it must invoke the STACK service routine to create a valid input stack. If your program builds an input stack without invoking the STACK service routine to do so, the I/O service routines issue an ABEND code of X'66D'.

To create an alternate input stack:

- Use the ENVIRON=CREATE operand of the STACK macro to create a new I/O environment consisting of a new ECT, input stack, and related data areas.

or

- Perform the following processing:

  1. Preserve the current input stack by saving the original value of the ECTIOWA field. The ECTIOWA field is contained in the ECT.

2. To build an alternate input stack and add an element, set the ECTIOWA pointer to zero and invoke the STACK service routine. The STACK service routine sets the ECTIOWA field to indicate that it has created an alternate input stack and added the element to the stack.
3. When processing using the alternate input stack is complete, restore the original value of the ECTIOWA field.

Use the ENVIRON=DESTROY operand of the STACK macro to destroy the I/O environment created with the ENVIRON=CREATE operand. The system will free the storage associated with the input stack as well as any other related data areas.

The STACK service routine saves the addressing mode of the program that invoked it. Address values are treated as 24-bit or 31-bit addressing mode, depending on the addressing mode of the original issuer of STACK for that element.

In the sections that follow, the following topics are discussed:

STACK Macro Effects on the REXX Data Stack

Whenever an application issues the STACK macro to add either a terminal element or an input file name to the input stack, the STACK service routine protects the previous contents of the REXX data stack by placing a terminal element, MARKTERM, on the REXX data stack. Similarly, when the terminal element or input file name is being removed from the input stack, the STACK service routine removes the MARKTERM terminal element from the REXX data stack.

However, when you create an alternate input stack, the STACK service routine will protect the REXX data stack through MARKTERM, but you must remove the MARKTERM element from the REXX data stack when you have completed using the alternate input stack. You can create an alternate input stack by clearing the ECTIOWA field in the ECT and then invoking the STACK macro to add a terminal element or an input file name. When you have completed using this alternate input stack, invoke the REXX stack routine, IRXSTK, with a function call of DROPTERM to remove the MARKTERM terminal element from the REXX data stack. By issuing DROPTERM when the input stack is no longer in use, you will keep the REXX data stack in synchronization with the input stack.

The List Form of the STACK macro instruction

The list form of the STACK macro instruction builds and initializes a STACK parameter block (STPB), according to the operands you specify in the macro. The STACK parameter block indicates to the STACK service routine which functions you want performed.

In the list form of the macro instruction, only

```
STACK      MF=L
```

Is required. When only STACK MF=L is specified, the STPB is zeroed. The other operands and their sublists are optional because they can be supplied by the execute form of the macro instruction.

Figure 81 on page 178 shows the list form of the STACK macro instruction; each of the operands is explained following the figure.

```
                        [ TERM=*                                      ]
                        [                                             ]
[symbol]     STACK      [ BARRIER= {*      }                          ]
                        [          {NONEST }                          ]
                        [                                             ]
                        [         {TOP    }                           ]
                        [ DELETE= {PROC   }                           ]
                        [         {ALL    }                           ]
                        [         {BARRIER}                           ]
                        [                                             ]
                        [ ENVIRON= {CREATE  }                         ]
                        [          {DESTROY }                         ]
                        [          {RESET   }                         ]
                        [                                             ]
                        [ INQUIRE= {ATTN }                            ]
                        [          {ERROR}                            ]
                        [          {TYPE }                            ]
                        [                                             ]
                        [                                {PROCN,PROMPT} ]
                        [ STORAGE=(element address,      {PROCL,PROMPT} ]
                        [                                {SOURCE      })],MF=L
                        [                                             ]
                        [         { *                        }        ]
                        [ DATASET=({ INDD=addr1,PROMPT,LIST})         ]
                        [         { MEMBER=addr3             }        ]
                        [         { OUTDD=addr2,CNTL,SEQ     }        ]
                        [         { CLOSE                    }        ]
```

Figure 81. The List Form of the STACK macro instruction

TERM=*

Adds a terminal element to the input stack.

Note: TERM=* is allowed by STACK to provide compatibility with existing modules when they are recompiled.

BARRIER=

Creates a barrier element, which divides the input stack into substacks, on top of the input stack.

***

CLISTs and REXX execs on opposing sides of this barrier are nested. They are able to use command output trapping and can communicate through global variables. Command processors can use routines IKJCT441 and IRXEXCOM to access variables on the opposing side of the barrier.

NONEST

CLISTs and REXX execs on opposing sides of the barrier are not nested. This type of barrier halts the effect of command output trapping and halts the use of the routines IKJCT441 and IRXEXCOM to access variables on the opposing side of the barrier. While CLIST global variables are not communicated across this barrier, CLISTs on top of this barrier can begin using global variables and communicate with further nested CLISTs through global variables.

Note: When stacking and removing barrier elements:

- Only STACK DELETE=BARRIER or STACK ENVIRON=RESET can remove a barrier element.
- If the application or Command Processor stacks a barrier element, the application or Command Processor must remove the barrier element when it is done using the task. Failure to remove the barrier element can result in miscommunication between the application that invoked your Command Processor and other command processors and applications that use barriers on the current input stack.

DELETE=

Deletes an element or elements from the input stack. TOP, PROC, ALL, or BARRIER further defines the element to be deleted.

TOP
Deletes the topmost element (the element that is most recently added to the input stack). If the top element is a barrier element, STACK DELETE=TOP is a no-operation instruction.

PROC
Deletes the current procedure element from the input stack. If the top element is not a PROC element, delete all elements down to, and including, the first PROC element.

ALL
Deletes all elements, except the bottom or first element, from the input stack. If one or more barrier elements exist on the input stack, delete all elements down to, but not including, the first barrier element.

BARRIER
Deletes all elements down to, and including, the first barrier element.

ENVIRON=
Specifies one of the following operations:

- A new TSO/E I/O environment is to be created.
- An existing TSO/E I/O environment is to be destroyed.
- The current TSO/E I/O environment is to be reset.

A TSO/E I/O environment consists of the control blocks that describe the input and output sources that are used by the I/O service routines. These control blocks include the environment control table (ECT) and the input stack.

CREATE
Specifies that a new TSO/E I/O environment is to be created. The STACK service routine creates a new environment using a model environment that is provided by your Command Processor. To create a new I/O environment, follow these steps:

1. Set the IOPLECT field in the input/output parameter list (IOPL) to the address of the ECT to be used as a model to create a new ECT. The IOPL is described in "The Input/Output Parameter List" on page 174. The ECT that you provide as the model is passed to your Command Processor in the Command Processor parameter list (CPPL). For more information about the CPPL, see "Interfacing with the TSO/E service routines" on page 13.
2. Invoke the STACK service routine, specifying the ENVIRON=CREATE operand.

When the STACK service routine returns control to your Command Processor, the STPBECTA field of the STACK parameter block (STPB) contains the address of the new ECT. The ECTIOWA field in the ECT contains the address of the newly created stack. The STACK service routine initializes the first (bottom) element of the new stack. This bottom element is the same as the bottom element of the model stack.

Note:

1. If you create a TSO/E I/O environment, and if you want to run REXX execs in that environment, you *must* create a new REXX environment by calling IRXINIT. See *z/OS TSO/E REXX Reference* for information on calling IRXINIT. Similarly, before you terminate the new TSO/E environment, you must end the new REXX environment by calling IRXTERM.
2. The CREATE operand creates an ALTLIB environment in which only the system CLIST library (ddname SYSPROC) and possibly the system REXX library (ddname SYSEXEC, by default) are searched. The ALTLIB environment in the new TSO/E I/O environment is independent of the ALTLIB environment in the model environment.
3. When TSO/E is processing an authorized command, you cannot use an alternate ECT for command output trapping or data stack prompting.

DESTROY
Specifies that an existing TSO/E I/O environment is to be destroyed. The environment to be destroyed must have been created by the ENVIRON=CREATE function. To destroy an existing I/O environment, follow these steps:

1. Set the IOPLECT field in the input/output parameter list (IOPL) to the address of the ECT associated with the environment to be destroyed. The IOPL is described in "The Input/Output Parameter List" on page 174.
2. Invoke the STACK service routine, specifying the ENVIRON=DESTROY operand.

Note:

1. You cannot destroy an I/O environment at one task level while another task is using the I/O environment, even at the task level that created the I/O environment. If you attempt to do so, the STACK service routine issues abend code X'66D'.
2. When you destroy an I/O environment, the ECT address and the input stack address, ECTIOWA, must be the same as when the I/O environment was created. If you attempt to destroy an I/O environment when the addresses are not the same, the STACK service routine passes a return code of 76 to the application program.

RESET
Specifies that all elements, including barrier elements, are to be removed from the input stack of the current environment. However, the first element on the input stack is not removed.

INQUIRE=
Returns a code that indicates:

- Whether there is a CLIST attention routine to run.
- Whether there is a CLIST error routine to run.
- The type of the topmost element on the input stack.

See "Return Codes from STACK" on page 190 for the meaning of each of the return codes.

ATTN
Returns a code that indicates whether a CLIST attention routine is present anywhere in the current substack.

ERROR
Returns a code that indicates whether a CLIST error routine is present in the top element of the current substack.

TYPE
Returns a code that indicates the type of the topmost element on the input stack.

STORAGE=*element address*
Adds an in-storage element to the input stack. The element address is the address of the list source descriptor (LSD). The LSD is a control block, pointed to by the STACK parameter block, which describes the in-storage list. The LSD must reside below 16 MB in virtual storage. See "Building the List Source Descriptor (LSD)" on page 189 for a description of the LSD.

The in-storage element must be further defined as a SOURCE, PROCN, or PROCL list. SOURCE is the default.

PROMPT
Specifies prompting by commands within a command procedure. PROMPT is used with the keywords PROCN and PROCL, which specify that the element to be added to the input stack is a command procedure.

PROCN
The element to be added to the input stack is a command procedure and the NOLIST option has been specified.

PROCL
The element to be added to the input stack is a command procedure and the LIST option has been specified. Each line that is read from the command procedure is written to the terminal.

SOURCE
The element to be added to the input stack is an in-storage source data set.

MF=L
Indicates that this is the list form of the macro instruction. This operand is required.

DATASET=
Expands the facilities of data set I/O for TSO/E commands to include reading from a SYSIN data set and writing to a SYSOUT data set. To use the data set function, the input and output files that are passed to the STACK service routine must be preallocated, either by a previously issued ALLOCATE command, a command processor that invokes dynamic allocation, a DD statement specified in the logon procedure, or, in the background, a user-supplied DD statement.

Enclose the DATASET operands in a pair of parentheses, unless there is only one value and it is * or CLOSE.

Enclosed in parentheses, you can code the following in any order:

*
INDD=addr1
PROMPT - This has an effect only if you also code INDD=
LIST - This has an effect only if you also code INDD=
MEMBER=addr2
CNT - This has an effect only if you also code MEMBER=
SEQ - This has an effect only if you also code MEMBER=
CLOSE - If you code this, it must be first.

*****
Specifies that STACK use the bottom element in the input stack for I/O operations. This operand is the functional equivalent of TERM=*.

INDD=*addr1*
Specifies the input file name.

PROMPT
Allows prompting if prompting is also allowed on the bottom element of the input stack.

LIST
Causes records from the identified data set in the input stream to be listed on the terminal.

MEMBER=*addr3*
Specifies an 8-character member name for a partitioned data set which was specified as the input file with the INDD operand.

OUTDD=*addr2*
Specifies the output file name.

CNTL
The output line has its own control character.

CLOSE
Closes the data control blocks (DCBs) of the input stack. These DCBs are created by the STACK service routine. Your program cannot modify the DCBs directly; you must invoke the STACK service routine to modify these control blocks.

SEQ
Indicates to data set I/O that sequence numbers should not be removed.

Notes:

1. INDD and OUTDD are only valid if the associated data set element is the last element that is stacked on the TSO/E I/O stack. If any element, such as a CLIST element, or elements from invoking the TSO/E service facility are stacked after the data set element, INDD and OUTDD will become incorrect. This causes the I/O to be routed to the bottom element on the I/O stack. This is the functional equivalent of DATASET=*, or TERM=*, and refers to SYSTSIN and SYSTSPRT.
2. PUTLINE/PUTGET use the OUTPUT data set instead of the terminal, if OUTDD is used.

The Execute Form of the STACK macro instruction

Use the execute form of the STACK macro instruction to perform the following functions:

- Set up the input/output parameter list (IOPL).
- Initialize those fields of the STACK parameter block (STPB) that are not initialized by the list form of the macro instruction, or to modify those fields already initialized.
- Pass control to the STACK service routine, which modifies the input stack.

The operands that you specify in the execute form of the STACK macro instruction are used to set up control information used by the STACK service routine. You can use the PARM, UPT, ECT, and ECB operands of the STACK macro instruction to complete, build, or alter an IOPL.

In the execute form of the STACK macro instruction only the following operands are required:

```
STACK      MF=(E,{list address})
                 {    (1)     }
```

The PARM, UPT, ECT, and ECB operands are not required if you have built an IOPL in your own code.

You are not required to specify the ENTRY operand.

The other operands and their sublists are optional because they can be supplied by the list form of the macro instruction.

Figure 82 on page 182 shows the execute form of the STACK macro instruction; each of the operands is explained following the figure.

```
   [symbol]      STACK     [[PARM=parm addr.][,UPT=upt addr.]         ]
                           [[,ECT=ect addr.][,ECB=ecb addr. ]         ]

                           [ TERM=*                                   ]
                           [ BARRIER={*     }                         ]
                           [         {NONEST}                         ]
                           [        { TOP    }                        ]
                           [ DELETE={ PROC   }                        ]
                           [        { ALL    }                        ]
                           [        { BARRIER}                        ]
                           [ ENVIRON= {CREATE  }                      ]
                           [          {DESTROY }                      ]
                           [          {RESET   }                      ]
                           [ INQUIRE= {ATTN  }                        ]
                           [          {ERROR }                        ]
                           [          {TYPE  }                        ]
                           [                          {PROCN,PROMPT}  ]
                           [ STORAGE=(element addr., {PROCL,PROMPT})  ]
                           [                          {SOURCE      }  ]
                           [         { *                       }      ]
                           [         { INDD=add1,PROMPT,LIST   }      ]
                           [ DATASET={ MEMBER=addr3            }      ]
                           [         { OUTDD=addr2,CNTL,SEQ    }      ]
                           [         { CLOSE                   }      ]
                           [,ENTRY= {entry addr.}],MF=(E,{list addr.})]
                           [        {   (15)    }]       {   (1)    } ]
```

Figure 82. The Execute Form of the STACK macro instruction

PARM=*parm addr*
Specifies the address of the 6-word STACK parameter block (STPB). It can be the address of the list form of the STACK macro instruction. The address is any address valid in an RX instruction, or the number of one of the general registers 2–12 enclosed in parentheses. This address is placed in the input/output parameter list (IOPL). Use the list form of STACK to create the STPB. If no list options are specified, the STPB is zeroed by the list form of the STACK macro instruction.

The STPB and IOPL (STPL) can be modified by STACK, so they should be in reentrant storage if used in a reentrant program.

UPT=*upt addr*
Specifies the address of the user profile table (UPT). This address can be obtained from the Command Processor parameter list (CPPL) pointed to by register 1 when the Command Processor is attached by the terminal monitor program. The address can be any address valid in an RX instruction or the number of one of the generalregisters 2–12 enclosed in parentheses. This address is placed in the input/output parameter list (IOPL).

ECT=*ect addr*
Specifies the address of the environment control table (ECT). This address can be obtained from the Command Processor parameter list (CPPL) pointed to by register 1 when the Command Processor is attached by the terminal monitor program. The address can be any address valid in an RX instruction or the number of one of the general registers 2–12 enclosed in parentheses. This address is placed in the IOPL.

ECB=*ecb addr*
Specifies the address of an event control block (ECB). This address is placed into the IOPL. You must provide a one-word event control block and pass its address to the STACK service routine by placing it into the IOPL. The address can be any address valid in an RX instruction or the number of one of the general registers 2–12 enclosed in parentheses.

TERM=*
Adds a terminal element to the input stack.

Note: TERM=* is allowed by STACK to provide compatibility with existing modules when they are recompiled.

BARRIER=
Creates a barrier element, which divides the input stack into substacks, on top of the input stack.

***
CLISTs and REXX execs on opposing sides of this barrier are nested. They are able to use command output trapping and can communicate through global variables. Command processors can use routines IKJCT441 and IRXEXCOM to access variables on the opposing side of the barrier.

NONEST
CLISTs and REXX execs on opposing sides of the barrier are not nested. This type of barrier halts the effect of command output trapping and halts the use of the routines IKJCT441 and IRXEXCOM to access variables on the opposing side of the barrier. While CLIST global variables are not communicated across this barrier, CLISTs on top of this barrier can begin using global variables and communicate with further nested CLISTs through global variables.

Note: When stacking and removing barrier elements:

1. Only STACK DELETE=BARRIER or STACK ENVIRON=RESET can remove a barrier element.
2. If the application or Command Processor stacks a barrier element, the application or Command Processor must remove the barrier element when it is done using the task. Failure to remove the barrier element can result in miscommunication between the application that invoked your Command Processor and other command processors and applications that use barriers on the current input stack.

DELETE=
Deletes one or more elements from the input stack. TOP, PROC, ALL, or BARRIER specifies which element(s).

TOP
Deletes the topmost element (the element that is most recently added to the input stack). If the top stack element is a barrier element, STACK DELETE=TOP is a no-operation instruction.

PROC
Deletes the current procedure element from the input stack. If the top element is not a procedure element, deletes all elements down to and including the first procedure element.

ALL
Deletes all elements, except the bottom or first element, from the input stack. If one or more barrier elements exist on the input stack, deletes all elements down to, but not including, the first barrier element.

BARRIER
Deletes all elements on the input stack down to, and including, the first barrier element.

ENVIRON=
Specifies one of the following operations:

- A new TSO/E I/O environment is to be created.
- An existing TSO/E I/O environment is to be destroyed.
- The current TSO/E I/O environment is to be reset.

A TSO/E I/O environment consists of the control blocks that describe the input and output sources used by the I/O service routines. These control blocks include the environment control table (ECT) and the input stack.

CREATE
Specifies that a new TSO/E I/O environment is to be created. The STACK service routine creates a new environment using a model environment provided by your Command Processor. To create a new I/O environment, follow these steps:

1. Set the IOPLECT field in the input/output parameter list (IOPL) to the address of the ECT to be used as a model to create a new ECT. The IOPL is described in "The Input/Output Parameter List" on page 174. The ECT that you provide as the model is passed to your Command Processor in the Command Processor parameter list (CPPL). For more information on the CPPL, see "Interfacing with the TSO/E service routines" on page 13.
2. Invoke the STACK service routine, specifying the ENVIRON=CREATE operand.

When the STACK service routine returns control to your Command Processor, the STPBECTA field of the STACK parameter block (STPB) contains the address of the new ECT. The ECTIOWA field in the ECT contains the address of the newly created stack. The STACK service routine initializes the first (bottom) element of the new stack. This bottom element is the same as the bottom element of the model stack.

Notes:

1. If you create a TSO/E I/O environment, and if you want to run REXX execs in that environment, you *must* create a new REXX environment by calling IRXINIT. See *z/OS TSO/E REXX Reference* for information on calling IRXINIT. Similarly, before you terminate the new TSO/E environment, you must end the new REXX environment by calling IRXTERM.
2. The CREATE operand creates an ALTLIB environment in which only the system CLIST library (ddname SYSPROC) and possibly the system REXX library (ddname SYSEXEC, by default) are searched. The ALTLIB environment in the new TSO/E I/O environment is independent of the ALTLIB environment in the model environment.

DESTROY
Specifies that an existing TSO/E I/O environment is to be destroyed. The environment to be destroyed must have been created by the ENVIRON=CREATE function. To destroy an existing I/O environment, follow these steps:

1. Set the IOPLECT field in the input/output parameter list (IOPL) to the address of the ECT associated with the environment to be destroyed. The IOPL is described in "The Input/Output Parameter List" on page 174.
2. Invoke the STACK service routine, specifying the ENVIRON=DESTROY operand.

Note:

1. You cannot destroy an I/O environment at one task level while another task is using the I/O environment, even at the task level that created the I/O environment. If you attempt to do so, the STACK service routine issues abend code X'66D'.

2. When you destroy an I/O environment, the ECT address and the input stack address, ECTIOWA, must be the same as when the I/O environment was created. If you attempt to destroy an I/O environment when the addresses are not the same, the STACK service routine passes a return code of 76 to the application program.
3. When TSO/E is processing an authorized command, you cannot use an alternate ECT for command output trapping or data stack prompting.

RESET
Specifies that all elements, including barrier elements, are to be removed from the input stack of the current environment. However, the first element on the input stack is not removed. Use this function only when a severe error condition requires your program to terminate all processing and cause the READY mode message to be issued.

INQUIRE=
Returns a code that indicates:

- Whether there is a CLIST attention routine to run.
- Whether there is a CLIST error routine to run.
- The type of the topmost element on the input stack.

See "Return Codes from STACK" on page 190 for the meaning of each of the return codes.

ATTN
Returns a code that indicates whether a CLIST attention routine is present anywhere in the current substack. Issue the STACK macro in the form:

```
(symbol) STACK INQUIRE=ATTN,MF=(E,ABC)
```

ERROR
Returns a code that indicates whether a CLIST error routine is present in the top element of the current substack. Issue the STACK macro in the form:

```
(symbol) STACK INQUIRE=ERROR,MF=(E,(ABC))
```

TYPE
Returns a code that indicates the type of the topmost element on the input stack.

STORAGE=*element address*
Adds an in-storage element to the input stack. The element address is the address of the list source descriptor (LSD). The LSD is a control block, pointed to by the stack parameter block, which describes the in-storage list. See "Building the List Source Descriptor (LSD)" on page 189 for a description of the LSD. The in-storage list must be further defined as a SOURCE, PROCN, or PROCL list. SOURCE is the default.

SOURCE
The element to be added to the input stack is an in-storage source data set.

PROCN
The element to be added to the input stack is a command procedure and the NOLIST option has been specified.

PROCL
The element to be added to the input stack is a command procedure and the LIST option has been specified. Each line that is read from the command procedure is written to the terminal.

PROMPT
Specifies prompting by commands within a command procedure. PROMPT is used with the keywords PROCN and PROCL, which specify that the element to be added to the input stack is a command procedure.

DATASET=
Expands the facilities of data set I/O for TSO/E commands to include reading from a SYSIN data set and writing to a SYSOUT data set. To use the data set function, the input and output files passed to the STACK service routine must be preallocated, either by a previously issued ALLOCATE

command, a command processor that invokes dynamic allocation, a DD statement specified in the logon procedure, or, in the background, a user-supplied DD statement.

*****
Specifies that STACK use the bottom element in the input stack for I/O operations. This operand is the functional equivalent of TERM=*.

INDD=*addr1*
Specifies the input file name.

PROMPT
Allows prompting if prompting is also allowed on the bottom element of the input stack.

LIST
Lists the input from the input stream.

MEMBER=*addr3*
Specifies an 8-character member name for a partitioned data set which was specified as the input file with the INDD operand.

OUTDD=*addr2*
Specifies the output file name.

CNTL
The output line has its own control character.

CLOSE
Closes the data control blocks (DCBs) of the input stack. These DCBs are created by the STACK service routine. Your program cannot modify the DCBs directly; you must invoke the STACK service routine to modify these control blocks.

SEQ
Indicates to data set I/O that sequence numbers should not be removed.

Note: INDD and OUTDD are only valid if the associated data set element is the last element that is stacked on the TSO/E I/O stack. If any element such as a CLIST element, or elements from invoking the TSO/E service facility are stacked after the DATASET element, INDD and OUTDD will become incorrect. This causes the I/O to be routed to the bottom element on the I/O stack. This is the functional equivalent of DATASET=*, or TERM=*, and refers to SYSTSIN and SYSTSPRT.

ENTRY=*entry address* | (15)
Specifies the entry point of the STACK service routine. The address can be any address valid in an RX instruction or (15) if the entry point address has been loaded into general register 15. If ENTRY is omitted, a LINK macro instruction is generated to invoke the STACK service routine.

MF=E
Indicates that this is the execute form of the macro instruction.

***listaddr* | (1)**
The address of the four-word input/output parameter list (IOPL). This can be a completed IOPL that you have built, or it can be 4 words of declared storage that is filled from the PARM, UPT, ECT, and ECB operands of this execute form of the STACK macro instruction. The address is any address valid in an RX instruction or (1) if the parameter list address has been loaded into general register 1.

The Sources of Input

There are two types of input sources you can add to the input stack: the terminal and an in-storage list.

Terminal

If the terminal is specified in the STACK macro instruction as the input source, all input and output requests through GETLINE, PUTLINE, and PUTGET are read from the terminal and written to the terminal. The user at the terminal controls TSO/E by entering commands; the system processes these commands as they are entered and returns to the user for another command.

When an online job is running, the first element in the input stack is a terminal element.

In-Storage List

An in-storage list can be either a list of commands or a source data set. It can contain variable-length records (with a length header) or fixed-length records (no header and all records the same length). In either case, no one record on an in-storage list can exceed 256 characters.

When a job is running in the background, the first element in the input stack is a data set element.

Specify an in-storage list and its processing by setting the STORAGE operand type to PROCN, PROCL, or SOURCE.

- PROCN or PROCL - Indicates that the in-storage list is a command procedure, which is a list of commands to be executed in the order specified.

  If you specify PROCN, requests through GETLINE are read from the in-storage list, but PROMPT requests from the executing Command Processor are suppressed. MODE messages, those messages normally sent to the terminal requesting entry of a command or a subcommand, are not sent; instead, a command is obtained from the in-storage list.

  If the PROCL option is specified, the command is displayed at the terminal as it is read from the list.
- SOURCE - Indicates that the in-storage list is a source data set. Requests through GETLINE are read from the in-storage list, but PROMPT requests from the executing Command Processor are honored if prompting is allowed, and a line is requested from the terminal. MODE messages are handled the same way as with PROCN or PROCL. No LIST facility is provided with SOURCE records.

If your Command Processor uses the STACK service routine to specify an in-storage list as the input source, you should create the in-storage list in subpool 78. However, if your Command Processor uses the STACK service routine to place either a data set or an in-storage list that is *not* in subpool 78 on the input stack, the Command Processor must remove the stack element before termination. To remove the stack element, your Command Processor should either:

- Issue the STACK macro instruction with the DELETE=TOP operand specified.
- Use the GETLINE or PUTGET service routine to process input until end-of-input is reached.

To add an in-storage list element to the input stack, you must build a list storage descriptor (LSD), which contains a description of the in-storage list, and pass its address to the STACK service routine. The STACK service routine then adds the in-storage list element to the input stack. The LSD is described in “Building the List Source Descriptor (LSD)” on page 189.

For an example showing how to use the STACK service routine to specify an in-storage list as the input source, see Figure 86 on page 195.

Building the STACK Parameter Block (STPB)

When the list form of the STACK macro instruction expands, it builds a 5-word STACK parameter block (STPB). The list form of the macro instruction initializes this STPB according to the operands you have coded. This initialized block, which you can later modify with the execute form of the macro instruction, indicates to the I/O service routine the functions you want performed.

By using the list form of the macro instruction to initialize the block, and the execute form to modify it, you can use the same STPB to perform different STACK functions. Keep in mind, however, that if you specify an operand in the execute form of the macro instruction, and that operand has a sublist as a value, the default values of the sublist will be coded into the STPB for any of the sublist values not coded. If you do not want the default values, you must code each of the values you require, each time you change any one of them.

For example, if you coded the list form of the STACK macro instruction as follows:

```
STACK      STORAGE=(element address,PROCN),MF=L
```

and then overrode it with the execute form of the macro instruction as follows:

```
STACK      STORAGE=(new element address),
           MF=(E,list address)
```

The element code in the STACK parameter block would default to SOURCE, the default value. If the new in-storage list was another PROCN list, you would have to respecify PROCN in the execute form of the macro instruction.

The STACK parameter block is defined by the IKJSTPB DSECT, which is provided in SYS1.MACLIB. Table 57 on page 188 describes the contents of the STPB.

Table 57. The STACK parameter block

Number of bytes	Field name	Contents or meaning
1	none	Operation code. A flag byte which describes the operation to be performed: **1... ....** One element is to be added to the top of the input stack. **.1.. ....** The top element is to be deleted from the input stack. **..1. ....** The current procedure is to be deleted from the input stack. If the top element is not a PROC element, all elements down to and including the first PROC element encountered are deleted, except the bottom element. **...1 ....** All elements except the bottom one (the first element) are to be deleted. **.... 1...** A barrier element is to be deleted from the input stack. **.... .1..** Determine whether there is a CLIST attention routine to run. **.... ..1.** Determine whether there is a CLIST error routine to run. **.... ...1** Determine the type of the topmost element on the input stack.
1	none	Element code. A flag byte describing the element to be added to the input stack: **1... ....** A terminal element. **.1.. ....** An in-storage element. **..1. ....** Input ddname present. **...1 ....** Output ddname present. **.... 1...** The in-storage element is an EXEC command element. **.... .1..** Prompting is allowed from the PROC element. **.... ..0.** The in-storage element is a source element. **.... ..1.** The in-storage element is a procedure element. **.... ...1** The list option (PROCL) has been specified.

Table 57. The STACK parameter block (continued)

Number of bytes	Field name	Contents or meaning
1	none	A flag byte describing the operation to be performed: **1... ....** A barrier element is to be added to the input stack. **.1.. ....** Create a new I/O environment. **..1. ....** Destroy the current I/O environment or the one requested by the caller. **...1 ....** Reset the input stack of the current environment. **.... xxxx** Reserved.
1	none	DATASET operation: **xxxx ....** Reserved. **.... 1...** Use BPAM with member. **.... .1..** Do not remove sequence numbers. **.... ..1.** User-specified CNTL. **.... ...1** Close option.
4	STPBALSD	The address of the list source descriptor (LSD). An LSD describes an in-storage list. If the input source is the terminal, or if DELETE has been specified, this field will contain zeros.
4	STPBINDD	Pointer to input ddname.
4	STPBODDN	Pointer to output ddname.
4	STPBMBRN	Pointer to membername.
4	STPBECTA	Pointer to the environment control table (ECT) created by the STACK service routine when ENVIRON=CREATE is specified.

If the DATASET or DELETE operands have been coded in the STACK macro instruction, the second word of the stack parameter block, the STPBALSD field, will contain zeroes and the control block structure will end with the STPB. Figure 83 on page 192 describes this condition.

To add an in-storage list element to the input stack, you must describe the in-storage list and pass a pointer to it to the STACK I/O service routine. You do this by building a list source descriptor (LSD).

Building the List Source Descriptor (LSD)

A list source descriptor (LSD) is a four-word control block that describes the in-storage list pointed to by the new element you are adding to the input stack. Note that the LSD must reside below 16 MB in virtual storage.

If you are designating the terminal as the input source, no LSD is necessary and the second word of the STPB will be zero. If you specify STORAGE as the input source in the STACK macro instruction, your code must build an LSD, and place a pointer to it as a sublist of the STORAGE operand.

The LSD must begin on a doubleword boundary, and must be created in subpool 78. Your Command Processor cannot modify the LSD after it is passed to the STACK service routine.

The LSD is defined by the IKJLSD DSECT, which is provided in SYS1.MACLIB. Table 58 on page 190 describes the contents of the LSD.

Table 58. The list source descriptor

Number of bytes	Field name	Contents or meaning
4	LSDADATA	The address of the in-storage list.
2	LSDRCLEN	The record length if the in-storage list contains fixed-length records. Zero if the record lengths are variable.
2	LSDTOTLN	The total length of the in-storage list; the sum of the lengths of all records in the list.
4	LSDANEXT	Pointer to the next record to be processed. Initialize this field to the address of the first record in the list. The field is updated by the GETLINE and PUTGET service routines.
4	LSDRSVRD	Reserved.

If you have provided an LSD, and specified the STORAGE operand in the STACK macro instruction, the second word of the stack parameter block will contain the address of the LSD, and the STACK control block structure will be as shown in Figure 84 on page 193.

Return Codes from STACK

When the STACK service routine returns to the program that invoked it, STACK provides one of the following return codes in general register 15:

Table 59. Return codes from the STACK service routine

Return code dec(Hex)	Meaning
0(0)	STACK has completed successfully.
4(4)	One or more of the parameters passed to STACK were not valid.
8(8)	INDD was specified and the file could not be opened.
12(C)	OUTDD was specified and the file could not be opened.
16(10)	MEMBER was specified but was not in the partitioned data set specified by INDD.
20(14)	GETMAIN failure (only possible if MEMBER is specified).
24(18)	One of the following occurred: • The INQUIRE=ATTN operand was specified and a CLIST attention routine is present in the current substack. • The INQUIRE=ERROR operand was specified and a CLIST error routine is present in the top element of the current substack.
28(1C)	One of the following occurred: • The INQUIRE=ATTN operand was specified and a CLIST attention routine is *not* present in the current substack. • The INQUIRE=ERROR operand was specified and a CLIST error routine is *not* present in the top element of the current substack.
32(20)	The INQUIRE=TYPE operand was specified and the topmost stack element is a terminal element.
36(24)	The INQUIRE=TYPE operand was specified and the topmost stack element is an in-storage list.
40(28)	The INQUIRE=TYPE operand was specified and the topmost stack element is a command procedure.

Table 59. Return codes from the STACK service routine (continued)

Return code dec(Hex)	Meaning
44(2C)	The INQUIRE=TYPE operand was specified and the topmost stack element is a BARRIER=* element.
48(30)	The INQUIRE=TYPE operand was specified and the topmost stack element is an input file name.
52(34)	The INQUIRE=TYPE operand was specified and the topmost stack element is an output file name.
56(38)	The INQUIRE=TYPE operand was specified and the topmost stack element has both an input file name and an output file name specified.
60(3C)	The INQUIRE=TYPE operand was specified and the topmost stack element is a TERMIN or TERMING element.
64(40)	The INQUIRE=TYPE operand was specified and the topmost stack element is an unknown element.
68(44)	The INQUIRE=TYPE operand was specified and the topmost stack element is a REXX element.
72(48)	A request to add a barrier element to the input stack contains a not valid STACK parameter block. This is a probable user error caused when the STACK macro is not used to invoke the STACK service. To correct the error, the terminal element bit in the element code byte of the STACK parameter block (described above) must be ON when requesting to add a barrier element to the input stack.
76(4C)	The ENVIRON option was specified, but an error occurred when creating or destroying an I/O environment.
80(50)	The INQUIRE=TYPE operand was specified and the topmost stack element is a BARRIER=NONEST element.

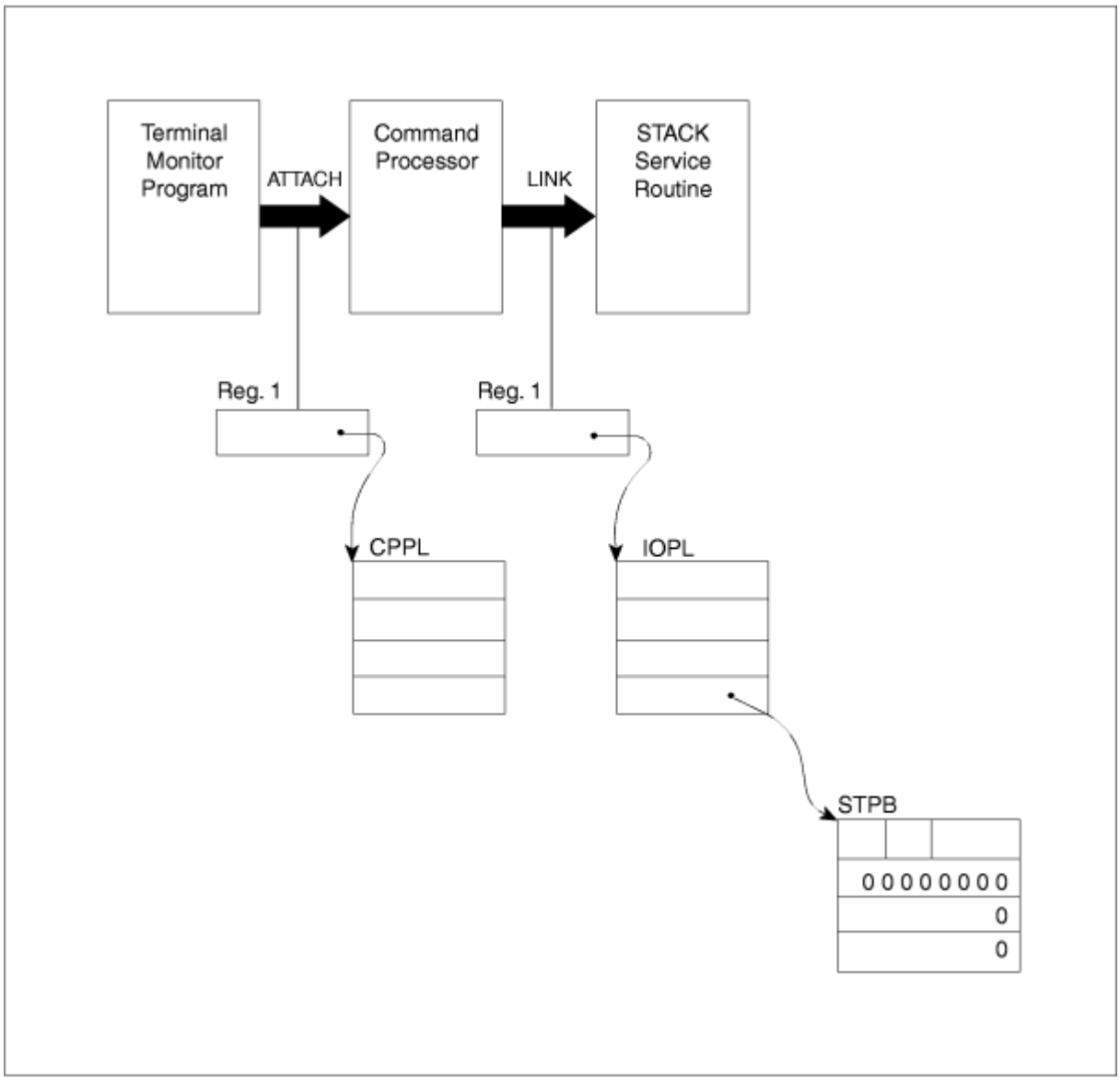


Figure 83. STACK Control Blocks: No In-Storage List

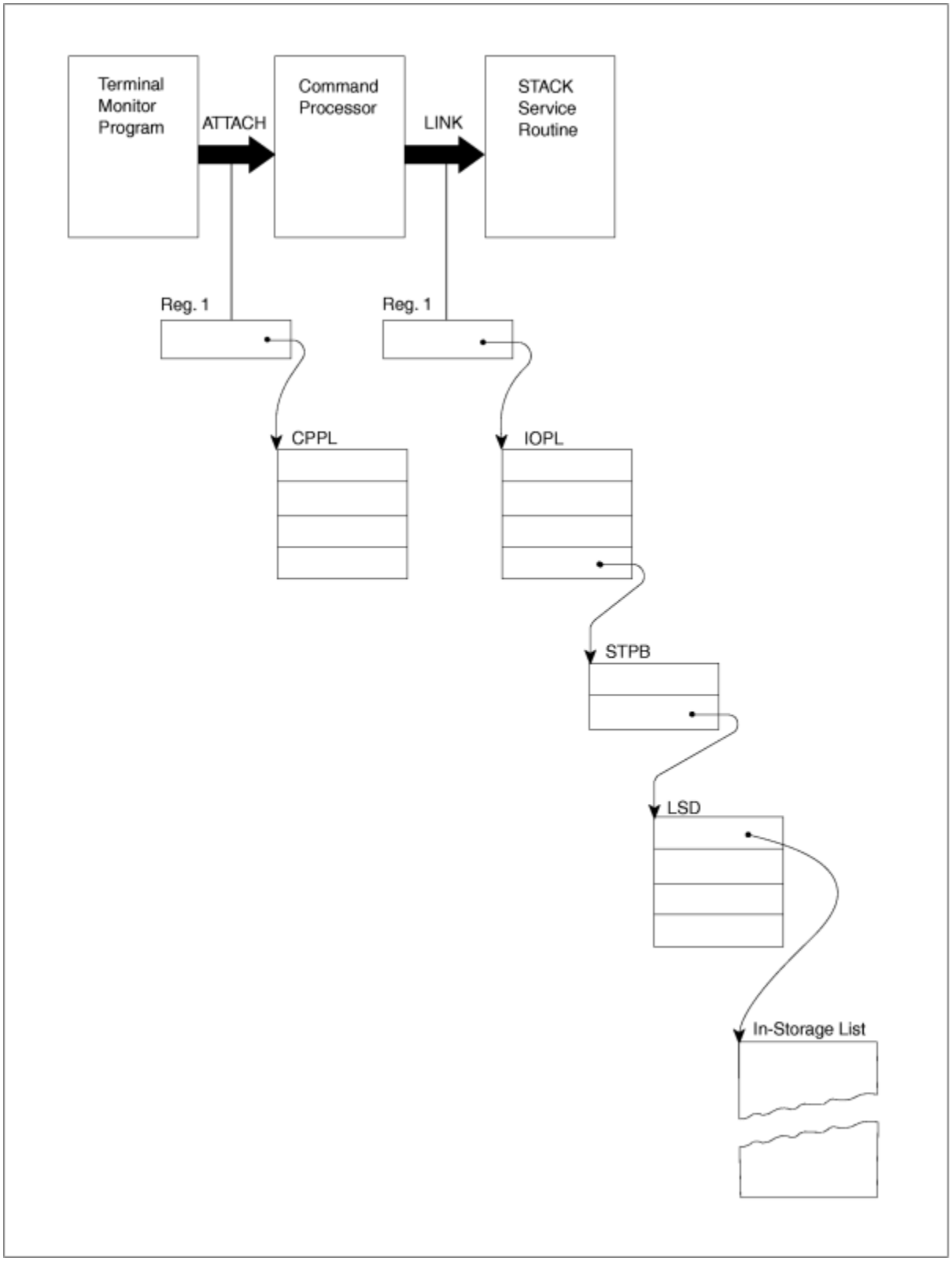


Figure 84. STACK Control Blocks: In-Storage List Specified

Examples Using STACK

Example 1

Figure 85 on page 194 is an example of the code required to add the terminal to the input stack as the current input source. In this example, the execute form of the STACK macro instruction is used to build the input/output parameter list for you. The list form of the STACK macro instruction expands into a STACK parameter block, and its address is passed to the execute form of the macro instruction as the PARM operand address.

Note: This sequence of code does not make use of the IKJCPPL DSECT to access the Command Processor parameter list, nor does it provide reentrant code.

```
*  ENTRY FROM THE TMP - REGISTER ONE CONTAINS A POINTER TO THE CPPL
*
*      SET UP ADDRESSABILITY
*      PERFORM SAVE AREA CHAINING
*               .
*               .
*               .
*
         LR    2,1                   SAVE THE ADDRESS OF THE CPPL
         L     3,4(2)                PLACE THE UPT ADDRESS INTO A
*                                    REGISTER
         L     4,12(2)               PLACE THE ECT ADDRESS INTO A
*                                    REGISTER
         L     5,ECB                 PLACE THE ECB ADDRESS INTO A
*                                    REGISTER
* ISSUE THE EXECUTE FORM OF THE STACK MACRO INSTRUCTION, SPECIFY
* THE TERMINAL AS THE INPUT SOURCE AND BUILD THE IOPL WITH THE
* STACK MACRO INSTRUCTION.
*
         STACK PARM STAKBLOK,UPT=(3),ECT=(4),ECB=(5),TERM=*,MF=(E,IOPL)
*
*        PROCESSING
*        STORAGE DECLARATIONS
*               .
*               .
*               .
IOPL     DC    4F'0'                 SPACE FOR THE INPUT/OUTPUT
*                                    PARAMETER LIST
ECB      DC    F'0'                  SPACE FOR THE EVENT CONTROL
*                                    BLOCK
STAKBLOK STACK MF=L                  THE LIST FORM OF THE STACK MACRO
                                     INSTRUCTION, WHICH WILL EXPAND
                                     INTO A STACK PARAMETER BLOCK
         END
```

Figure 85. Example of STACK Specifying the Terminal as the Input Source

Example 2

Figure 86 on page 195 is an example of the code required to use the STACK macro instruction to place a pointer to an in-storage list on the input stack.

In the example, the GETMAIN macro instruction is used to obtain storage in subpool 78 for the list source descriptor and the in-storage list itself. The execute form of the STACK macro instruction initializes the input/output parameter list required by the STACK service routine. The list form of the STACK macro instruction expands into a STACK parameter block, and its address is passed to the STACK service routine via the PARM operand in the execute form of the STACK macro instruction.

```
*  THIS CODE ASSUMES ENTRY FROM THE TMP - REGISTER ONE CONTAINS THE
* ADDRESS OF THE COMMAND PROCESSOR PARAMETER LIST.
*
*      SET UP ADDRESSABILITY
*      PERFORM SAVE AREA CHAINING
*               .
*               .
*               .
*
         LR    2,1                    SAVE THE ADDRESS OF THE CPPL
         USING CPPL,2                 SET UP ADDRESSABILITY FOR THE
*                                     CPPL
         L     3,CPPLUPT              PLACE THE ECT ADDRESS INTO A
*                                     REGISTER
         L     4,CPPLECT              PLACE THE ECB ADDRESS INTO A
*                                     REGISTER
* ISSUE A GETMAIN FOR SUBPOOL 78.  THE LIST SOURCE DESCRIPTOR AND THE
* IN-STORAGE LIST ITSELF MUST BE LOCATED IN SUBPOOL 78.
*
         GETMAIN    LU,LA=REQUEST,A=ANSWER,SP=78,LOC=BELOW
*
* OBTAIN THE ADDRESS IN SUBPOOL 78 FOR THE LIST SOURCE DESCRIPTOR
* AND MOVE THE LSD INTO THAT AREA.
*
         L     5,ANSWER
         MVC   0(16,5),ANLSD
*
* OBTAIN THE ADDRESS IN SUBPOOL 78 FOR THE IN-STORAGE LIST AND MOVE
* THE IN-STORAGE LIST INTO THAT AREA
*
         L     6,ANSWER+4
         ST    6,0(5)                 STORE THE ADDRESS OF THE IN-
         ST    6,8(5)                 STORAGE LIST INTO TWO FIELDS
*                                     IN THE LIST SOURCE DESCRIPTOR
         MVC   0(100,6),INLIST
*
* ISSUE AN EXECUTE FORM OF THE STACK MACRO INSTRUCTION TO PUT A
* POINTER TO THE IN-STORAGE LIST ON THE INPUT STACK.
*
         STACK PARM=STCKLST,UPT=(3),ECT=(4),ECB=ECBADS,               X
               STORAGE=((5),PROCN),MF=(E,IOPLADS)
*
* TEST THE RETURN CODE FOR SUCCESSFUL COMPLETION OF THE STACK
* SERVICE ROUTINE.
*
         LTR   15,15
         BNZ   ERRTN
```

Figure 86. Example of STACK Specifying an In-storage List as the Input Source

Figure of 'Example of STACK Specifying an In-storage List as the Input Source' (**Continued**)

```
*            .
*            .
ERRTN
*            .
*            .
*            .
* STORAGE DECLARATIONS
*
ANLSD    DS    A                     THE TOTAL LENGTH OF THE LIST
         DC    X'0000'               SOURCE DESCRIPTOR, ANLSD, IS
         DC    X'0064'               16 BYTES (DECIMAL).
         DS    A
         DC    F'0'
*
INLIST   DC    X'00140000'
         DC    C'EDIT OPA OPB OPC'
         DC    X'00180000'
         DC    C'TEST OPTA OPTB OPTC '
         DC    X'00240000'
         DC    C'PROFILE     NOMSGID    NOPROMPT'
         DC    X'00140000'
         DC    C'EXEC MYPROG LIST'
*
```

```
* THE TOTAL LENGTH OF THE IN-STORAGE LIST, INLIST, IS 100 DECIMAL
* BYTES.
*
* SET UP THE LIST OF STORAGE AMOUNTS REQUIRED.  THE ADDRESS OF THIS
* LIST IS CODED AS THE LA= OPERAND ON THE GETMAIN MACRO INSTRUCTION.
*
REQUEST  DC    F'16'                 SIXTEEN BYTES FOR THE LSD.
         DC    X'80'                 END OF LIST INDICATOR
         DC    AL3(104)              100 BYTES FOR THE IN-STORAGE LIST
*                                    SINCE THE GETMAIN MACRO REQUIRES
*                                    THAT THE REQUEST BE DIVISIBLE BY
*                                    8, WE REQUEST 104 BYTES.
*
* SET ASIDE 2 FULLWORDS TO RECEIVE THE ADDRESS RETURNED BY THE GETMAIN
* MACRO INSTRUCTION.
*
ANSWER   DC    2F'0'
*
STCKLST  STACK MF=L                  THIS LIST FORM OF THE STACK
*                                    MACRO INSTRUCTION PROVIDES SPACE
*                                    FOR THE STACK PARAMETER BLOCK
*
ECBADS   DC    F'0'                  EVENT CONTROL BLOCK
IOPLADS  DC    4F'0'                 INPUT/OUTPUT PARAMETER LIST
         IKJCPPL                     DSECT FOR THE COMMAND PROCESSOR
*                                    PARAMETER LIST
         END
```

Example 3

Figure 87 on page 197 is an example of the code required to use the STACK macro instruction to create a new TSO/E I/O environment.

```
* Instructions
MAINLINE DS    OH
         USING CPPL,R1
         L     R10,CPPLUPT
         ST    R10,UPTP               Store caller UPT pointer in
*                                     dynamic area
         L     R10,CPPLPSCB
         ST    R10,PSCBP              Store caller PSCB pointer in
*                                     dynamic area
         L     R10,CPPLECT
         ST    R10,ECTP               Store caller ECT pointer in
*                                     dynamic area
*********************************************************************
*    Create a new ECT                                               *
*********************************************************************
         LA    R10,STCKLSTD
         ST    R10,STPBPTR            Basing for the STPB mapping
         USING STPB,R10
         L     R2,UPTP                R2 points to UPT for STACK macro
         L     R3,ECTP                R3 points to ECT for STACK macro
         XC    ECBSTCK,ECBSTCK        Clear fullword containing ECB
         LA    R4,ECBSTCK             R4 points to ECB for STACK macro
         L     R14,STCKLST1L          Move the
         BCTR  R14,0                    static copy of
         EX    R14,MVCTARGT               STACK to the dynamic copy
         LA    R6,STCKLSTD            R6 points to dynamic copy of
*                                       STACK
         STACK PARM=(R6),UPT=(R2),ECT=(R3),ECB=(R4),                   +
               ENVIRON=CREATE,                                         +
               MF=(E,STCKIOPL)
         L     R9,STPBECTA
         ST    R9,ECTP                Store new ECT pointer in
*                                     dynamic area
         DROP  R10

* Static Storage Declarations
STCKLST1 STACK MF=L
STCKLST1L DC   A(*-STCKLST1)
MVCTARGT MVC   STCKLSTD(0),STCKLST1

* Dynamic Storage Declarations
UPTP     DS    AL4                    Address of the UPT
PSCBP    DS    AL4                    Address of the PSCB
ECTP     DS    AL4                    Address of the ECT
STPBPTR  DS    AL4                    Pointer to STACK parameter block
STCKLSTD STACK MF=L                   Dynamic form of STACK
         DS    OF                     Reach a fullword boundary for ECB
ECBSTCK  DS    BL32                   STACK ECB

* Control Block Mappings
            IKJCPPL                  CPPL mapping
            IKJSTPB                  STACK Parameter Block mapping
```

Figure 87. Example of STACK Creating a New TSO/E I/O Environment

Using GETLINE to Get a Line of Input

Use the GETLINE macro instruction to obtain all input lines other than commands, subcommands, and prompt message responses. Commands, subcommands, and prompt message responses should be obtained with the PUTGET macro instruction.

When you issue a GETLINE macro instruction, the GETLINE service routine obtains a line of input from either:

- The terminal or the REXX data stack
- An in-storage list (including a command procedure)

The processing of the input line varies according to several factors. Included in these factors are the source of input, and the options you specify for logical or physical processing of the input line. The GETLINE service routine determines the type of processing to be performed from the operands coded on the GETLINE macro instruction, and returns a line of input.

The sections that follow describe the following topics:

- The list and execute forms of the GETLINE macro instruction
- The sources of input
- The GETLINE parameter block
- The input line format
- Return codes from GETLINE

The List Form of the GETLINE macro instruction

The list form of the GETLINE macro instruction builds and initializes a GETLINE parameter block (GTPB), according to the operands you specify in the GETLINE macro. The GETLINE parameter block indicates to the GETLINE service routine which functions you want performed.

In the list form of the macro instruction, only

```
GETLINE      MF=L
```

is required. The other operands and their sublists are optional because they can be supplied by the execute form of the macro instruction, or automatically supplied if you want the default values.

The operands you specify in the list form of the GETLINE macro instruction set up control information used by the GETLINE service routine. The INPUT and TERMGET operands set bits in the GETLINE parameter block to indicate to the GETLINE service routine which options you want performed.

Figure 88 on page 198 shows the list form of the GETLINE macro instruction; each of the operands is explained following the figure.

```
[symbol]     GETLINE       [INPUT=({ISTACK} {,LOGICAL})]
                           [       {TERM  } {,PHYSICAL}]

                           [,TERMGET=({EDIT}  {,WAIT }),]
                           [          {ASIS}  {,NOWAIT}],MF=L

                           [,SUBSTACK=({NO } ) ]
                           [           {YES}   ]
```

Figure 88. The List Form of the GETLINE macro instruction

INPUT=
indicates that an input line is to be obtained. This input line is further described by the INPUT sublist operands ISTACK, TERM, LOGICAL, and PHYSICAL. ISTACK and LOGICAL are the default values.

ISTACK | TERM

ISTACK
Obtain an input line from the currently active input source indicated by the input stack.

TERM
indicates to GETLINE that the current source of input, indicated by the top element of the input stack, is to be ignored. Input is to be returned from the REXX data stack (if elements exist), from a CLIST DATA-ENDDATA group, or from the terminal. For more information about how GETLINE determines the source of input, refer to “Sources of Input” on page 202.

LOGICAL | PHYSICAL

LOGICAL
The input line to be obtained is a logical line; the GETLINE service routine is to perform logical line processing.

PHYSICAL
The input line to be obtained is a physical line. The GETLINE service routine need not inspect the input line.

Note: If the input line you are requesting is a logical line coming from the input source indicated by the input stack, you need not code the INPUT operand or its sub-list operands. The input line description defaults to ISTACK, LOGICAL.

TERMGET=
specifies the options requested. The options are EDIT or ASIS, and WAIT or NOWAIT. The default values are EDIT and WAIT.

EDIT | ASIS

EDIT
specifies that in addition to minimal editing (see ASIS), the buffer is to be padded with trailing blanks.

ASIS
specifies that minimal editing is to be done as follows:

- Transmission control characters are removed.
- The line of input is translated from terminal code to EBCDIC.
- Line-deletion and character-deletion editing is performed.
- Line feed and carrier return characters, if present, are removed.

No line continuation checking is done.

WAIT | NOWAIT

WAIT
specifies that control is to be returned to the routine that issued the GETLINE macro instruction only after an input message has been read.

NOWAIT
specifies that control is to be returned to the routine that issued the GETLINE macro instruction whether a line of input is available. If a line of input is not available, a return code of 12 decimal is returned in register 15 to the Command Processor.

MF=L
indicates that this is the list form of the macro instruction.

SUBSTACK=
SUBSTACK=YES indicates that normal stack operations continue until GETLINE reaches a barrier element. GETLINE then passes the caller a return code indicating that a barrier element was reached. The barrier element remains on the stack until the caller explicitly deletes it. SUBSTACK=NO is the default value and indicates that the barrier feature is not used.

Note: If your Command Processor issues GETLINE without SUBSTACK=YES, and a barrier element exists on the input stack, normal stack operations continue until GETLINE reaches a barrier element. In foreground mode, GETLINE then treats the barrier element as a terminal element. In background mode, GETLINE passes an end-of-data return code to the caller. Processing continues in this manner until your Command Processor explicitly deletes the barrier element.

The Execute Form of the GETLINE macro instruction

Use the execute form of the GETLINE macro instruction to perform the following functions:

- Set up the input/output parameter list (IOPL).
- Initialize those fields of the GETLINE parameter block (GTPB) that are not initialized by the list form of the macro instruction, or to modify those fields already initialized.
- Pass control to the GETLINE service routine, which gets the line of input.

In the execute form of the GETLINE macro instruction only the following is required:

```
GETLINE      MF=(E,{list address})
                   {  (1)       }
```

The PARM, UPT, ECT, and ECB operands are not required if you have built your IOPL in your own code. The other operands and their sublists are optional because you can supply them in the list form of the macro instruction or in a previous execution of GETLINE, or because you are using the default values.

The operands you specify in the execute form of the GETLINE macro instruction are used to set up control information used by the GETLINE service routine. You can use the PARM, UPT, ECT, and ECB operands of the GETLINE macro instruction to build, complete, or modify an IOPL. The INPUT and TERMGET operands set bits in the GETLINE parameter block. These bit settings indicate to the GETLINE service routine which options you want performed.

Figure 89 on page 200 shows the execute form of the GETLINE macro instruction; each of the operands is explained following the figure.

```
  [symbol]  GETLINE       [PARM=parameter address]   [,UPT=upt address)
                          [,ECT=ect address][,ECB=ecb address)

                          [,INPUT=( {ISTACK}    {,LOGICAL }) ]
                          [         {TERM  } {,PHYSICAL} ]

                          [,TERMGET=({ EDIT} {,WAIT }) ]
                          [          { ASIS} {,NOWAIT} ]

                          [,ENTRY={entry address}],MF=(E,{list address })
                          [       {    (15)     }]       {    (1)      }

                          [,SUBSTACK=({ NO }) ]
                          [           { YES} ]
```

Figure 89. The Execute Form of the GETLINE macro instruction

PARM=*parameter address*
specifies the address of the 2-word GETLINE parameter block (GTPB). It can be the address of a list form GETLINE macro instruction. The address is any address valid in an RX instruction, or the number of one of the general registers 2–12 enclosed in parentheses. This address will be placed in the input/output parameter list (IOPL).

UPT=*upt address*
specifies the address of the user profile table (UPT). You can obtain this address from the Command Processor parameter list pointed to by register 1 when the Command Processor is attached by the terminal monitor program. The address can be any address valid in an RX instruction or the number of one of the general registers 2–12 enclosed in parentheses. This address will be placed in the IOPL.

ECT=*ect address*
specifies the address of the environment control table (ECT). You can obtain this address from the CPPL pointed to by register 1 when the Command Processor is attached by the terminal monitor program. The address can be any address valid in an RX instruction or the number of one of the general registers 2–12 enclosed in parentheses. This address will be placed into the IOPL.

ECB=*ecb address*
specifies the address of an event control block (ECB). You must provide a one-word event control block and pass its address to the GETLINE service routine by placing it into the IOPL. The address can be any address valid in an RX instruction or the number of one of the general registers 2–12 enclosed in parentheses. This address will be placed into the IOPL.

INPUT=
indicates that an input line is to be obtained. This input line is further described by the INPUT sublist operands ISTACK, TERM, LOGICAL, and PHYSICAL. ISTACK and LOGICAL are the default values.

ISTACK | TERM

ISTACK
obtains an input line from the currently active input source indicated by the input stack.

TERM
indicates to GETLINE that the current source of input, indicated by the top element of the input stack, is to be ignored. Input is to be returned from the REXX data stack (if elements exist), from a CLIST DATA-ENDDATA group, or from the terminal. For more information about how GETLINE determines the source of input, refer to "Sources of Input" on page 202.

LOGICAL | PHYSICAL

LOGICAL
The input line to be obtained is a logical line; the GETLINE service routine is to perform logical line processing. A logical line is a line that has additional processing performed by the GETLINE service routine before it is returned to the requesting program.

PHYSICAL
The input line to be obtained is a physical line. A physical line is a line that is returned to the requesting program exactly as it is received from the input source.

Note: If the input line you are requesting is a logical line coming from the input source indicated by the input stack, you do not need to code the INPUT operand or its sublist operands. The input line description defaults to ISTACK, LOGICAL.

TERMGET=
specifies the options requested. The options are EDIT or ASIS, and WAIT or NOWAIT. The default values are EDIT and WAIT.

EDIT | ASIS

EDIT
specifies that in addition to minimal editing (see ASIS), the input buffer is to be padded with trailing blanks. All station control characters are suppressed from the data.

ASIS
specifies that minimal editing is to be done. The following editing functions will be performed:

- Station control characters remain in the data.
- The line of input is translated from terminal code to EBCDIC.
- Line-deletion and character-deletion editing are performed.
- Line feed and carrier return characters, if present, are removed.

No line continuation checking is done.

WAIT | NOWAIT

WAIT
specifies that control is to be returned to the routine that issued the GETLINE macro instruction, only after an input message has been read.

NOWAIT
specifies that control is to be returned to the routine that issued the GETLINE macro instruction whether a line of input is available. If a line of input is not available, a return code of 12 decimal is returned in register 15 to the Command Processor.

ENTRY=*entry address* | (15)
specifies the entry point of the GETLINE service routine. The address can be any address valid in an RX instruction or (15) if the entry point address has been loaded into general register 15. The ENTRY operand need not be coded in the macro instruction. If it is not, a LINK macro instruction will be generated to invoke the I/O service routine.

MF=E
indicates that this is the execute form of the macro instruction.

***listaddr* | (1)**
The address of the four-word input/output parameter list (IOPL). This can be a completed IOPL that you have built, or it can be 4 words of declared storage that will be filled from the PARM, UPT, ECB, and ECT operands of this execute form of the GETLINE macro instruction. The address is any address valid in an RX instruction or (1) if the parameter list address has been loaded into general register 1.

SUBSTACK=
SUBSTACK=YES indicates that normal stack operations continue until GETLINE reaches a barrier element. GETLINE then passes the caller a return code indicating that a barrier element was reached. The barrier element remains on the stack until your Command Processor explicitly deletes it. SUBSTACK=NO is the default value and indicates that the barrier feature is not used.

Note: If your Command Processor issues GETLINE without SUBSTACK=YES, and a barrier element exists on the input stack, normal stack operations continue until GETLINE reaches a barrier element. In foreground mode, GETLINE then treats the barrier element as a terminal element. In background mode, GETLINE passes an end-of-data return code to the caller. Processing continues in this manner until the caller explicitly deletes the barrier element.

Sources of Input

The GETLINE service routine obtains a line of input from either:

- The terminal or REXX data stack
- The input source described by the topmost element of the input stack

A Command Processor can determine the source of input with which GETLINE will satisfy an input request according to the following procedure:

1. If you specify GETLINE INPUT=TERM, the input is either from the REXX data stack (if elements exist on the REXX data stack), or the terminal. To determine if elements exist on the REXX data stack, use step 3..
2. Before you specify GETLINE INPUT=ISTACK, first invoke the STACK macro with the INQUIRE=TYPE operand to determine the type of element on the top of the input stack.
   a. If the top element of the input stack is an in-storage list (for example, a command procedure), the source indicated by the in-storage list is the current source of input.
   b. If the top element of the input stack is a barrier element that is *not* a NONEST barrier element (indicated by a decimal return code of 44 from STACK), the end of the substack has been reached. GETLINE returns a return code or considers the barrier a terminal element, depending on what was specified on the SUBSTACK operand. For more information on the SUBSTACK operand, see "The Execute Form of the GETLINE macro instruction" on page 199.
   c. If the top element of the input stack is a NONEST barrier element (indicated by a decimal return code 80 from STACK) and if there are elements on the REXX data stack, the current source of input is the REXX data stack. Otherwise, the NONEST barrier acts as a BARRIER=* element as described in step 2b. To determine if elements exist on the REXX data stack, use step 3.
   d. If the top element of the input stack is a terminal element, the input is either from the REXX data stack (if elements exist on the REXX data stack), or the terminal. To determine if elements exist on the REXX data stack, use step 3.
3. To determine if elements exist on the REXX data stack, invoke the REXX data stack replaceable routine, IRXSTK, with the QUEUED function. If the number of queued elements is greater than zero, elements exist on the REXX data stack.

   Note: If the current source of input might be the REXX data stack, and if the Command Processor is invoked by a CLIST and a CLIST DATA-ENDDATA group exists, input is from the CLIST DATA-ENDDATA group.

The REXX Data Stack

A Command Processor invoked by a REXX exec can receive input through the REXX data stack using GETLINE. GETLINE selects the source of input depending on:

- The value of the INPUT parameter, stated explicitly or by default
- Whether elements are present on the data stack
- The state of the input stack when the Command Processor invokes GETLINE.

When you specify GETLINE INPUT=ISTACK, either explicitly or by default, GETLINE obtains input from the REXX data stack first, if there are elements on the REXX data stack and if the topmost element on the input stack is either a terminal element or a NONEST-type barrier element. A NONEST-type barrier element is indicated by a return code of decimal 80 from the STACK service routine. When GETLINE has processed all lines of input on the data stack, it then obtains input from the terminal.

When the topmost element on the input stack is an in-storage list element (including a command procedure), GETLINE obtains input from the source indicated by the in-storage list element. This ensures compatibility with applications that are not sensitive to the REXX data stack (for example, a CLIST invoked from within a REXX exec).

When you specify GETLINE INPUT=TERM, GETLINE obtains input from the REXX data stack first if there are elements on the REXX data stack. If there are no elements on the REXX data stack, GETLINE returns input from a CLIST DATA-ENDDATA group, if present, or from the terminal.

The Input Stack

There are two sources of input: the terminal and an in-storage list (including a command procedure).

Terminal

GETLINE obtains input from the terminal under either of the following conditions:

- You specify GETLINE with the TERM operand and the GETLINE service routine determined that there are no elements on the REXX data stack or the REXX data stack is not available for the current environment.
- You specify GETLINE with the ISTACK operand and the current source of input is either a terminal element or a NONEST-type barrier element, and the current data stack is either empty or not available. STACK indicates a NONEST-type barrier element with a return code of decimal 80.

When GETLINE obtains input from the terminal, you can process the input either as a logical line by including the LOGICAL operand or as a physical line by including the PHYSICAL operand. LOGICAL is the default value.

- **Physical Line Processing:** A physical line is a line that is returned to the requesting program exactly as it is received from the input source. The contents of the line are not inspected by the GETLINE service routine.
- **Logical Line Processing:** A logical line is a line that has undergone additional processing by the GETLINE service routine before it is returned to the requesting program. If logical line processing is requested, each line returned to the routine that issued the GETLINE is inspected to see if the last character of the line is a continuation mark (a dash '-' or a plus '+'). A continuation mark signals GETLINE to get another line from the terminal and to concatenate that line with the line previously obtained. The continuation mark is overlaid with the first character of the new line. However, when ASIS is specified, GETLINE does not recognize line continuation.

In-Storage List

If the top element of the input stack is an in-storage list, and you do not specify TERM in the GETLINE macro instruction, the line will be obtained from the in-storage list. The in-storage list is a resident data set that has been previously made available to the I/O service routines with the STACK service routine.

The STACK service routine saves the addressing mode of the program that invoked it. Address values will be treated as 24-bit or 31-bit addressing mode depending on the original issuer of STACK for that element.

No logical line processing is performed on the lines because it is assumed that each line in the in-storage list is a logical line. It is also assumed that no single record has a length greater than 256 bytes.

End of Data Processing

If you issue a GETLINE macro against an in-storage list from which all the records have already been read, GETLINE senses an end of data (EOD) condition. GETLINE deletes the top element from the input stack and passes a return code of 16 in register 15. Return code 16 indicates that no line of input has been returned by the GETLINE service routine. You can use this EOD code (16) as an indication that all input from a particular source has been exhausted and no more GETLINE macro instructions should be issued against this input source.

If you reissue a GETLINE macro instruction against the input stack after a return code of 16, a record will be returned from the next input source indicated by the input stack. You can identify the source of this record by the return code (0 = terminal, 4 = in-storage). See “Return Codes from GETLINE” on page 206 for a list of the return codes.

Building the GETLINE Parameter Block

When the list form of the GETLINE macro instruction expands, it builds a two word GETLINE parameter block (GTPB). The list form of the macro instruction initializes this GTPB according to the operands you have coded in the macro instruction. This initialized block, which you can later modify with the execute form of the macro instruction, indicates to the GETLINE service routine the function you want performed.

You must supply the address of the GTPB to the execute form of the GETLINE macro instruction. For non-reentrant programs you can do this simply by placing a symbolic name in the symbol field of the list form of the macro instruction, and passing this symbolic name to the execute form of the macro instruction as the PARM value. The GETLINE parameter block is defined by the IKJGTPB DSECT, which is provided in SYS1.MACLIB. Table 60 on page 204 describes the contents of the GTPB.

Table 60. The GETLINE parameter block

Number of bytes	Field name	Contents or meaning
2		Control flags. These bits describe the requested input line to the GETLINE service routine. Byte 1: **..0. ....** The input line is a logical line. **..1. ....** The input line is a physical line. **...0 ....** The input line is to be obtained from the current input source indicated by the input stack. **...1 ....** The input line is to be obtained from the terminal. **xx.. xxxx** Reserved bits. Byte 2: **1... ....** SUBSTACK=YES is specified. **.xxx xxxx** Reserved.

Table 60. The GETLINE parameter block (continued)

Number of bytes	Field name	Contents or meaning
2		GET options field. These bits indicate to the GETLINE service routine which of the options you want to use for GET. Byte 1: **1... ....** Always set to 1. **...0 ....** WAIT processing has been requested. Control will be returned to the issuer of GETLINE only after an input message has been read. **...1 ....** NOWAIT processing has been requested. Control will be returned to the issuer of the GETLINE macro instruction whether a line of input is available. **.... ..00** EDIT processing has been requested. In addition to the editing provided by ASIS processing, the input buffer is to be filled out with trailing blanks to the next doubleword boundary. **.... ..01** ASIS processing has been requested. (See the ASIS operand of the GETLINE macro instruction description.) **.xx. xx..** Reserved bits. Byte 2: **xxxx xxxx** Reserved.
4	GTPBIBUF	The address of the input buffer. The GETLINE service routine fills this field with the address of the input buffer in which the input line has been placed.

Input Line Format - The Input Buffer

The second word of the GETLINE parameter block contains zeros until the GETLINE service routine returns a line of input. The service routine places the requested input line into an input buffer beginning on a doubleword boundary located in subpool 1. It then places the address of this input buffer into the second word of the GTPB.

Note: The application that invoked GETLINE should release the input buffer's storage to prevent the accumulation of unused storage. The application can free the storage with the FREEMAIN macro instruction after the application has processed or copied an input line.

For commands not running on a command invocation platform:

- Input buffer storage returned by GETLINE is automatically freed when the Command Processor relinquishes control.
- The application should free the input buffer's storage after it uses the storage. This prevents unused storage from accumulating while the application is running.

For commands running on a command invocation platform:

- Input buffer storage returned by GETLINE is not freed when the Command Processor relinquishes control.
- It is important to free the input buffer's storage after use to prevent the unused storage from accumulating during a TSO/E session.
- The storage cannot be freed after the application ends because the storage addresses are not known to new applications.

For more information on commands that are eligible to execute on a command invocation platform, see *z/OS TSO/E Customization*.

Regardless of the source of input, an in-storage list or the terminal, the input line returned to the Command Processor by the GETLINE service routine is in a standard format. All input lines are in a variable-length record format with a fullword header followed by the text returned by GETLINE. Figure 90 on page 206 shows the format of the input buffer returned by the GETLINE service routine.

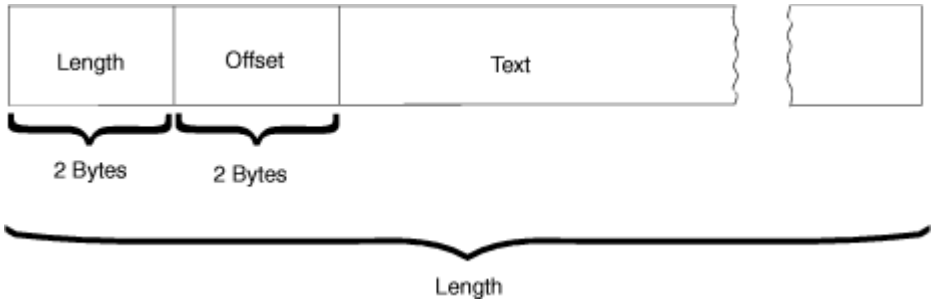


Figure 90. Format of the GETLINE Input Buffer

The two-byte length field contains the length of the input line including the header length (4 bytes). You can use the length field to determine the length of the input line to be processed, and later, to free the input buffer with the R-form of the FREEMAIN macro instruction.

The two-byte offset field is always set to zero on return from the GETLINE service routine.

Figure 91 on page 207 shows the GETLINE control block structure after the GETLINE service routine has returned an input line.

Return Codes from GETLINE

When the GETLINE service routine returns to the program that invoked returns one of the following codes in general register 15:

Table 61. Return codes from the GETLINE service routine

Return code dec(Hex)	Meaning
0(0)	GETLINE has completed successfully. The line was obtained from either the REXX data stack, a command procedure DATA-ENDDATA group, or the terminal.
4(4)	GETLINE has completed successfully. The line was obtained from an in-storage list or command procedure.
8(8)	The GETLINE function was not completed. An attention interruption occurred during GETLINE processing, and the user's attention routine turned on the completion bit in the communications ECB.
12(C)	The NOWAIT option was specified and no line was obtained.
16(10)	An EOD condition occurred. An attempt was made to get a line from an in-storage list but the list had been exhausted.
20(14)	Incorrect parameters were passed to the GETLINE service routine.
24(18)	GETLINE was unable to obtain sufficient storage to satisfy the request for input buffers.
28(1C)	The terminal has been disconnected.
32(20)	An attempt to obtain a line from a command procedure DATA-ENDDATA group failed.
40(28)	A barrier element is on the top of the stack and SUBSTACK=YES was specified. No command buffer is passed back.

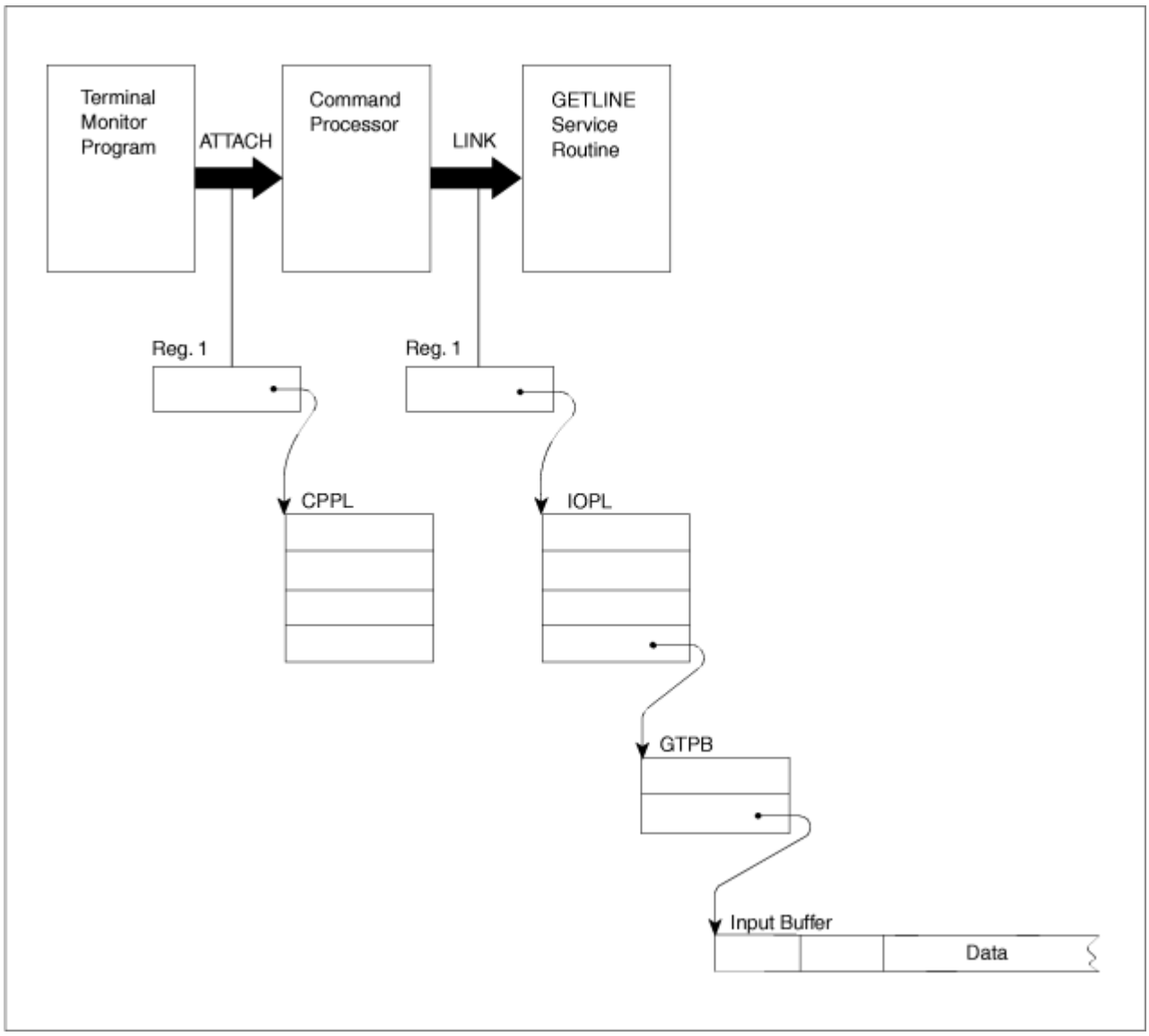


Figure 91. GETLINE Control Blocks - Input Line Returned

Examples Using GETLINE

Figure 92 on page 208 is an example of the code required to execute the GETLINE macro instruction. In this example two execute forms of the GETLINE macro instruction are issued. The first one builds the IOPL, and uses the parameters initialized by the list form of the macro instruction to get a physical line from the terminal with the NOWAIT and ASIS options.

In the second execution of the GETLINE macro instruction, the same IOPL is used, but the GETLINE options are changed explicitly from TERM to ISTACK and from NOWAIT to WAIT, and by default from PHYSICAL to LOGICAL and from ASIS to EDIT.

Notice also that the IKJCPPL DSECT is used to map the Command Processor parameter list, and the IKJGTPB DSECT is used to map the GETLINE parameter block.

```
* ON ENTRY FROM THE TMP, REGISTER 1 CONTAINS A POINTER TO THE COMMAND
* PROCESSOR PARAMETER LIST.
*
*      SET UP ADDRESSABILITY
*      SAVE AREA CHAINING
*
         LR    2,1                   SAVE THE ADDRESS OF THE CPPL.
         USING CPPL,2                ADDRESSABILITY FOR THE CPPL
*
* ISSUE AN EXECUTE FORM OF THE GETLINE MACRO INSTRUCTION TO GET A
* PHYSICAL LINE FROM THE TERMINAL.  THIS EXECUTE FORM BUILDS AND
* INITIALIZES THE INPUT/OUTPUT PARAMETER LIST.
*
         L     3,CPPLUPT             PLACE THE ADDRESS OF THE UPT
*                                    INTO A REGISTER.
         L     4,CPPLECT             PLACE THE ADDRESS OF THE ECT
*                                    INTO A REGISTER.
         GETLINE    PARM=GETBLOCK,UPT=(3),ECT=(4),ECB=ECBADS,          X
               MF=(E,IOPLADS)
*
* THIS EXECUTE FORM OF THE GETLINE MACRO INSTRUCTION USES THE TERM,
* PHYSICAL, ASIS, AND NOWAIT OPERANDS CODED IN THE LIST FORM OF
* THE GETLINE MACRO INSTRUCTION.
*
* GET THE ADDRESS OF THE RETURNED LINE FROM THE GETLINE PARAMETER
* BLOCK.
*
         LA    6,GETBLOCK            SET UP ADDRESSABILITY FOR
         USING GTPB,6                THE GTPB.
         L     5,GTPBIBUF            GET THE ADDRESS OF THE LINE.
*
*        PROCESS THE LINE
*
* ISSUE ANOTHER EXECUTE FORM OF THE GETLINE MACRO INSTRUCTION.
* THIS ONE GETS A LINE FROM THE CURRENTLY ACTIVE INPUT SOURCE - IT
* USES THE IOPL CONSTRUCTED BY THE FIRST EXECUTION OF THE GETLINE
* MACRO INSTRUCTION AND MODIFIES THE GTPB CREATED BY THE LIST FORM
* OF THE GETLINE MACRO INSTRUCTION.
```

```
         GETLINE    INPUT=(ISTACK),TERMGET=(WAIT),MF=(E,IOPLADS)
*
* THIS EXECUTE FORM OF THE GETLINE MACRO INSTRUCTION CHANGES TERM
* TO ISTACK, DEFAULTS TO LOGICAL, CHANGES NOWAIT TO WAIT, AND TAKES
* THE DEFAULT VALUE EDIT.
*
*

* BLOCK.
*
         L     5,GTPBIBUF
*
*        PROCESS THE LINE
*        STORAGE DECLARATIONS
IOPLADS  DC    4F'0'                 SPACE FOR THE INPUT/OUTPUT
*                                    PARAMETER LIST
```

Figure 92. Example Showing Two Executions of GETLINE

Figure 'Example Showing Two Executions of GETLINE' **(Continued)**

```
* THE LIST FORM OF THE GETLINE MACRO INSTRUCTION EXPANDS INTO AN
* INITIALIZED GTPB.
*
         GETLINE    INPUT=(TERM,PHYSICAL),TERMGET=(ASIS,NOWAIT),MF=L
ECBADS   DC    F'0'                  SPACE FOR AN EVENT CONTROL BLOCK.
         IKJCPPL                     DSECT FOR THE COMMAND PROCESSOR
*                                    PARAMETER LIST. THIS EXPANDS WITH
*                                    THE SYMBOLIC ADDRESS, CPPL.
         IKJGTPB                     DSECT FOR THE GETLINE PARAMETER
*                                    BLOCK. THIS EXPANDS WITH THE
*                                    SYMBOLIC ADDRESS GTPB.
         END
```

Using PUTLINE to Put a Line Out to the Terminal

Use the PUTLINE macro instruction to prepare a line and write it to the terminal. Use PUTLINE to put out lines that do not require immediate response from the terminal; use PUTGET to put out lines that require immediate response. The types of lines which do not require response from the terminal are defined as data lines and informational message lines.

The PUTLINE service routine prepares a line for output according to the operands you code into the list and execute forms of the PUTLINE macro instruction. The operands of the macro instruction indicate to the PUTLINE service routine the type of line being put out (data line or informational message line), the type of processing to be performed on the line (format only, second level informational message chaining, text insertion), and the options requested.

This topic describes:

- The list and execute forms of the PUTLINE macro instruction
- The PUTLINE parameter block
- The types and formats of output lines
- PUTLINE message processing
- Return codes from PUTLINE

The List Form of the PUTLINE macro instruction

The list form of the PUTLINE macro instruction builds and initializes a PUTLINE parameter block (PTPB), according to the operands you specify in the macro instruction. The PUTLINE parameter block indicates to the PUTLINE service routine which functions you want performed.

In the list form of the macro instruction, only

```
PUTLINE      MF=L
```

is required. The parameters that are coded on the execute form of the macro instruction are used to fill in the PTPB. Any parameters that are not coded on the execute form of the macro instruction are set to their defaults. Figure 93 on page 209 shows the list form of the PUTLINE macro instruction each of the operands is explained following the figure.

```
 [symbol]   PUTLINE      [                            +
    {,SINGLE }                    ]
                         [OUTPUT=(output address {,TERM } {+
,MULTLVL} {,INFOR} {,NOTRANS})]
                         [                          {FORMAT} {+
,MULTLIN} {,DATA } {,TRANS  } ]

                         [              {EDIT   }                                    ]
                         [ ,TERMPUT=(   {ASIS   } {,WAIT  } {+
,NOHOLD} {,NOBREAK})]
                         [              {CONTROL} {,NOWAIT} {+
,HOLD  } {,BREAKIN} ]

                           ,MF=L
```

Figure 93. The List Form of the PUTLINE macro instruction

OUTPUT=*output address*
indicates that an output line is to be written to the terminal. The type of line provided and the processing to be performed on that line by the PUTLINE service routine are described by the OUTPUT sublist operands TERM, FORMAT, SINGLE, MULTI, MULTLIN, INFOR, DATA, NOTRANS, and TRANS. The default values are TERM, SINGLE, INFOR, and NOTRANS.

The output address differs depending upon whether the output line is an informational message or a data line. For DATA requests, it is the address of the beginning (the fullword header) of a data record to be written to the terminal. For informational message requests (INFOR), it is the address of the output line descriptor. The output line descriptor (OLD) describes the message to be put out, and contains the address of the beginning (the fullword header) of the message or messages to be written to the terminal by the PUTLINE service routine.

When a barrier element is the top stack element, and PUTLINE is operating in the foreground, PUTLINE displays the output at the terminal; if PUTLINE is operating in the background, it places the output in the SYSTSOUT data set.

TERM | FORMAT

TERM
Write the line out to the terminal.

FORMAT
The output request is only to format a single message and not to put the message out to the terminal. The PUTLINE service routine returns the address of the formatted line by placing it in the third word of the PUTLINE parameter block.

SINGLE | MULTLVL | MULTLIN

SINGLE
The output line is a single line.

MULTLVL
The output message consists of multiple levels. INFOR must be specified.

MULTLIN
The output data consists of multiple lines. DATA must be specified.

INFOR | DATA

INFOR
The output line is an informational message.

DATA
The output line is a data line.

NOTRANS | TRANS

NOTRANS
specifies that the output line is not to be translated.

TRANS
specifies that the output line is to be written in the language specified in the user profile table (UPT). INFOR must be specified if TRANS is specified.

Note: For more information about providing translated messages, see "PUTLINE Message Line Processing" on page 224.

TERMPUT=
specifies the options requested. The options are EDIT, ASIS, or CONTROL, WAIT or NOWAIT, NOHOLD or HOLD, and NOBREAK or BREAKIN. The default values are EDIT, WAIT, NOHOLD, and NOBREAK.

EDIT | ASIS | CONTROL

EDIT
specifies that in addition to minimal editing (see ASIS), the following functions are requested:

1. Any trailing blanks are removed before the line is written to the terminal. If a blank line is sent, the terminal vertically spaces one line.
2. Control characters are added to the end of the output line to position the cursor to the beginning of the next line.
3. All terminal control characters (for example: bypass, restore, horizontal tab, new line) are replaced with a printable character. Backspace is an exception; see item 4 under ASIS.

ASIS
specifies that minimal editing is to be performed as follows:

1. The line of output is translated from EBCDIC to terminal code. Incorrect characters are converted to a printable character to prevent program-caused I/O errors. This does not mean that all unprintable characters are eliminated. Restore, upper case, lower case, bypass, and bell ring, for example, might be valid but nonprinting characters at some terminals. (See CONTROL.)
2. Transmission control characters are added.
3. EBCDIC NL, placed at the end of the message, indicates that the cursor is to be returned at the end of the line. NL is replaced with whatever is necessary for that particular terminal type to cause the cursor to return. This NL processing occurs only if you specify ASIS, and the NL is the last character in your message.

   If you specify EDIT, NL is handled as described by item 3 under EDIT.

   If the NL is embedded in your message, a semicolon or colon may be substituted for NL and sent to the terminal. No idle characters are added (see item 6 below). This can cause overprinting, particularly on terminals that require a line-feed character to position the carrier on a new line.
4. If you have used backspace in your output message, but the backspace character does not exist on the terminal type to which the message is being routed, the PUTLINE service routine attempts alternate methods to accomplish the backspace.
5. Control characters are added as needed to cause the message to occur on several lines if the output line is longer than the terminal line size.
6. Idle characters are sent at the end of each line to prevent typing as the carrier returns.

CONTROL
specifies that the output line is composed of terminal control characters and will not print or move the carrier on the terminal. This option should be used for transmission of characters such as bypass, restore, or bell ring.

WAIT | NOWAIT

WAIT
specifies that control will not be returned until the output line has been placed into a terminal output buffer.

NOWAIT
specifies that control should be returned whether a terminal output buffer is available. If no buffer is available, a return code of 8 (decimal) will be returned in register 15 to the Command Processor.

NOHOLD | HOLD

NOHOLD
specifies that control is to be returned to the routine that issued the PUTLINE macro instruction, and that the routine can continue processing as soon as the output line has been placed on the output queue.

HOLD
specifies that the routine that issued the PUTLINE macro instruction cannot continue its processing until this output line has been put out to the terminal or deleted.

NOBREAK | BREAKIN

NOBREAK
specifies that if the terminal user has started to enter input, the user is not to be interrupted. The output message is placed on the output queue to be printed after the terminal user has completed the line.

BREAKIN
specifies that output has precedence over input. If the user at the terminal is transmitting, transmission is interrupted, and this output line is sent. Any data that was received before the interruption is kept and displayed at the terminal following this output line.

MF=L
indicates that this is the list form of the macro instruction.

The Execute Form of the PUTLINE macro instruction

Use the execute form of the PUTLINE macro instruction to put a line or lines out to the terminal, to chain second-level messages, and to format a line and return the address of the formatted line to the code that issued the PUTLINE macro instruction. Use the execute form of the PUTLINE macro instruction to perform the following functions:

- Set up the input/output parameter list (IOPL).
- Initialize those fields of the PUTLINE parameter block (PTPB) not initialized by the list form of the macro instruction, or to modify those fields already initialized.
- Pass control to the PUTLINE service routine.

The operands you specify in the execute form of the PUTLINE macro instruction set up control information used by the PUTLINE service routine. You can use the PARM, UPT, ECT, and ECB operands of the PUTLINE macro instruction to build, complete or modify an IOPL. The OUTPUT and TERMPUT operands and their sublist operands initialize the PUTLINE parameter block. The PUTLINE parameter block is referred to by the PUTLINE service routine to determine which functions you want PUTLINE to perform. The PUTLINE service routine makes use of the IOPL and the PTPB to determine which of the PUTLINE functions you want performed.

In the execute form of the PUTLINE macro instruction only the following is required:

```
PUTLINE      MF=(E,{list address})
                   {    (1)     }
```

The PARM, UPT, ECT, and ECB operands are not required if you have built your IOPL in your own code.

The output line address is required for each issuance of the PUTLINE macro instruction, but you can supply it in the list form of the macro instruction.

The other operands and sublists are optional because you can supply them in the list form of the macro instruction or in a previous execute form, or because you might want to use the default values which are automatically supplied by the macro expansion itself.

Figure 94 on page 213 shows the execute form of the PUTLINE macro instruction; each of the operands is explained following the figure.

```
 [symbol]   PUTLINE      [PARM=parameter address] [,UPT=upt address)
                         [,ECT=ect address ] [,ECB=ecb address]

                         [OUTPUT=(output address {,TERM } {+
,SINGLE } {,INFOR} {,NOTRANS})]
                         [                       {FORMAT} {+
,MULTLVL} {,DATA } {,TRANS  } ]
                         [                                {,MULTLIN}                    ]

                         [             {EDIT   +
}                              ]
                         [ ,TERMPUT=(  {ASIS   } {,WAIT  } {+
,NOHOLD} {,NOBREAK})]
                         [             {CONTROL} {,NOWAIT} {+
,HOLD  } {,BREAKIN} ]

                         [ ,ENTRY={entry address}]+
  ,MF=(E {,list address})
                         [        {    (15)     }]+
         {     (1)     }
```

Figure 94. The Execute Form of the PUTLINE macro instruction

PARM=*parameter address*
specifies the address of the 3-word PUTLINE parameter block (PTPB). It can be the address of a list form of the PUTLINE macro instruction. The address can be any address valid in an RX instruction, or the number of one of the general registers 2–12 enclosed in parentheses. This address will be placed into the IOPL.

UPT=*upt address*
specifies the address of the user profile table (UPT). You can obtain this address from the Command Processor parameter list (CPPL) pointed to by register 1 when a Command Processor is attached by the terminal monitor program. The address can be any address valid in an RX instruction or it can be placed in one of the general registers 2–12 and the register number enclosed in parentheses. This address will be placed into the IOPL.

ECT=*ect address*
specifies the address of the environment control table (ECT). You can obtain this address from the CPPL pointed to by register 1 when a Command Processor is attached by the terminal monitor program. The address can be any address valid in an RX instruction or it can be placed in one of the general registers 2–12 and the register number enclosed in parentheses. This address will be placed into the IOPL.

ECB=*ecb address*
specifies the address of the event control block (ECB). You must provide a one-word event control block and pass its address to the PUTLINE service routine. This address will be placed into the IOPL. The address can be any address valid in an RX instruction or it can be placed in one of the general registers 2–12 and the register number enclosed in parentheses.

OUTPUT=*output address*
indicates that an output line is provided. The type of line provided and the processing to be performed on that line by the PUTLINE service routine are described by the OUTPUT sublist operands TERM, FORMAT, SINGLE MULTLVL, MULTLIN, INFOR, DATA, NOTRANS, and TRANS. The default values are TERM, SINGLE, INFOR, and NOTRANS.

The output address differs depending upon whether the output line is an informational message or a data line. For DATA requests, it is the address of the beginning (the fullword header) of a data record to be put out to the terminal. For informational message requests (INFOR), it is the address of the output line descriptor. The output line descriptor (OLD) describes the message to be put out, and contains the address of the beginning (the fullword header) of the message or messages to be written to the terminal by the PUTLINE service routine.

When a barrier element is the top stack element, and PUTLINE is operating in the foreground, PUTLINE displays the output at the terminal; if PUTLINE is operating in the background, it places the output in the SYSTSOUT data set.

TERM | FORMAT

TERM
Write the line out to the terminal.

FORMAT
The output request is only to format a single message and not to put the messages out to the terminal. The PUTLINE service routine returns the address of the formatted line by placing it in the third word of the PUTLINE parameter block.

SINGLE | MULTLVL | MULTLIN

SINGLE
The output line is a single line.

MULTLVL
The output message consists of multiple levels. INFOR must be specified.

MULTLIN
The output data consists of multiple lines. DATA must be specified.

INFOR | DATA

INFOR
The output line is an informational message.

DATA
The output line is a data line.

NOTRANS | TRANS

NOTRANS
specifies that the output line is not to be translated.

TRANS
specifies that the output line is to be written in the language specified in the user profile table (UPT). INFOR must be specified if TRANS is specified.

Note: For more information about providing translated messages, see "PUTLINE Message Line Processing" on page 224.

TERMPUT=
specifies the options requested. The options are EDIT, ASIS, or CONTROL; WAIT or NOWAIT; NOHOLD or HOLD; and NOBREAK or BREAKIN. The default values are EDIT, WAIT, NOHOLD, and NOBREAK.

EDIT | ASIS | CONTROL

EDIT
specifies that in addition to minimal editing (see ASIS), the following functions are requested:

1. Any trailing blanks are removed before the line is written to the terminal. If a blank line is sent, the terminal vertically spaces one line.
2. Control characters are added to the end of the output line to position the cursor to the beginning of the next line.
3. All terminal control characters (for example: bypass, restore, horizontal tab, new line) are replaced with a printable character. Backspace is an exception; see item 4 under ASIS.

ASIS
specifies that minimal editing is to be performed as follows:

1. The line of output is translated from EBCDIC to terminal code. Incorrect characters are converted to a printable character to prevent program-caused I/O errors. This does not mean that all unprintable characters are eliminated. Restore, uppercase, lowercase, bypass, and bell ring, for example, might be valid but nonprinting characters at some terminals. (See CONTROL.)
2. Transmission control characters are added.

3. EBCDIC NL, placed at the end of the message, indicates that the cursor is to be returned at the end of the line. NL is replaced with whatever is necessary for that particular terminal type to cause the cursor to return. This NL processing occurs only if you specify ASIS, and the NL is the last character in your message.

   If you specify EDIT, NL is handled as described in 3 under EDIT.

   If the NL is embedded in your message, a semicolon or colon may be substituted for NL and sent to the terminal. No idle characters are added (see item 6 below). This can cause overprinting, particularly on terminals that require a line-feed character to position the cursor on a new line.
4. If you have used backspace in your output message, but the backspace character does not exist on the terminal type to which the message is being routed, the PUTLINE service routine attempts alternate methods to accomplish the backspace.
5. Control characters are added as needed to cause the message to occur on several lines if the output line is longer than the terminal line size.
6. Idle characters are sent at the end of each line to prevent typing as the carrier returns.

CONTROL
specifies that the output line is composed of terminal control characters and will not display or move the cursor on the terminal. This option should be used for transmission of characters such as bypass, restore, or bell ring.

WAIT | NOWAIT

WAIT
specifies that control will not be returned until the output line has been placed into a terminal output buffer.

NOWAIT
specifies that control should be returned whether or not a terminal output buffer is available. If no buffer is available, a return code of 8 is returned in register 15.

NOHOLD | HOLD

NOHOLD
specifies that control is returned to the routine that issued the PUTLINE macro instruction, and it can continue processing, as soon as the output line has been placed on the output queue.

HOLD
specifies that the module that issued the PUTLINE macro instruction is not to resume processing until the output line has been put out to the terminal or deleted.

NOBREAK | BREAKIN

NOBREAK
specifies that if the terminal user has started to enter input, the user is not to be interrupted. The output message is placed on the output queue to be displayed after the terminal user has completed the line.

BREAKIN
specifies that output has precedence over input. If the user at the terminal is transmitting, the user is interrupted, and the output line is sent. Any data that was received before the interruption is kept and displayed at the terminal following the output line.

ENTRY=*entry address* | (15)
specifies the entry point of the PUTLINE service routine. If ENTRY is omitted, the PUTLINE macro expansion will generate a LINK macro instruction to invoke the PUTLINE service routine. The address can be any address valid in an RX instruction or (15) if the entry point address has been loaded into general register 15.

MF=E
indicates that this is the execute form of the PUTLINE macro instruction.

***list address* | (1)**
The address of the four-word input/output parameter list (IOPL). This can be a completed IOPL that you have built, or 4 words of declared storage to be filled from the PARM, UPT, ECT, and ECB operands of this execute form of the PUTLINE macro instruction. The address is any address valid in an RX instruction or (1) if the parameter list address has been loaded into general register 1.

Building the PUTLINE Parameter Block

When the list form of the PUTLINE macro instruction expands, it builds a three-word PUTLINE parameter block (PTPB). The list form of the macro instruction initializes the PTPB according to the operands you have coded in the macro instruction. The initialized block, which you can later modify with the execute form of the PUTLINE macro instruction, indicates to the PUTLINE service routine the function you want performed. You must supply the address of the PTPB to the execute form of the PUTLINE macro instruction. Because the list form of the macro instruction expands into a PTPB, all you need do is pass the address of the list form of the macro instruction to the execute form as the PARM value.

The PUTLINE parameter block is defined by the IKJPTPB DSECT, which is provided in SYS1.MACLIB. Table 62 on page 216 describes the contents of the PTPB.

Table 62. The PUTLINE parameter block

Number of bytes	Field name	Contents or meaning
2		Control flags. These bits describe the output line to the PUTLINE service routine. Byte 1: **..0. ....** The output line is a message. **..1. ....** The output line is a data line. **...1 ....** The output line is a single level or a single line. **.... 1...** The output is multiline. **.... .1..** The output is multilevel. **.... ..1.** The output line is an informational message. **xx.x xx.x** Reserved bits. Byte 2: **..1. ....** The format only function was requested. **.... ..1.** The output line is to be written in the language specified in the UPT. **xx.x xx.x** Reserved bits.

Table 62. The PUTLINE parameter block (continued)

Number of bytes	Field name	Contents or meaning
2		PUT options field. These bits indicate to the PUTLINE service routine which of the options you want to use for PUT. Byte 1: **0... ....** Always set to 0. **...0 ....** WAIT processing has been requested. Control will be returned to the issuer of PUTLINE only after the output line has been placed into a terminal output buffer. **...1 ....** NOWAIT processing has been requested. Control will be returned to the issuer of PUTLINE whether or not a terminal output buffer is available. **.... 0...** NOHOLD processing has been requested. The Command Processor that issued the PUTLINE can resume processing as soon as the output line has been placed on the output queue. **.... 1...** HOLD processing has been requested. The Command Processor that issued the PUTLINE is not to resume processing until the output line has been written to the terminal or deleted. **.... .0..** NOBREAK processing has been requested. The output line will be printed only when the terminal user is not entering a line. **.... .1..** BREAKIN processing has been requested. The output line is to be sent to the terminal immediately. If the terminal user is entering a line, the user is to be interrupted. **.... ..00** EDIT processing has been requested. **.... ..01** ASIS processing has been requested. **.... ..10** CONTROL processing has been requested. Byte 2: Reserved.
4	PTPBOPUT	The address of the output line descriptor (OLD) if the output line is a message. The address of the fullword header preceding the data if the output line is a single data line. The address of a forward-chain pointer preceding the fullword data header, if the output is multiline data.
4	PTPBFLN	Address of the format only line. The PUTLINE service routine places the address of the formatted line into this field.

Types and Formats of Output Lines

There are two types of output lines processed by the PUTLINE service routine: data lines and message lines.

Use the OUTPUT sublist operands in the PUTLINE macro instruction to indicate to the PUTLINE service routine which type of line you want processed (DATA, INFOR), whether the output consists of one line, several lines, or several levels of messages (SINGLE, MULTLIN, MULTLVL), whether the output line is to be written in the language specified in the UPT (TRANS, NOTRANS), and whether the line is to be written to the terminal (TERM), or formatted only (FORMAT).

Data Lines

A data line is the simplest type of output processed by the PUTLINE service routine. It is simply a line of text to be written to the terminal. PUTLINE does not format the line or process it in any way; it merely

writes the line, as it appears, out to the terminal. Use the DATA operand on the PUTLINE macro instruction to indicate that the output line is a data line.

There are two kinds of data lines, single line data and multiline data; each is handled differently by the PUTLINE service routine.

- **Single Line Data:** Single line data is one contiguous character string that PUTLINE places out to the terminal as one logical line. If the line of data you provide exceeds the terminal line length, the PUTLINE service routine segments the line and puts it out as several terminal lines. PUTLINE accepts single line data in the format shown in Figure 95 on page 218.

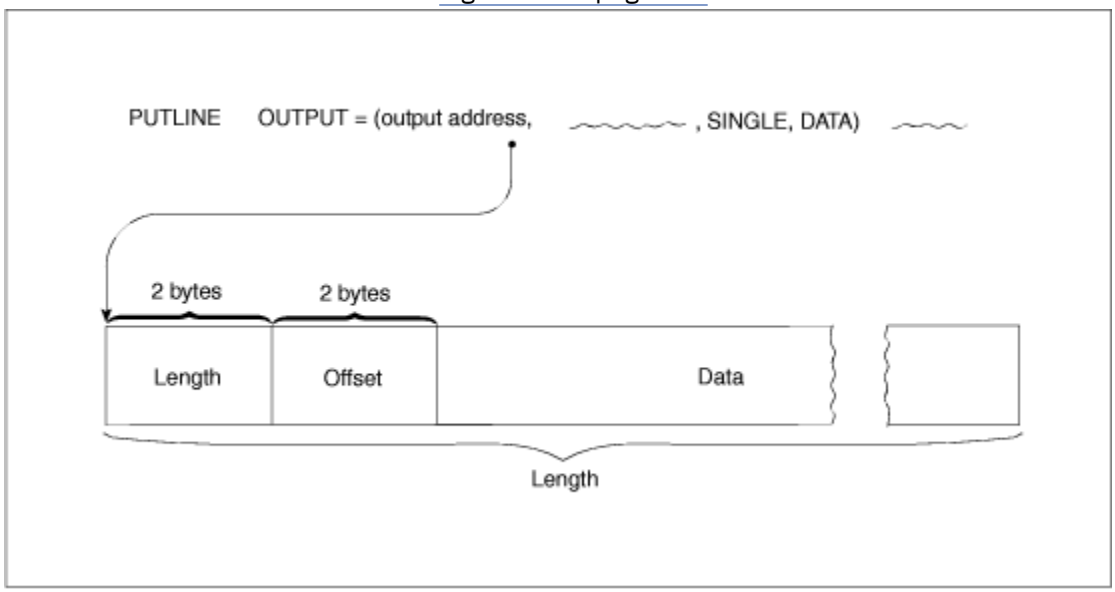


Figure 95. PUTLINE Single Line Data Format

You must precede your line of data with a 4-byte header field. The first two bytes contain the length of the output line, including the header; the second two bytes are reserved for offsets and are set to zero for data lines.

Pass the address of the output line to the PUTLINE service routine by coding the beginning address of the four-byte header as the OUTPUT operand address in either the list or the execute form of the macro instruction. When the macro instruction expands, it places this data line address into the second word of the PUTLINE parameter block.

Figure 97 on page 220 is an example of the code that could be used to write a single line of data to the terminal using the PUTLINE macro instruction. Note that the execute form of the PUTLINE macro instruction is used in this example to construct the input/output parameter list, and that the TERMPUT operands are not coded in either the list or the execute form of the macro instruction; the default values will be assumed by the PUTLINE service routine.

- **Multiline Data:** Multiline data is a chain of single lines. Each line of data is processed by the PUTLINE service routine exactly as if it were single line data. Each element of the chain, however, begins a new line to the terminal. By specifying multiline data (MULTLIN) in the PUTLINE macro instruction, you can put out several variable-length, non-contiguous lines at the terminal with one execution of the macro instruction. PUTLINE accepts multiline data in a format similar to that of single line data except that each line is prefaced with a fullword forward chain pointer. Figure 96 on page 219 shows the format of PUTLINE multiline data.

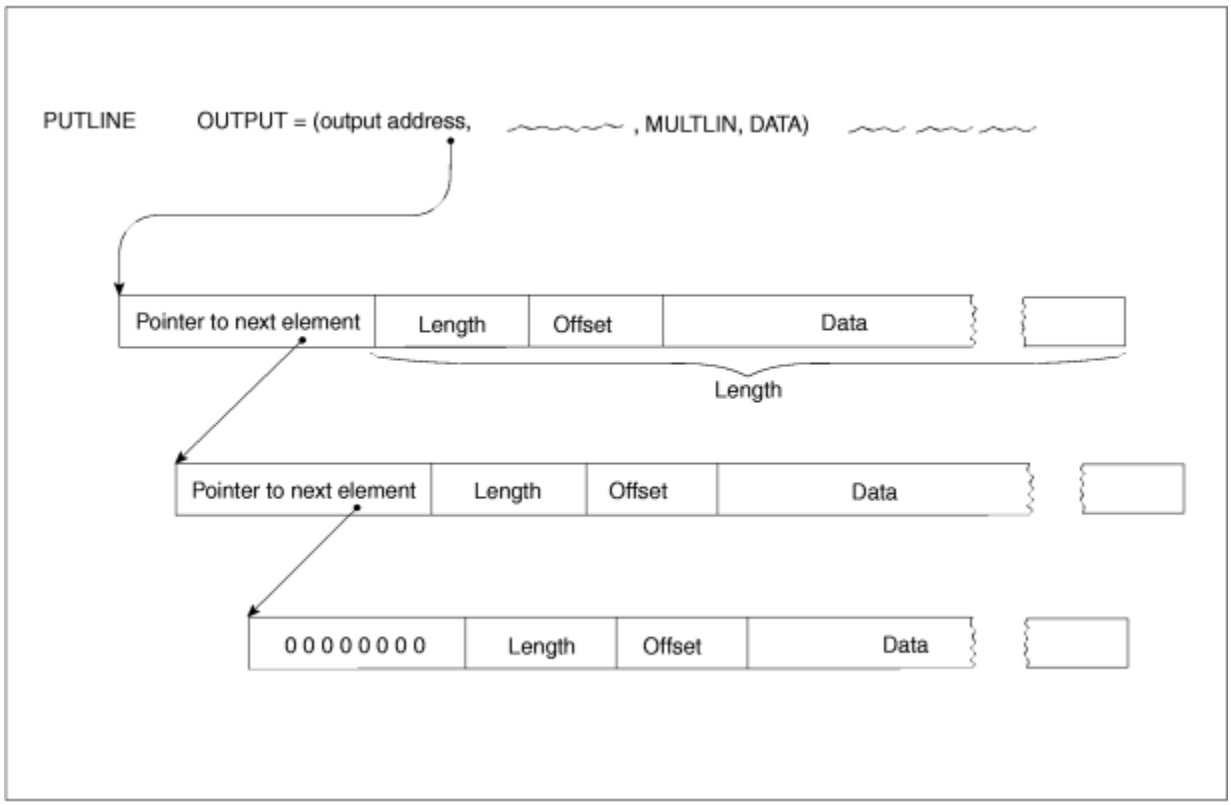


Figure 96. PUTLINE Multiline Data Format

Each of the forward-chain pointers points to the next data line to be written to the terminal. The forward-chain pointer in the last data line contains zeros. In the case of multiline data, you pass the address of the output line to the PUTLINE service routine by coding the beginning address of the first forward-chain pointer as the OUTPUT operand address in either the list or the execute form of the macro instruction. When the macro instruction expands, it places this multiline data address into the second word of the PUTLINE parameter block.

Figure 98 on page 221 is an example of the code required to write multiple lines of data to the terminal using the PUTLINE macro instruction.

Note that the programmer has built his own IOPL rather than build it with the execute form of the PUTLINE macro instruction. Note also the use of the IOPL and CPPL DSECTs (generated by the IKJIOPL and IKJCPPL macro instructions). These provide an easy method of accessing the fields within the IOPL and the CPPL, and they protect your code from changes made to the control blocks.

Message Lines

If you code INFOR in the PUTLINE macro, the PUTLINE service routine writes the information you supply as an informational message and provides additional functions not applicable to data lines. An informational message is a line of output from the program in control to the user at the terminal. It is used solely to pass output to the terminal; no input from the terminal is required after an informational message. For information about the additional functions that PUTLINE provides for message lines, see “PUTLINE Message Line Processing” on page 224.

There are two types of informational messages processed by the PUTLINE service routine: single-level messages and multilevel messages.

- **Single-Level Messages:** A single-level message is composed of one or more message segments to be formatted and written to the terminal with one execution of the PUTLINE macro instruction. Use the SINGLE operand on the PUTLINE macro instruction to indicate that the output line is a single-level message.

- **Multilevel Messages:** Multilevel messages are composed of one or more message segments to be formatted and written to the terminal, and one or more message segments to be formatted and placed on an internal chain in shared subpool 78. This internal chain can either be put out to the terminal or purged by a second execution of the PUTLINE macro instruction. Use the MULTLVL operand on the PUTLINE macro instruction to indicate that a multilevel message is to be written to the terminal.

```
* ON ENTRY FROM THE TMP, REGISTER 1 CONTAINS A POINTER TO THE COMMAND
* PROCESSOR PARAMETER LIST (CPPL).
*
*      SET UP ADDRESSABILITY
*      SAVE AREA CHAINING
*
         LR    2,1                   SAVE THE ADDRESS OF THE CPPL.
         USING CPPL,2                ADDRESSABILITY FOR THE CPPL
         L     3,CPPLUPT             PLACE THE ADDRESS IF THE UPT
*                                    INTO A REGISTER
         L     4,CPPLECT             PLACE THE ADDRESS OF THE ECT
*                                    INTO A REGISTER
* ISSUE THE EXECUTE FORM OF THE PUTLINE MACRO INSTRUCTION.  USE IT
* TO WRITE A SINGLE LINE OF DATA TO THE TERMINAL AND TO BUILD THE
* IOPL.  IT DOES NOT SPECIFY THE TERMPUT OPERANDS, AND THEREFORE
* PUTLINE WILL USE THE DEFAULT VALUES.
*
         PUTLINE    PARM=PUTBLOK,UPT=(3),ECT=(4),ECB=ECBADS,           X
               OUTPUT=(TEXTADS,TERM,SINGLE,DATA),MF=(E,IOPLADS)
*
*        PROCESSING
*        STORAGE DECLARATIONS
*
ECBADS   DS    F'0'                  SPACE FOR THE EVENT CONTROL BLOCK
PUTBLOK  PUTLINE    MF=L             LIST FORM OF THE PUTLINE MACRO
*                                    INSTRUCTION. THIS EXPANDS INTO A
*                                    PUTLINE PARAMETER BLOCK.
TEXTADS  DC    H'20'                 LENGTH OF THE OUTPUT LINE
         DC    H'0'                  RESERVED
         DC    CL16' SINGLELINE DATA'
IOPLADS  DC    4F'0'                 SPACE FOR THE INPUT/OUTPUT
*                                    PARAMETER LIST
         IKJCPPL                     DSECT FOR THE CPPL
         END
```

Figure 97. Example Showing PUTLINE Single Line Data Processing

```
* ON ENTRY FROM THE TMP, REGISTER 1 CONTAINS A POINTER TO THE COMMAND
* PROCESSOR PARAMETER LIST (CPPL).
*
*      SET UP ADDRESSABILITY
*      SAVE AREA CHAINING
*
         LR    2,1                   SAVE THE ADDRESS OF THE CPPL.
         USING CPPL,2                ADDRESSABILITY FOR THE CPPL
         L     3,CPPLUPT             PLACE THE ADDRESS IF THE UPT
*                                    INTO A REGISTER
         L     4,CPPLECT             PLACE THE ADDRESS OF THE ECT
*                                    INTO A REGISTER
         LA    5,ECBADS              PLACE THE ADDRESS OF THE ECB
*                                    INTO A REGISTER
* SET UP ADDRESSABILITY FOR THE INPUT/OUTPUT PARAMETER LIST DSECT.
*
         LA    7,IOPLADS
         USING IOPL,7
* FILL IN THE IOPL EXCEPT FOR THE PTPB ADDRESS
         ST    3,IOPLUPT
         ST    4,IOPLECT
         ST    5,IOPLECB
*
* ISSUE THE EXECUTE FORM OF THE PUTLINE MACRO INSTRUCTION
*
         PUTLINE    PARM=PUTBLOK,OUTPUT=(TEXTADS,MULTLIN,DATA),        X
               MF=(E,IOPLADS)
*
*      PROCESSING
*      STORAGE DECLARATIONS
*
ECBADS  DS    F
IOPLADS DS    4F'0'
TEXTADS DC    A(TEXT2)              FORWARD POINTER TO THE NEXT LINE.
        DC    H'20'                 LENGTH OF THE FIRST LINE.
        DC    H'0'                  RESERVED.
        DC    CL16'MULTILINE DATA 1'

PUTBLOK PUTLINE    MF=L             LIST FORM OF THE PUTLINE MACRO
*                                   INSTRUCTION.
*
TEXT2   DC    A(0)                  END OF CHAIN INDICATOR.
        DC    H'20'                 LENGTH OF THE SECOND LINE.
        DC    H'0'                  RESERVED.
        DC    CL16'MULTILINE DATA 2'
*
        IKJCPPL                     DSECT FOR THE COMMAND PROCESSOR
*                                   PARAMETER LIST.  THIS EXPANDS
*                                   WITH THE SYMBOLIC NAME CPPL.
        IKJIOPL                     DSECT FOR THE INPUT/OUTPUT
*                                   PARAMETER LIST.  THIS EXPANDS
*                                   WITH THE SYMBOLIC NAME IOPL.
        END
```

Figure 98. Example Showing PUTLINE Multiline Data Processing

Passing the Message Lines to PUTLINE

You must build each of the message segments to be processed by the PUTLINE service routine as if it were a line of single line data. The segment must be preceded by a four-byte header field, where the first two bytes contain the length of the segment, including the header, and the second two bytes contain an offset value. See “Offset Values” on page 227 for a discussion of offset values. This message line format is required whether the message is a single-level message or a multilevel message.

Because of the additional operations performed on message lines, however, you must provide the PUTLINE service routine with a description of the line or lines that are to be processed. This is done with an output line descriptor (OLD).

There are two types of output line descriptors, depending on whether the messages are single level or multilevel.

The OLD required for a single-level message is a variable-length control block which begins with a fullword value representing the number of segments in the message, followed by fullword pointers to each of the segments.

The format of the OLD for multilevel messages varies from that required for single-level messages in only one respect. You must preface the OLD with a fullword forward-chain pointer. This chain pointer points to another output line descriptor or contains zero to indicate that it is the last OLD on the chain. Table 63 on page 222 shows the format of the output line descriptor.

Table 63. The output line descriptor (OLD)

Number of bytes	Field name	Contents or meaning
4	none	The address of the next OLD, or zero if this is the last one on the chain. This field is present only if the message pointed to is a multilevel message.
4	none	The number of message segments pointed to by this OLD.
4	none	The address of the first message segment.
4	none	The address of the next message segment.

You must build the output line descriptor and pass its address to the PUTLINE service routine as the OUTPUT operand address in either the list or the execute form of the macro instruction. When the macro expands, it places the address of the output line descriptor into the second word of the PUTLINE parameter block.

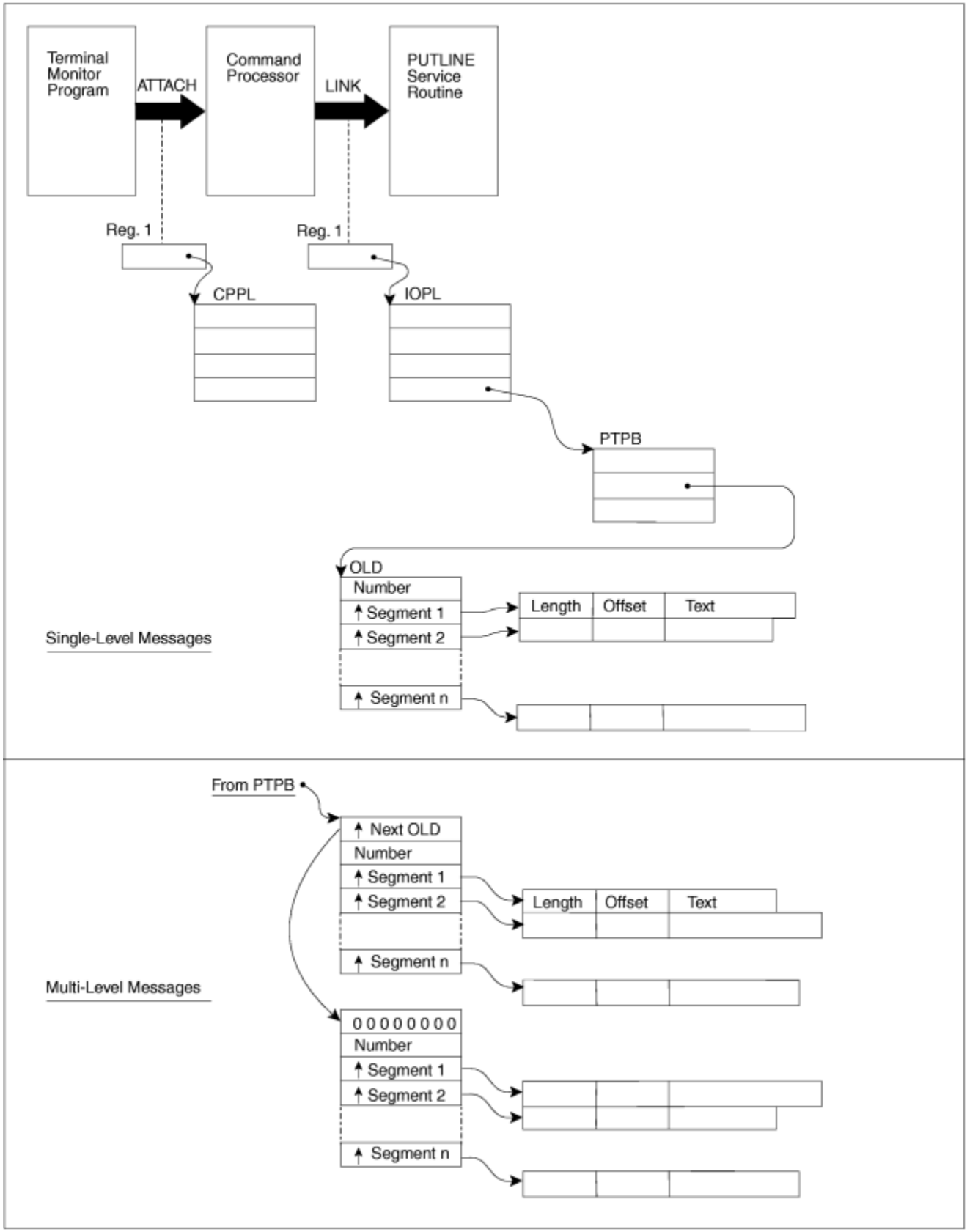


Figure 99. Control Block Structures for PUTLINE Messages

PUTLINE Message Line Processing

In addition to writing a message out to the terminal, the PUTLINE service routine provides the following additional functions for message line processing when the INFOR operand is specified:

- Message identification stripping
- Text insertion
- Formatting only
- Second level informational chaining
- Display of translated text

Table 64 on page 224 shows the output message types for which these PUTLINE service routine functions can be used.

Table 64. PUTLINE Functions and Message Types

	Message types	
	Single level	**Multilevel**
Message ID Stripping	x	x
Text Insertion	x	x
Formatting Only	x	
Second Level Informational Chaining		x
Display of Translated Text	x	x

Stripping Message Identifiers

The user can indicate whether message identifiers should be displayed at the terminal by using the TSO/E PROFILE command. See *z/OS TSO/E Command Reference* and *z/OS TSO/E User's Guide* for a description of the PROFILE command. If the terminal user indicates no message identifiers are to be displayed, the PUTLINE service routine strips them off the message before writing the message to the terminal.

A message identifier must be a variable-length character string, containing no leading or embedded blanks, must not exceed a maximum length of 255 characters, and must be terminated by a blank.

Messages without message identifiers must begin with a blank. A message beginning with a blank is handled by the PUTLINE service routine as a message that does not require message identifier stripping, regardless of what the user at the terminal has requested. If you do not provide a message identifier, and do not begin your message with a blank, the beginning of your message up to the first blank will be stripped off by the PUTLINE service routine if message identifier stripping is requested from the terminal. If the message segment does not contain at least one blank, PUTLINE will return a code of 12, which indicates incorrect parameters, in register 15, even if message ID stripping is not requested from the terminal.

The following examples show the effects of the PUTLINE message identifier stripping function.

If you provide message identifiers on your messages and the terminal user does not request message ID stripping, your message will appear at the terminal exactly as it appears here:

```
MESSAGE0010 THIS IS A MESSAGE.
```

message will appear as:

```
THIS IS A MESSAGE.
```

If you do not want to use message identifiers on your output messages, begin your message with a blank. A message beginning with a blank is unaffected by a terminal user's request for message ID stripping and will appear as you wrote it, minus the blank.

Using the PUTLINE Text Insertion Function

The text insertion function of the PUTLINE service routine allows you to build or modify messages at the time you put them out to the terminal. With text insertion you can respond to different output message requirements with one basic message (the primary segment). You can insert text into this primary segment or add text to it, and thereby build an output message to meet the current processing situation.

To use text insertion, pass your messages to the PUTLINE service routine as a variable number of text segments; from 1 to 255 segments are permissible.

Figure 100 on page 226 shows an example of using the PUTLINE text insertion facility to insert text before the primary segment.

```
TINS0    CSECT  ,
TINS0    AMODE  31
TINS0    RMODE  ANY
@MAINENT DS     0H
         STM    R14,R12,12(R13)       ENTRY LINKAGE
         LR     R12,R15
@PSTART  EQU    TINS0
         USING  @PSTART,R12
         ST     R13,SAVEAREA+4
         LA     R11,SAVEAREA
         ST     R11,8(,R13)
         LA     R13,SAVEAREA
*
*
MAIN     DS     0H
*
         LR     2,1                   Save the address of the CPPL
         USING  CPPL,2                Addressability for the CPPL
         L      3,CPPLUPT             R3<-address of the UPT
         L      4,CPPLECT             R4<-address of the ECT
*
*       Issue the execute form of the PUTLINE macro instruction.
*       PUTLINE builds the IOPL, provides text insertion and writes
*       a message to the terminal,
*
         PUTLINE PARM=PUTBLK,UPT=(3),ECT=(4),ECB=ECBADS,                +
               OUTPUT=(ONEOLD,TERM,SINGLE,INFOR),MF=(E,IOPLADS)
```

```
*
         DS     0H
         L      R13,4(,R13)           EXIT  LINKAGE
         LM     R14,R12,12(R13)
         SLR    R15,R15
         BR     R14
*
```

```
*       Storage declarations follow
*
ECBADS   DC      F'0'           Space for the event control block
IOPLADS  DC     4F'0'           Space for the I/O parameter block
*
PUTBLK   PUTLINE  MF=L          List form of PUTLINE macro.  It
*                               expands into space for the PTPB.
*
ONEOLD   DC      F'4'           Indicate four text segments.
         DC      A(SEG1)        Address of 1st segment
         DC      A(SEG2)        Address of 2nd segment
         DC      A(SEG3)        Address of 3rd segment
         DC      A(SEG4)        Address of 4th segment
SEG1     DC      H'5'           Length of 1st segment
         DC      H'0'           Offset of prime is always zero
         DC      CL1'.'
SEG2     DC      H'13'          Length of 2nd segment
         DC      H'0'           Offset zero - place before 1st segment
         DC      CL9'Segments '
SEG3     DC      H'12'          Length of 3rd segment
         DC      H'0'           Offset zero - place before 1st segment
         DC      CL8'showing '
SEG4     DC      H'45'          Length of 4th segment
         DC      H'0'           Offset zero - place before 1st segment
         DC      CL41'text insertion before the primary segment'
*
*
```

Figure 100. Example of PUTLINE Text Insertion - Before the Primary Segment

Figure of 'Example of PUTLINE Text Insertion - Before the Primary Segment' (**Continued**)

```
*
SAVEAREA DS   18F
*
R15      EQU  15
R14      EQU  14
R13      EQU  13
R12      EQU  12
R11      EQU  11
*
         IKJCPPL
         END
```

Each segment can contain from 0 to 32763 characters, as long as the total number of characters in all the segments does not exceed 32763. You must precede each of these text segments with a four-byte header in which the first two bytes contain the length of the message, including the header, and the second two bytes contain an offset value.

Offset Values

The offset value in the primary segment must be zero. The offset in any secondary segments can be from zero to the length of the primary segment's text field. An offset of zero in a secondary segment implies that the segment is to be placed before the primary segment. An offset that is equal to the length of the primary segment's text field implies that the secondary segment is to be placed after the primary segment. An offset of n, where n represents a value greater than zero but less than the total length of the primary segment, implies that the segment is to be inserted after the nth byte of the primary segment. PUTLINE places the secondary segment after that character, completes the message, and puts it out to the terminal.

If you specify an offset in a secondary segment, greater than the length of the primary segment, PUTLINE cannot handle the request and returns an error code of 12, which indicates incorrect parameters, in register 15. In addition, if the secondary segments do not appear in the OLD with their offsets in ascending order, PUTLINE returns an error code of 12 in register 15.

If you provide more than one secondary segment to be inserted into a primary segment, the offset fields on each of the secondary segments must indicate the position within the original primary segment at which you want them to appear. PUTLINE determines the points of insertion by counting the characters of the original primary segment only. As an example, if you provided one primary and two secondary segments as shown:

2 bytes	2 bytes	28 bytes
32	0	PLEASE ENTER TO PROCESSING
9	13	TEXT
13	16	CONTINUE

PUTLINE would place the first insert, TEXT, after the 13th character, and the second insert, CONTINUE, after the 16th character of the text field of the primary segment. After PUTLINE inserts the two text segments, the message would read:

```
PLEASE ENTER TEXT TO CONTINUE PROCESSING
```

The leading and trailing blanks are automatically stripped off before the message is written to the terminal.

Figure 101 on page 230 is an example of the code required to make use of the text insertion feature of the PUTLINE service routine; it uses the text segments shown above.

Note that the operand INFOR, which indicates to the PUTLINE service routine that the text segments are to be processed as informational messages, requires an output line descriptor to point to the message

segments. Only one output line descriptor (ONEOLD) is required to point to the 3 message segments because the 3 segments are to be combined into one single-level message.

Using the Format Only Function

You can also use the PUTLINE service routine to format a message but not write it at the terminal. To do this, code the FORMAT operand in the PUTLINE macro instruction and pass PUTLINE the same message segment structure required for the text insertion function. The PUTLINE service routine performs text insertion if requested and places the finished message in subpool 1, which is not shared. It then places the address of the formatted line into the third word of the PUTLINE parameter block. The storage occupied by the formatted message belongs to your program and, if space is a consideration, must be freed by it. The returned formatted line is in the variable-length record format; that is, it is preceded by a four-byte header. You can use the first two bytes of this header to determine the length of the returned message, and later, to free the real storage occupied by the message with the R form of the FREEMAIN macro instruction.

One difference between format only processing and text insertion processing is that format only processing can be used only on single-level messages. You cannot use the format only feature to format multilevel messages. You can however, use the second level informational chaining function of PUTLINE to format second-level messages and place them on an internal chain.

If you specify the TRANS operand with the FORMAT operand, the PUTLINE service routine places the finished message in shared subpool 78. The format of the message buffer returned is multiline data format, even if translation fails. Multiline data format is necessary because the translated message text may consist of more lines than the original message text. Figure 96 on page 219 shows the format of multiline data.

Building a Second Level Informational Chain

PUTLINE can accept two levels of informational messages at each execution of the service routine. It formats the first-level message and puts it out to the terminal. The second-level message is formatted and a copy of it is placed on an internal chain in shared subpool 78. This internal chain, the second level informational chain, is maintained by the I/O service routines for the duration of one command or subCommand Processor. You can use the PUTLINE service routine to purge this chain or to put it out to the terminal in its entirety.

To purge the chain without putting it out to the terminal, you must turn on the high-order bit in the first byte (ECTMSGF) of the third word of the environment control table (ECT). The ECT is pointed to by the second word of the input/output parameter list, and can be mapped by the IKJECT DSECT. The next time any chaining or unchaining is requested with PUTLINE or PUTGET, the second-level informational chain will be eliminated.

To put the entire chain out to the terminal, use the PUTLINE macro instruction and place a zero address where the output line address is normally required. This will cause the PUTLINE service routine to write the chain to the terminal and eliminate the internal chain. You will normally use this procedure only if your attention exit routine is using the PUTLINE macro instruction to process a question mark entered from the terminal.

Figure 102 on page 231 is an example of the code required to build a second-level informational chain. It executes the PUTLINE service routine by using two different execute form macro instructions to modify the PUTLINE parameter block built by the list form of the PUTLINE macro instruction.

The code shown puts two messages out to the terminal and places two second-level messages on an internal chain. It then executes a third execute form of the PUTLINE macro instruction with a zero OUTPUT address to put the second level chain out to the terminal.

Note that the offset value for the primary message segment must always be zero, and when placing second-level messages on an internal chain, the offset value for the second-level message must also be zero. Note also that you do not place a message identifier on a second-level message.

Displaying Translated Message Text

You can specify that the message text should be displayed in the language specified in the user profile table (UPT). This is done by using the TRANS operand on the PUTLINE macro instruction.

Return Codes from PUTLINE

When the PUTLINE service routine returns control to the program that invoked it, PUTLINE provides one of the following return codes in general register 15:

Table 65. Return codes from the PUTLINE service routine

Return code dec(Hex)	Meaning
0(0)	PUTLINE completed normally.
4(4)	The PUTLINE service routine did not complete. An attention interruption occurred during its execution, and the attention handler turned on the completion bit in the communications ECB.
8(8)	The NOWAIT option was specified and the line was not written to the terminal.
12(C)	Incorrect parameters were supplied to the PUTLINE service routine.
16(10)	PUTLINE was unable to obtain sufficient storage to satisfy the request for output buffers.
20(14)	The terminal has been disconnected.

Note: See Chapter 21, “Analyzing error conditions with GNRLFAIL/VSAMFAIL,” on page 357 for information on how to issue meaningful error messages for PUTLINE error codes.

Figure 101 on page 230 shows an example of PUTLINE text insertion.

```
* ON ENTRY FROM THE TMP, REGISTER 1 CONTAINS A POINTER TO THE COMMAND
* PROCESSOR PARAMETER LIST (CPPL).
*
*      SET UP ADDRESSABILITY
*      SAVE AREA CHAINING
*
         LR    2,1                   SAVE THE ADDRESS OF THE CPPL.
         USING CPPL,2                ADDRESSABILITY FOR THE CPPL.
         L     3,CPPLUPT             PLACE THE ADDRESS OF THE UPT
*                                    INTO A REGISTER.
         L     4,CPPLECT             PLACE THE ADDRESS OF THE ECT
*                                    INTO A REGISTER.
* ISSUE THE EXECUTE FORM OF THE PUTLINE MACRO INSTRUCTION.  LET IT
* INITIALIZE THE IOPL.
         PUTLINE    PARM=PUTBLK,UPT=(3),ECT=(4),ECB=ECBADS,            X
               OUTPUT=(ONEOLD,TERM,SINGLE,INFOR),MF=(E,IOPLADS)
*
*      PROCESSING
*      STORAGE DECLARATIONS
*
ECBADS   DC    F'0'                  SPACE FOR THE EVENT CONTROL BLOCK
IOPLADS  DC    4F'0'                 SPACE FOR THE INPUT/OUTPUT
*                                    PARAMETER LIST.
PUTBLK   PUTLINE    MF=L             THE LIST FORM OF THE PUTLINE
*                                    MACRO INSTRUCTION.  IT EXPANDS
*                                    INTO SPACE FOR A PTPB.
ONEOLD   DC    F'3'                  INDICATE THREE TEXT SEGMENTS.
         DC    A(FIRSTSEG)           ADDRESS OF THE FIRST TEXT
*                                    SEGMENT.
         DC    A(SECSEG)             ADDRESS OF THE SECOND TEXT
*                                    SEGMENT.
         DC    A(THIRDSEG)           ADDRESS OF THE THIRD TEXT
*                                    SEGMENT.

FIRSTSEG DC    H'32'                 LENGTH OF THE FIRST SEGMENT
*                                    INCLUDING THE HEADER.
         DC    H'0'                  OFFSET OF PRIME SEGMENT IS
*                                    ALWAYS ZERO.
         DC    CL28' PLEASE ENTER TO PROCESSING '
*                                    PRIMARY SEGMENT.
SECSEG   DC    H'9'                  LENGTH OF SECOND SEGMENT
*                                    INCLUDING THE HEADER.
         DC    H'14'                 OFFSET INTO FIRST SEGMENT AFTER
*                                    WHICH SECOND SEGMENT IS TO BE
*                                    INSERTED.
         DC    CL5'TEXT '            TEXT OF THE SECOND SEGMENT
THIRDSEG DC    H'13'                 LENGTH OF THIRD SEGMENT
*                                    INCLUDING THE HEADER.
         DC    H'17'                 OFFSET INTO FIRST SEGMENT AFTER
*                                    WHICH THIRD SEGMENT IS TO BE
*                                    INSERTED.
         DC    CL9'CONTINUE '        TEXT OF THE THIRD SEGMENT
         IKJCPPL                     CPPL DSECT - THIS EXPANDS WITH
*                                    THE SYMBOLIC ADDRESS CPPL.
         END
```

Figure 101. Example Showing PUTLINE Text Insertion

Figure 102 on page 231 shows an example of chaining for PUTLINE second-level information.

```
* ON ENTRY FROM THE TMP, REGISTER 1 CONTAINS A POINTER TO THE COMMAND
* PROCESSOR PARAMETER LIST (CPPL).
*
*      SET UP ADDRESSABILITY
*      SAVE AREA CHAINING
*
         LR    2,1                   SAVE THE ADDRESS OF THE CPPL.
         USING CPPL,2                ADDRESSABILITY FOR THE CPPL.
         L     3,CPPLUPT             PLACE THE ADDRESS IF THE UPT
*                                    INTO A REGISTER.
         L     4,CPPLECT             PLACE THE ADDRESS OF THE ECT
*                                    INTO A REGISTER.
* ISSUE THE EXECUTE FORM OF THE PUTLINE MACRO INSTRUCTION.  THIS ONE
* BUILDS THE IOPL, WRITES A MESSAGE TO THE TERMINAL, AND PLACES ONE
* SECOND-LEVEL MESSAGE ON THE CHAIN.
*
         PUTLINE    PARM PUTBLK,UPT=(3),ECT=(4),ECB=ECBADS,           X
               OUTPUT=(OLD1,TERM,MULTLVL,INFOR),MF=(E,IOPLADS)
*
*     PROCESSING
*
* ISSUE A SECOND EXECUTE FORM OF THE PUTLINE MACRO INSTRUCTION.  IT
* USES THE SAME IOPL AND PTPB AS THE PREVIOUS EXECUTE FORM.  IT GIVES
* A NEW OUTPUT LINE DESCRIPTOR ADDRESS AS THE OUTPUT= OPERAND.  THIS
* EXECUTION OF THE PUTLINE MACRO INSTRUCTION WRITES ONE MESSAGE TO THE
* TERMINAL AND CHAINS ANOTHER.
*
        PUTLINE    PARM=PUTBLK,OUTPUT=(OLD2,MULTLVL,INFOR),           X
             MF=(E,IOPLADS)
*
*     PROCESSING
*
* TO WRITE THE SECOND-LEVEL MESSAGE CHAIN TO THE TERMINAL AND THEN
* PURGE THE CHAIN, ISSUE THE EXECUTE FORM OF THE PUTLINE MACRO
* INSTRUCTION WITH A ZERO ADDRESS WHERE THE OUTPUT LINE ADDRESS IS
* REQUIRED.
*
        PUTLINE    PARM=PUTBLK,OUTPUT=0,MF=(E,IOPLADS)
*
*     PROCESSING
*     STORAGE DECLARATIONS
IOPLADS  DC    4F'0'                 SPACE FOR THE INPUT/OUTPUT
*                                    PARAMETER LIST.
PUTBLK   PUTLINE    MF=L             THE LIST FORM OF THE PUTLINE
*                                    MACRO INSTRUCTION. IT EXPANDS
*                                    INTO SPACE FOR A PTPB.
ECBADS   DC    F'0'                  SPACE FOR THE EVENT CONTROL BLOCK
OLD1     DC    A(NEXTLEN)            FORWARD POINTER TO NEXT OLD
         DC    F'1'                  ONLY ONE SEGMENT.
         DC    A(MESSAGE1)           ADDRESS OF TEXT SEGMENT.
NEXTLEV  DC    A(0)                  INDICATE LAST OLD ON CHAIN
         DC    F'1'                  ONLY ONE SEGMENT.
         DC    A(MESSAGE2)           ADDRESS OF SECOND LEVEL TEXT.
MESSAGE1 DC    H'32'                 LENGTH OF SEGMENT INCLUDING
*                                    HEADER.
         DC    H'0'                  OFFSET OF PRIME SEGMENT MUST BE
*                                    ZERO.
         DC    CL28'MYMSG1 PLEASE ENTER USER ID.'
*                                    FIRST-LEVEL MESSAGE.
```

Figure 102. Example Showing PUTLINE Second-Level Informational Chaining

Figure of 'Example Showing PUTLINE Second-Level Informational Chaining' (**Continued**)

```
MESSAGE2 DC    H'36'                  LENGTH OF SEGMENT INCLUDING
*                                     HEADER.
         DC    H'0'                   OFFSET MUST BE ZERO.
         DC    CL32' USER ID REQUIRED FOR ACCOUNTING'
*                                     SECOND-LEVEL MESSAGE.  NOTE THAT
*                                     IT MUST NOT HAVE A MESSAGE ID.

OLD2     DC    A(NEXTOLD)             FORWARD POINTER TO NEXT OLD.
         DC    F'1'                   ONLY ONE SEGMENT.
         DC    A(SECMSG1)             ADDRESS OF PRIME SEGMENT.
NEXTOLD  DC    A(0)                   INDICATE THIS IS THE LAST OLD
*                                     ON THIS CHAIN.
         DC    F'1'                   ONLY ONE SEGMENT.
         DC    A(SECMSG2)             ADDRESS OF THE SECOND LEVEL TEXT

SECMSG1  DC    H'33'                  LENGTH OF THE TEXT SEGMENT
*                                     INCLUDING THE HEADER.
         DC    H'0'                   OFFSET OF PRIME SEGMENT MUST
*                                     BE ZERO.
         DC    CL29'MYMSG2 PLEASE ENTER PROC NAME'
*                                     FIRST-LEVEL MESSAGE.

SECMSG2  DC    H'41'                  LENGTH OF THE TEXT SEGMENT
*                                     INCLUDING THE HEADER.
         DC    H'0'                   OFFSET MUST BE ZERO.
         DC    CL37' PROCEDURE NAME REQUIRED BY PROCESSOR'
*                                     SECOND-LEVEL MESSAGE.  NOTE THAT
*                                     IT MUST NOT HAVE A MESSAGE ID
         IKJCPPL                      CPPL DSECT.  THIS EXPANDS WITH
*                                     THE SYMBOLIC ADDRESS CPPL.
         END
```

Using PUTGET to Put a Message Out to the Terminal and Obtain a Line of Input in Response

Use the PUTGET macro instruction to put messages out to the terminal and to obtain a response to those messages. A message to the user at the terminal which requires a response is called a conversational message. There are two types of conversational messages:

Mode messages
These messages tell the user at the terminal which processing mode is active so the user can enter a response applicable to that processing mode. Examples of mode messages are the READY message sent to the terminal by the terminal monitor program to indicate that it expects a command to be entered, and the command name, such as EDIT or TEST, sent by a Command Processor to indicate that it is ready to accept a subcommand name.

Prompt messages
These messages prompt the user at the terminal to enter parameters required by the program in control, or to reenter those parameters which were previously entered incorrectly.

When you issue a PUTGET macro instruction, the PUTGET service routine obtains a line of input from either:

- The terminal or the REXX data stack
- An in-storage list (including a command procedure)

PUTGET determines the source of input from the top element of the input stack unless you have specified the TERM or ATTN operands on the PUTGET macro instruction.

The input line returned by the PUTGET service routine can come from the terminal or an in-storage list, or from the REXX data stack; PUTGET determines the source of input from the top element of the input stack unless you have specified the TERM or ATTN operands in the PUTGET macro instruction.

PUTGET, like PUTLINE and GETLINE, has many parameters. The parameters are passed to the PUTGET service routine according to the operands you code in the list and the execute forms of the PUTGET macro instruction.

This topic describes:

- The list and execute forms of the PUTGET macro instruction
- Building the PUTGET parameter block
- Types and formats of the output line
- Passing the message lines to PUTGET
- PUTGET processing
- Input line format - the input buffer
- Return codes from PUTGET

The List Form of the PUTGET macro instruction

The list form of the PUTGET macro instruction builds and initializes a PUTGET parameter block (PGPB), according to the operands you specify in the PUTGET macro instruction. The PUTGET parameter block indicates to the PUTGET service routine which of the PUTGET functions you want performed.

In the list form of the PUTGET macro instruction, only

```
PUTGET      MF=L
```

is required.

The output line address is not specifically required in the list form of the PUTGET macro instruction, but must be coded in either the list or the execute form.

The other operands and their sublists are optional because you can supply them in the execute form of the macro instruction, or if you want the default values, they are supplied automatically by the expansion of the macro instruction.

The operands you specify in the list form of the PUTGET macro instruction set up control information used by the PUTGET service routine. This control information is passed to the PUTGET service routine in the PUTGET parameter block, a four-word parameter block built and initialized by the list form of the PUTGET macro instruction.

Figure 103 on page 234 shows the list form of the PUTGET macro instruction; each of the operands is explained following the figure.

```
                          [                                    {+
,PROMPT}            ]
 [symbol]    PUTGET       [OUTPUT=(output address {,SINGLE } {+
,MODE  } {,NOTRANS})]
                          [                        {,MULTLVL} {+
,PTBYPS} {,TRANS  } ]
                          [                                    {,TERM  }            ]
                          [                                    {,ATTN  }            ]

                          [            {EDIT   }                                  ]
                          [,TERMPUT=( {ASIS   } {,WAIT  } {+
,NOHOLD} {,NOBREAK})]
                          [            {CONTROL} {,NOWAIT} {+
,HOLD  } {,BREAKIN} ]

                          [,TERMGET=( {EDIT} {,WAIT  })] ,MF=L
                          [           {ASIS} {,NOWAIT} ]

                          [,SUBSTACK=( {NO })]
                          [             {YES} ]
```

Figure 103. The List Form of the PUTGET macro instruction

OUTPUT=*output address*

Specify the address of the output line descriptor or a zero. The output line descriptor (OLD) describes the message to be put out, and contains the address of the beginning (the one-word header) of the message or messages to be written to the terminal. You have the option under MODE processing to provide or not provide an output message. If you do not provide an output line, code OUTPUT=0, and only the GET functions will take place. If you do provide an output message, the type of message and the processing to be performed by the PUTGET service routine are described by the OUTPUT sublist operands SINGLE, MULTLVL, PROMPT, MODE, PTBYPS, TERM, ATTN, NOTRANS, and TRANS. SINGLE, PROMPT, and NOTRANS are the default values.

SINGLE | MULTLVL

SINGLE
The output message is a single-level message.

MULTLVL
The output message consists of multiple levels. The first-level message is written to the terminal, the second-level messages are printed at the terminal, one at a time, in response to question marks entered from the terminal. PROMPT must also be specified or defaulted.

PROMPT
The output line is a prompt message.

MODE
The output line is a mode message.

PTBYPS
The output line is a prompt message and the terminal user's response will not be displayed at those terminals that support the print inhibit feature. A terminal user can override bypass processing by pressing an attention followed by pressing the Enter key before entering input.

TERM
indicates to PUTGET that the current source of input, indicated by the top element of the input stack, is to be ignored. The output line, which is a mode message, is to be written to the terminal. Input is to be returned from the REXX data stack (if elements exist) or from the terminal. For more information about how PUTGET determines the source of input, refer to “What Is the Input Source? ” on page 248.

ATTN
specifies that the output line, which is a mode message, is to be initially suppressed, but an input line is to be returned from the terminal.

NOTRANS | TRANS

NOTRANS
specifies that the output line is not to be translated.

TRANS
specifies that the output line is to be written in the language specified in the user profile table (UPT).

Note: For more information about providing translated messages, see "PUTLINE Message Line Processing" on page 224.

TERMPUT=
specifies the options requested. The options are EDIT, ASIS or CONTROL, WAIT, or NOWAIT, NOHOLD or HOLD, and NOBREAK or BREAKIN. The default values are EDIT, WAIT, NOHOLD, and NOBREAK.

EDIT | ASIS | CONTROL

EDIT
specifies that in addition to minimal editing (see ASIS), the following functions are requested:

1. Any trailing blanks are removed before the line is written to the terminal. If a blank line is sent, the terminal vertically spaces one line.
2. Control characters are added to the end of the output line to position the cursor to the beginning of the next line.
3. All terminal control characters (for example, bypass, restore, horizontal tab, new line) are replaced with a printable character. Backspace is an exception; see item 4 under ASIS.

ASIS
specifies that minimal editing is to be performed as follows:

1. The line of output is to be translated from EBCDIC to terminal code. Incorrect characters will be converted to printable characters to prevent program caused I/O errors. This does not mean that all unprintable characters will be eliminated. Restore, upper case, lower case, bypass, and bell ring, for example, might be valid but nonprinting characters at some terminals. (See CONTROL.)
2. Transmission control characters will be added.
3. EBCDIC NL, placed at the end of the message, indicates that the cursor is to be returned at the end of the line. NL is replaced with whatever is necessary for that particular terminal type to cause the cursor to return. This NL processing occurs only if you specify ASIS, and the NL is the last character in your message.

   If you specify EDIT, NL is handled as described in item 3 under EDIT.

   If the NL is embedded in your message, a semicolon or colon may be substituted for NL and sent to the terminal. No idle characters are added (see item 6 below). This might cause overprinting, particularly on terminals that require a line-feed character to position the cursor on a new line.
4. If you have used backspace in your output message but the backspace character does not exist on the terminal type to which the message is being routed, the PUTGET service routine attempts alternate methods to accomplish the backspace.
5. Control characters are added as needed to cause the message to occur on several lines if the output line is longer than the terminal line size.
6. Idle characters are sent at the end of each line to prevent typing as the carrier returns.

No line continuation checking is done.

CONTROL
specifies that the output line is composed of terminal control characters and will not display or move the cursor on the terminal. This option should be used for transmission of characters such as bypass, restore, or bell ring. See ASIS for additional information.

WAIT | NOWAIT

WAIT
specifies that control will not be returned to the program that issued the PUTGET until the output line has been placed into a terminal output buffer.

NOWAIT
specifies that control should be returned to the program that issued the PUTGET macro instruction, whether or not a terminal output buffer is available. If no buffer is available a return code of 16 (decimal) is returned.

NOHOLD | HOLD

NOHOLD
specifies that control is to be returned to the issuer of the PUTGET macro instruction, and that program can resume processing as soon as the output line has been placed on the output queue.

HOLD
specifies that the program that issued the PUTGET macro instruction cannot continue its processing until this output line has been put out to the terminal or deleted.

NOBREAK | BREAKIN

NOBREAK
specifies that if the terminal user has started to enter input, transmission is not to be interrupted. The output message is placed on the output queue to be displayed after the terminal user has completed the line.

BREAKIN
specifies that output has precedence over input. If the user at the terminal is transmitting, transmission is interrupted, and this output line is sent. Any data that was received before the interruption is kept and displayed at the terminal following this output line.

TERMGET=
specifies the options requested. The options are EDIT or ASIS, and WAIT or NOWAIT. The default values are EDIT and WAIT.

EDIT | ASIS

EDIT
specifies that in addition to minimal editing (see ASIS), the buffer is to be padded with trailing blanks.

ASIS
specifies that minimal editing is to be done as follows:

1. Transmission control characters are removed.
2. The line of input is translated from terminal code to EBCDIC.
3. Line-deletion and character-deletion editing is performed.
4. Line feed and cursor return characters, if present, are removed.

No line continuation checking is done.

WAIT | NOWAIT

WAIT
specifies that control is to be returned to the program that issued the PUTGET macro instruction, only after an input message has been read.

NOWAIT
specifies that control should be returned to the program that issued the PUTGET macro instruction whether or not a line of input is available. If a line of input is not available, a return code of 20 (decimal) is returned in register 15 to the Command Processor.

MF=L
indicates that this is the list form of the macro instruction.

SUBSTACK=

SUBSTACK=YES indicates that normal stack operations continue until PUTGET reaches a barrier element. PUTGET then passes the caller a return code indicating that a barrier element was reached. The barrier element remains on the stack until the caller explicitly deletes it. SUBSTACK=NO is the default value and indicates that the barrier feature is not used.

Note: If the caller issues PUTGET without SUBSTACK=YES, and a barrier element exists on the input stack, normal stack operations continue until PUTGET reaches a barrier element. In foreground mode, PUTGET then treats the barrier element as a terminal element. In background mode, PUTGET passes an end-of-data return code to the caller. Processing continues in this manner until the caller explicitly deletes the barrier element.

The Execute Form of the PUTGET macro instruction

Use the execute form of the PUTGET macro instruction to do the following:

- Prepare a mode or a prompt message for output to the terminal.
- Determine whether that message should be sent to the terminal.
- Return a line of input from the source indicated by the top element of the input stack to the program that issued the PUTGET macro instruction.

You can use the execute form of the PUTGET macro instruction to build and initialize the input/output parameter list required by the PUTGET service routine, and to request PUTGET functions not already requested by the list form of the macro instruction, or to change those functions previously requested in either a list form or a previous execute form of the PUTGET macro instruction.

In the execute form of the PUTGET macro instruction, only the following is required:

```
PUTGET      MF=(E,{list address})
                  {    (1)     }
```

The PARM, UPT, ECT, and ECB operands are not required if you have built your IOPL in your own code.

The output line address is not specifically required in the execute form of the PUTGET macro instruction, but must be coded in either the list or the execute form.

The other operands and sublists are optional because you can supply them in the list form of the macro or in a previous execute form, or because you might want to use the default values which are automatically supplied by the macro expansion itself.

The operands you specify in the execute form of the PUTGET macro set up control information used by the PUTGET service routine. You can use the PARM, UPT, ECT, and ECB operands of the PUTGET macro to build, complete, or modify an IOPL. The OUTPUT, TERMPUT, and TERMGET operands and their sublist operands initialize the PUTGET parameter block. The PUTGET parameter block is referred to by the PUTGET service routine to determine which functions you want PUTGET to perform.

Figure 104 on page 238 shows the execute form of the PUTGET macro instruction; each of the operands is explained following the figure.

```
 [symbol]   PUTGET       [PARM=parameter address] [,UPT=upt address)
                         [,ECT=ect address ] [,ECB=ecb address]

                                                    {,PROMPT}
                         [OUTPUT=(output address {,SINGLE } {+
,MODE  } {,NOTRANS})]
                         [                        {,MULTLVL} {,PTBYPS} {,TRANS  } ]
                         [                                   {,TERM  }            ]
                         [                                   {,ATTN  }            ]

                         [           {EDIT   }                                    ]
                         [,TERMPUT=( {ASIS   } {,WAIT  } {+
,NOHOLD} {,NOBREAK})]
                         [           {CONTROL} {,NOWAIT} {+
,HOLD  } {,BREAKIN} ]

                         [,TERMGET=( {EDIT} {,WAIT  })]
                         [           {ASIS} {,NOWAIT} ]

                         [ ,ENTRY={entry address}]+
  ,MF=(E {,list address})
                         [        {    (15)     }]+
        {     (1)     }

                         [,SUBSTACK=( {NO })]
                         [            {YES} ]
```

Figure 104. The Execute Form of the PUTGET macro instruction

PARM=*parameter address*
specifies the address of the four-word PUTGET parameter block (PGPB). This address is placed into the input/output parameter list (IOPL). It can be the address of a list form of the PUTGET macro instruction. The address is any address valid in an RX instruction, or you can put it in one of the general registers 2–12, and use that register number, enclosed in parentheses, as the parameter address.

UPT=*upt address*
specifies the address of the user profile table (UPT). This address is placed into the IOPL when the execute form of the PUTGET macro instruction expands. You can obtain this address from the Command Processor parameter list (CPPL) pointed to by register 1 when the Command Processor is attached by the terminal monitor program. The address can be used as received in the CPPL or you can put it in one of the general registers 2–12, and use that register number, enclosed in parentheses, as the UPT address.

ECT=*ect address*
specifies the address of the environment control table (ECT). This address is placed into the IOPL when the execute form of the PUTGET macro instruction expands. You can obtain this address from the Command Processor parameter list (CPPL) pointed to by register 1 when the Command Processor is attached by the terminal monitor program. The address can be used as received in the CPPL or you can put it in one of the general registers 2–12, and use that register number, enclosed in parentheses, as the ECT address.

ECB=*ecb address*
specifies the address of the Command Processor event control block (ECB). This address is placed into the IOPL by the execute form of the PUTGET macro instruction when it expands.

You must provide a one-word event control block and pass its address to the PUTGET service routine by placing the address into the IOPL. If you code the address of the ECB in the execute form of the PUTGET macro instruction, the macro instruction places the address into the IOPL for you. The address can be any address valid in an RX instruction, or you can put it in one of the general registers 2–12, and use that register number, enclosed in parentheses, as the ECB address.

If an attention interruption occurs while a mainline routine's PUTGET macro is prompting for input, and if an attention exit was previously identified by the STAX macro, the exit receives control to process the attention request. If the attention routine sets the completion bit by posting the mainline

routine's PUTGET ECB, then the mainline PUTGET receives a return code 8. However, if the attention routine does not set the completion bit, PUTGET continues as if the attention interruption never occurred.

OUTPUT=*output address*
specifies the address of the output line descriptor or a zero. The output line descriptor (OLD) describes the message to be issued, and contains the address of the beginning (the one-word header) of the message or messages to be written to the terminal. You have the option under MODE processing to provide or not provide an output message. If you do not provide an output line, code OUTPUT=0, and only the GET function will take place. If you do provide an output message, the type of message and the processing to be performed by the PUTGET service routine are described by the OUTPUT sublist operands SINGLE, MULTLVL, PROMPT, MODE, PTBYPS, TERM, ATTN, NOTRANS, and TRANS. The default values are SINGLE, PROMPT, and NOTRANS.

SINGLE | MULTLVL

SINGLE
The output message is a single-level message.

MULTLVL
The output message consists of multiple levels. The first-level message is written to the terminal, the second-level messages are displayed at the terminal, one at a time, in response to question marks entered from the terminal. PROMPT must also be specified or defaulted.

PROMPT
The output line is a prompt message.

MODE
The output line is a mode message.

PTBYPS
The output line is a prompt message and the terminal user's response will not display at those terminals that support the print inhibit feature. A terminal user can override bypass processing by pressing an attention followed by pressing the Enter key before entering input.

TERM
specifies that the output line, which is a mode message, is to be written to the terminal, and a line is to be returned from the terminal, regardless of the top element of the input stack.

ATTN
specifies that the output line, which is a mode message, is to be initially suppressed, but an input line is to be returned from the terminal.

NOTRANS | TRANS

NOTRANS
specifies that the output line is not to be translated.

TRANS
specifies that the output line is to be written in the language specified in the user profile table (UPT).

Note: For more information about providing translated messages, see "PUTLINE Message Line Processing" on page 224.

TERMPUT=
specifies the options requested. The options are EDIT, ASIS or CONTROL, WAIT or NOWAIT, NOHOLD or HOLD, and NOBREAK or BREAKIN. The default values are EDIT, WAIT, NOHOLD and NOBREAK.

EDIT | ASIS | CONTROL

EDIT
specifies that in addition to minimal editing (see ASIS), the following TPUT functions are requested:

1. Any trailing blanks are removed before the line is written to the terminal. If a blank line is sent, the terminal vertically spaces one line.

2. Control characters are added to the end of the output line to position the cursor to the beginning of the next line.
3. All terminal control characters (for example, bypass, restore, horizontal tab, new line) are replaced with a printable character. Backspace is an exception; see item 4 under ASIS.

ASIS
specifies that minimal editing is to be performed as follows:

1. The line of output is translated from EBCDIC to terminal code. Incorrect characters are converted to a printable character to prevent program caused I/O errors. This does not mean that all unprintable characters will be eliminated. Restore, upper case, lower case, bypass, and bell ring, for example, might be valid but nonprinting characters at some terminals. (See CONTROL.)
2. Transmission control characters are added.
3. EBCDIC NL, placed at the end of the message, indicates that the cursor is to be returned at the end of the line. NL is replaced with whatever is necessary for that particular terminal type to cause the cursor to return. This NL processing occurs only if you specify ASIS, and the NL is the last character in your message.

   If you specify EDIT, NL is handled as described in item 3 under EDIT.

   If the NL is embedded in your message, a semicolon or colon may be substituted for NL and sent to the terminal. No idle characters are added (see item 6 below). This might cause overprinting, particularly on terminals that require a line-feed character to position the cursor on a new line.
4. If you have used backspace in your output message, but the backspace character does not exist on the terminal type to which the message is being routed, the PUTGET service routine attempts alternate methods to accomplish the backspace.
5. Control characters are added as needed to cause the message to occur on several lines if the output line is longer than the terminal line size.
6. Idle characters are sent at the end of each line to prevent typing as the cursor returns.

No line continuation checking is done.

CONTROL
specifies that this line is composed of terminal control characters and will not display or move the cursor on the terminal. This option should be used for transmission of characters such as bypass, restore, or bell ring.

WAIT | NOWAIT

WAIT
specifies that control will not be returned to the program that issued the PUTGET until the output line has been placed into the terminal output buffer.

NOWAIT
specifies that control should be returned to the program that issued the PUTGET macro instruction, whether or not a terminal output buffer is available. If no buffer is available, a return code of 16 (decimal) is returned.

NOHOLD | HOLD

NOHOLD
specifies that control is to be returned to the program that issued the PUTGET macro instruction, and it can continue processing as soon as the output line has been placed on the output queue.

HOLD
specifies that the program that issued the PUTGET macro instruction cannot continue its processing until the output line has been put out to the terminal or deleted.

NOBREAK | BREAKIN

NOBREAK
specifies that if the terminal user has started to enter input, transmission is not to be interrupted. The output message is placed on the output queue to be displayed after the terminal user has completed the line.

BREAKIN
specifies that output has precedence over input. If the user at the terminal is transmitting, the user is interrupted, and this output line is sent. Any data that was received before the interruption is kept and displayed at the terminal following this output line.

TERMGET=
specifies the options requested. The options are EDIT or ASIS, and WAIT or NOWAIT. The default values are EDIT and WAIT.

EDIT | ASIS

EDIT
specifies that in addition to minimal editing (see ASIS), the buffer is filled out with trailing blanks.

ASIS
specifies that minimal editing is done as follows:

1. Transmission control characters are removed.
2. The line of input is translated from terminal code to EBCDIC.
3. Line-deletion and character-deletion editing is performed.
4. Line feed and cursor return characters, if present, are removed.

No line continuation checking is done.

WAIT | NOWAIT

WAIT
specifies that control is to be returned to the program that issued the PUTGET macro instruction, only when an input message has been read.

NOWAIT
specifies that control should be returned to the program that issued the PUTGET macro instruction whether or not a line of input is available. If a line of input is not available, a return code of 20 (decimal) is returned in register 15.

ENTRY=*entry point address* | (15)
specifies the entry point of the PUTGET service routine. If ENTRY is omitted, the PUTGET macro expansion generates a LINK macro instruction to invoke the PUTGET service routine. The address can be any address valid in an RX instruction or (15) if you load the entry point address into general register 15.

MF=E
indicates that this is the execute form of the PUTGET macro instruction.

***listaddr* | (1)**
The address of the four-word input/output parameter list (IOPL). This can be a completed IOPL that you have built, or it can be 4 words of declared storage that will be filled from the PARM, UPT, ECT, and ECB operands of this execute form of the PUTGET macro instruction. The address must be any address valid in an RX instruction or (1) if you have loaded the parameter list address into general register 1.

SUBSTACK
SUBSTACK=YES indicates that normal stack operations continue until PUTGET reaches a barrier element. PUTGET then passes the caller a return code indicating that a barrier element was reached. The barrier element remains on the stack until the caller explicitly deletes it. SUBSTACK=NO is the default value and indicates that the barrier feature is not used.

Note: If the caller issues PUTGET without SUBSTACK=YES, and a barrier element exists on the input stack, normal stack operations continue until PUTGET reaches the barrier element. In foreground mode, PUTGET then treats the barrier element as a terminal element. In background mode, PUTGET passes an end-of-data return code to the caller. Processing continues in this manner until the caller explicitly deletes the barrier element.

Building the PUTGET Parameter Block (PGPB)

When the list form of the PUTGET macro instruction expands, it builds a four-word PUTGET parameter block (PGPB). This PGPB combines the functions of the PUTLINE and the GETLINE parameter blocks and contains information used by the PUT and the GET functions of the PUTGET service routine. The list form of the PUTGET macro instruction initializes this PGPB according to the operands you have coded in the macro instruction. This initialized block, which you can later modify with the execute form of the PUTGET macro instruction, indicates to the PUTGET service routine the functions you want performed. It also contains a pointer to the output line descriptor that describes the output message, and it provides a field into which the PUTGET service routine places the address of the input line returned from the input source.

You must pass the address of the PGPB to the execute form of the PUTGET macro instruction. Because the list form of the macro instruction expands into a PGPB, all you need do is pass the address of the list form of the macro instruction to the execute form as the PARM value.

The PUTGET parameter block is defined by the IKJPGPB DSECT, which is provided in SYS1.MACLIB. Table 66 on page 242 describes the contents of the PUTGET parameter block.

Table 66. The PUTGET parameter block

Number of bytes	Field name	Contents or meaning
2		PUT control flags. These bits describe the output line to the PUTGET service routine. Byte 1: **..0. ....** Always zero. **...1 ....** The output line is a single-level message. **.... 0...** Must be zero. **.... .1..** The output line is a multilevel message. **.... ...1** The output line is a PROMPT message. **xx.. ..x.** Reserved. Byte 2: **1... ....** The output line is a MODE message. **...1 ....** BYPASS processing is requested. **.... 1...** ATTN processing is requested. **.... .1..** SUBSTACK=YES is specified. **.... ..1.** The output line is to be written in the language specified in the UPT. **.xx. ...x** Reserved bits.

Table 66. The PUTGET parameter block (continued)

Number of bytes	Field name	Contents or meaning
2		PUT options field. These bits indicate to the PUTGET service routine which of the options you want to use for PUT. Byte 1: **0... ....** Always set to 0. **...0 ....** WAIT processing has been requested. Control will be returned to the issuer after the output line has been placed into a terminal output buffer. **...1 ....** NOWAIT processing has been requested. Control will be returned to the issuer whether or not a terminal output buffer is available. **...0 ....** NOHOLD processing has been requested. The issuer can resume processing as soon as the output line has been placed on the output queue. **.... 1...** HOLD processing has been requested. The issuer is not to resume processing until the output line has been written to the terminal or deleted. **.... .0..** NOBREAK processing has been requested. The output line will be displayed only when the terminal user is not entering a line. **.... .1..** BREAKIN processing has been requested. The output line is to be sent to the terminal immediately. If the terminal user is entering a line, the user is to be interrupted. **.... ..00** EDIT processing has been requested. **.... ..01** ASIS processing has been requested. **.... ..10** CONTROL processing has been requested. **.xx. ....** Reserved. Byte 2: Reserved.
4		The address of the output line descriptor.
2		GET control flags. Byte 1: **.00. ....** Always zero. **...1 ....** TERM processing is requested. **x... xxxx** Reserved bits. Byte 2: **xxxx xxxx** Reserved.

Table 66. The PUTGET parameter block (continued)

Number of bytes	Field name	Contents or meaning
2		GET options field. These bits indicate to the PUTGET service routine which of the options you want to use for GET. Byte 1: **1... ....** Always set to 1. **...0 ....** WAIT processing has been requested. Control will be returned to the issuer only after an input message has been read. **...1 ....** NOWAIT processing has been requested. Control will be returned to the issuer whether or not a line of input is available. If no line was available, PUTGET returns a code of 20 (decimal) in general register 15. **.... ..00** EDIT processing has been requested. In addition to the editing provided by ASIS processing, the input buffer is to be filled out with trailing blanks to the next doubleword boundary. **.... ..01** ASIS processing has been requested. (See the ASIS operand of the PUTGET macro instruction description.) **.xx. xx..** Reserved bits. Byte 2: **xxxx xxxx** Reserved.
4	PGPBIBUF	The address of the input buffer. The PUTGET service routine fills this field with the address of the input buffer in which the input line has been placed.

Types and Formats of the Output Line

The PUTGET service routine writes only conversational messages to the terminal, it does not handle data lines. For information on how to write a data line or a nonconversational message to the terminal, see “Using PUTLINE to Put a Line Out to the Terminal” on page 209.

PUTGET accepts two output line formats depending upon whether the message you provide is a single-level message or a multilevel message.

Single-Level Messages

A single-level message is composed of one or more message segments to be formatted and written to the terminal with one execution of the PUTGET macro instruction.

Multilevel Messages

A multilevel message is composed of one or more segments to be formatted and written to the terminal, and one or more message segments to be formatted and written to the terminal in response to question marks entered from the terminal. Note, however, that if you specify MODE in the PUTGET macro instruction, you can process only single-level messages. To have second-level messages written to the terminal, one at a time, in response to successive question marks entered from the terminal, specify PROMPT and TERMGET=EDIT on the PUTGET macro instruction. Note that if you specify PUTGET with TERMGET=ASIS, the user's terminal will not recognize the question mark. If PROMPT messages are to be available to the user at the terminal, the top element of the input stack must not specify a procedure element as the current source of input, and the terminal user must not have inhibited prompting. (See the PROFILE command in *z/OS TSO/E User's Guide*.)

Passing the Message Lines to PUTGET

You must build each of the message segments to be processed by the PUTGET service routine as if it were a line of single line data. The segment must be preceded by a four-byte header field, where the first two bytes contain the length of the segment including the header, and the second two bytes contain zeros or an offset value if you use the text insertion facility provided by PUTGET. This message line format is required whether the message is a single-level message or a multilevel message.

Because of the additional functions performed on message lines, (message ID stripping, text insertion, and multilevel processing), you must provide the PUTGET service routine with a description of the line or lines that are to be processed. This is done with an output line descriptor (OLD).

There are two types of output line descriptors. The type depends on whether the messages are single level or multilevel.

The OLD required for a single-level message is a variable-length control block which begins with a fullword value representing the number of segments in the message, followed by fullword pointers to each of the segments.

The format of the OLD for multilevel messages varies from that required for single-level messages in only one respect. You must preface the OLD with a fullword forward-chain pointer. This chain pointer points to another output line descriptor or contains zero to indicate that it is the last OLD on the chain. Table 67 on page 245 shows the format of the output line descriptor.

Table 67. The output line descriptor (OLD)

Number of bytes	Field name	Contents or meaning
4		The address of the next OLD, or zero if this is the last one on the chain. This field is present only if the message pointed to is a multilevel message.
4		The number of message segments pointed to by this OLD.
4		The address of the first message segment.
4		The address of the next message segment.
4		The address of the nth message segment.

You must build the output line descriptor and pass its address to the PUTLINE service routine as the OUTPUT operand address in either the list or the execute form of the macro instruction. When the macro instruction expands, it places this OLD address into the second word of the PUTLINE parameter block.

Figure 105 on page 247 shows the two control block structures possible when passing an output message to the PUTGET service routine. Note that MODE, TERM, or ATTN cannot be coded in the PUTGET macro instruction if you want to provide multilevel messages to the terminal because mode messages can have only one level.

Message segments for PUTGET must follow the same rules as those for PUTLINE informational processing. (See “Stripping Message Identifiers” on page 224.) Note that if a PUTGET message segment does not contain at least one blank, PUTGET sets a return code of 24, indicating not valid parameters, in register 15.

PUTGET Processing

Text insertion, message identifier stripping, and text translation are available to all output messages processed by the PUTGET service routine. For a detailed description of these functions see “PUTLINE Message Line Processing” on page 224.

The PUTGET service routine provides other processing capabilities depending upon whether the message is a mode or a prompt message.

Mode Message Processing

A mode message is a message put out to the terminal when a command or a subcommand is anticipated. The processing of mode messages by the PUTGET service routine is dependent upon the following two conditions:

- Are you providing an output line?
- From what source is the input line coming?

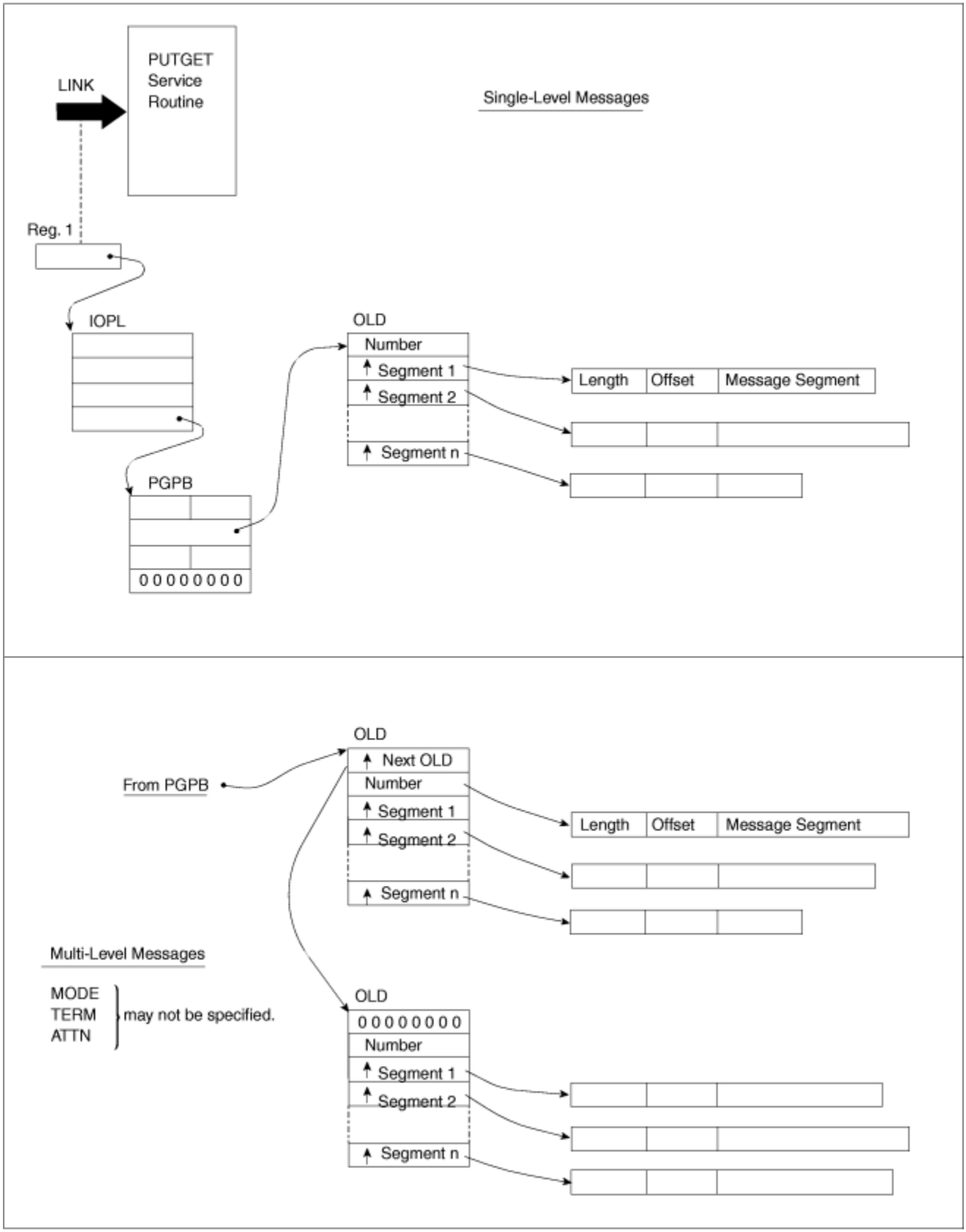


Figure 105. Control Block Structures for PUTGET Output Messages

Is an Output Line Present?

You are not required to provide an output line to the PUTGET service routine. If you do provide an output line address then PUT processing will take place. Whether your output line is written to the terminal is then dependent upon the input source indicated by the input stack. If you do not provide an output line (OUTPUT=0) then only the GET function of the PUTGET service routine takes place.

What Is the Input Source?

The PUTGET service routine obtains a line of input from either:

- The REXX data stack
- The input source described by the topmost element of the input stack.

A Command Processor executing in a REXX exec can use the data stack to communicate with the user using PUTGET in one of two methods:

- With PUTGET prompt message processing, for example:

```
/* rexx */
x = prompt('on')      /* Prompting must first be enabled                */
queue "DISPLAY"       /* Responds to the prompt for an action           */
address tso "ALTLIB"  /* Command processor that needs a prompt satisfied */
```

- With PUTGET mode message processing, for example:

```
/* rexx */
queue "ALTLIB DISPLAY" /* Queued for later execution                    */
queue "PROFILE"        /* Queued after the above command               */
exit                   /* Leave the exec and execute commands          */
```

In each of these cases, if you do not specify either the ATTN or TERM operand, PUTGET obtains input from the REXX data stack first, if there are elements on the REXX data stack, and if the topmost element on the input stack is either a terminal element or a barrier element. When PUTGET has processed all lines of input on the data stack, it then obtains input from the terminal.

When the topmost element on the input stack is an in-storage list element (including a command procedure), PUTGET obtains input from the source indicated by the in-storage list element. This ensures compatibility with applications that are not sensitive to the REXX data stack (for example, a CLIST invoked from within a REXX exec).

When you specify PUTGET with the TERM operand, PUTGET obtains input from the REXX data stack first, if there are elements on the REXX data stack. If there are no elements on the REXX data stack, PUTGET returns input from the terminal.

When you specify PUTGET with the ATTN operand, the input source is the terminal.

A Command Processor can determine the source of input with which PUTGET will satisfy an input request according to the following procedure:

1. If you specify PUTGET OUTPUT=ATTN, the input is from the terminal.
2. If you specify PUTGET OUTPUT=TERM, the input is from the REXX data stack (if elements exist on the REXX data stack), or from the terminal. To determine if elements exist on the REXX data stack, use step “4” on page 249.
3. Before you specify PUTGET without OUTPUT=ATTN or OUTPUT=TERM, first invoke the STACK macro with the INQUIRE=TYPE operand to determine the type of element on the top of the input stack.
   a. If the top element of the input stack is an in-storage list (for example, a command procedure), the source indicated by the in-storage list is the source of input.
   b. If the top element of the input stack is a barrier element that is *not* a NONEST barrier element (indicated by a decimal return code of 44 from STACK), the end of the substack has been reached. PUTGET returns a return code or considers the barrier a terminal element, depending on what was specified on the SUBSTACK operand. For more information on the SUBSTACK operand, see “The Execute Form of the PUTGET macro instruction” on page 237.

c. If the top element of the input stack is a NONEST barrier element (indicated by a decimal return code of 80 from STACK) and if there are elements on the REXX data stack, the source of input is the REXX data stack. Otherwise, the NONEST barrier acts as a BARRIER=* element as described in step "3.b" on page 248. To determine if elements exist on the REXX data stack, use step "4" on page 249.

d. If the top element of the input stack is a terminal element, the source of input is the REXX data stack (if there are elements on the REXX data stack), or the terminal. To determine if elements exist on the REXX data stack, use step "4" on page 249.

4. To determine if elements exist on the REXX data stack, invoke the REXX data stack replaceable routine, IRXSTK, with the QUEUED function. If the number of queued elements is greater than zero, elements exist on the REXX data stack. Otherwise, the source of input is the terminal.

   Note: If the source of input might be the REXX data stack, and if the Command Processor is invoked by a CLIST and a CLIST DATA-ENDDATA group exists, input is from the CLIST DATA-ENDDATA group.

Mode Message Response Processing

The source of the input line, as determined by the top element of the input stack, determines the type of processing performed by the PUTGET service routine.

- When you provide an output line and the current source of input is the terminal (there are no elements on the REXX data stack), the PUTGET service routine:
  1. Puts out the mode message to the terminal.
  2. Returns a line from the terminal.
  3. Places the address of the returned line into the fourth word of the PUTGET parameter block.
- If the line returned from the terminal is a question mark, the PUTGET service routine:
  1. Writes the second-level message (if one exists) for a message written before the mode message, to the terminal. If no second-level message exists, PUTGET puts out message `IKJ66760I NO INFORMATION AVAILABLE`.
  2. Puts out the previously written mode message.
  3. Returns a line from the terminal.

Note: Whenever terminal input is expected and there are elements on the REXX data stack, the PUTGET service routine satisfies the input request from the REXX data stack rather than obtaining a line from the terminal.

Pause Processing

If the terminal user has requested the PAUSE option on the PROFILE command, the PUTGET service routine makes the second-level messages available to the terminal, even if the current input source is not the terminal.

PAUSE processing works as follows. If a second-level message does exist, PUTGET puts out a message to the terminal informing the terminal user that PAUSE processing is in effect. At this point the terminal user can enter either a question mark to request second-level messages be sent to the terminal, or press the Enter key to indicate that the information is not needed. If the user presses the Enter key, the second-level message is eliminated. If the user enters any response other than a question mark or hitting the Enter key, PUTGET prompts for a correct response.

Prompt Message Processing

A prompt message is a message that is issued to the terminal when the program in control requires input from the terminal user. PROMPT information must come from the terminal and cannot be obtained from any other source of input. There are three cases when a request for PROMPT processing is denied by PUTGET:

- When the current source of input, as determined by the top element of the input stack, is an in-storage procedure that is not an EXEC command procedure.

- When the NOPROMPT attribute is specified in the user's profile table (UPT).
- When an EXEC command procedure, executing in the background, does not have a DATA PROMPT entry to satisfy the request or a PROMPT control statement.

When the PUTGET service routine returns control to the program that invoked it, it returns a return code of 12 when no prompting was allowed on a PROMPT request because:

- The current source of input is an in-storage list other than an EXEC command procedure.
- The NOPROMPT attribute is specified in the user's profile table (UPT).
- The current source of input is an EXEC command procedure running in the background, and there is no DATA PROMPT entry to satisfy the request.

If PROMPT processing is enabled, the PUTGET service routine writes the first-level message to the terminal and obtains an input line from either the REXX data stack or the terminal. If the input line is a question mark, PUTGET either returns the next-level message provided or a message informing the user that no information is available. PUTGET continues to respond to each question mark by writing one more second-level message to the terminal until the chain is exhausted. PUTGET then issues a message informing the user that no more information is available. The task then goes into a wait state until the user enters a line. When the user enters a line, PUTGET places the address of the line into the fourth word of the PUTGET parameter block.

Note that for message prompting, PUTGET with TERMGET=EDIT is required.

Input Line Format - The Input Buffer

The fourth word of the PUTGET parameter block contains zeros until the PUTGET service routine returns a line of input. The service routine places the requested input line into an input buffer beginning on a doubleword boundary located in subpool 1. It then places the address of this input buffer into the fourth word of the PGPB.

Note: The application that invoked PUTGET should release the input buffer's storage to prevent the accumulation of unused storage. The application can free the storage with the FREEMAIN macro instruction after the application has processed or copied an input line.

For commands *not* running on a command invocation platform:

- Input buffer storage returned by PUTGET is automatically freed when the Command Processor relinquishes control.
- The application should free the input buffer's storage after it uses the storage. This prevents storage from accumulating while the application is running.

For commands running on a command invocation platform:

- Input buffer storage returned by PUTGET is not freed when the Command Processor relinquishes control.
- It is important to free the input buffer's storage after use to prevent the unused storage from accumulating during a TSO/E session.
- The storage cannot be freed after the application ends because the storage addresses are not known to new applications.

Regardless of the source of input, the input line returned by the PUTGET service routine is in a standard format. All input lines are in the variable-length record format with a fullword header followed by the text returned by PUTGET. Figure 106 on page 251 shows the format of the input buffer returned by the PUTGET service routine.

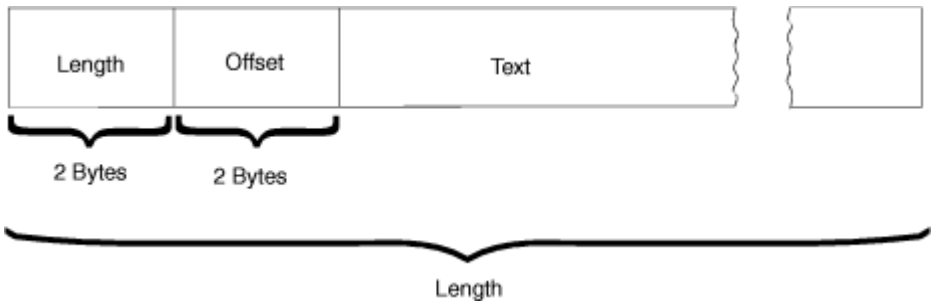


Figure 106. Format of the PUTGET Input Buffer

The two-byte length field contains the length of the returned input line including the header (4 bytes). You can use this length field to determine the length of the input line to be processed, and later, to free the input buffer with the R form of the FREEMAIN macro instruction. The two-byte offset field is always set to zero on return from the PUTGET service routine.

Figure 107 on page 252 shows the PUTGET control block structure for a multilevel PROMPT message after the PUTGET service routine has returned an input line.

Return Codes from PUTGET

When the PUTGET service routine returns control to the program that invoked it, PUTGET provides one of the following return codes in general register 15.

Table 68. Return codes from the PUTGET service routine

Return code dec(Hex)	Meaning
0(0)	PUTGET completed successfully. The line was obtained from either: the REXX data stack, a command procedure DATA-ENDDATA group, or the terminal.
4(4)	PUTGET completed successfully. The line was obtained from an in-storage list or command procedure. (MODE messages only.)
8(8)	The PUTGET service routine did not complete. An attention interruption occurred during the execution of PUTGET, and the attention handler turned on the completion bit in the communications ECB.
12(C)	One of the following situations occurred: • No prompting was allowed on a PROMPT request. Either the user at the terminal requested no prompting with the PROFILE command, or the current source of input is an in-storage list other than an EXEC command procedure. • A line could not be obtained after a MODE request. Second-level messages exist, and the current stack element is not a terminal, but the terminal user did not request PAUSE processing with the PROFILE command. The messages are, therefore, not available to him.
16(10)	One of the following situations occurred: • The NOWAIT option was specified for PUT processing and no line was put out. • A barrier element is on top of the stack, the current source of input is a data set, and SUBSTACK=NO was specified or defaulted. No command buffer is passed back.
20(14)	The NOWAIT option was specified for GET processing and no line was received.
24(18)	Incorrect parameters were supplied to the PUTGET service routine.

Table 68. Return codes from the PUTGET service routine (continued)	
Return code dec(Hex)	**Meaning**
28(1C)	PUTGET was unable to obtain sufficient storage to satisfy the request for output buffers.
32(20)	The terminal has been disconnected.
40(28)	A barrier element is on the top of the stack and SUBSTACK=YES was specified. No command buffer is passed back.

Note: User abend 204 is issued when the return code from PUTGET is greater than 12 and less than 40.

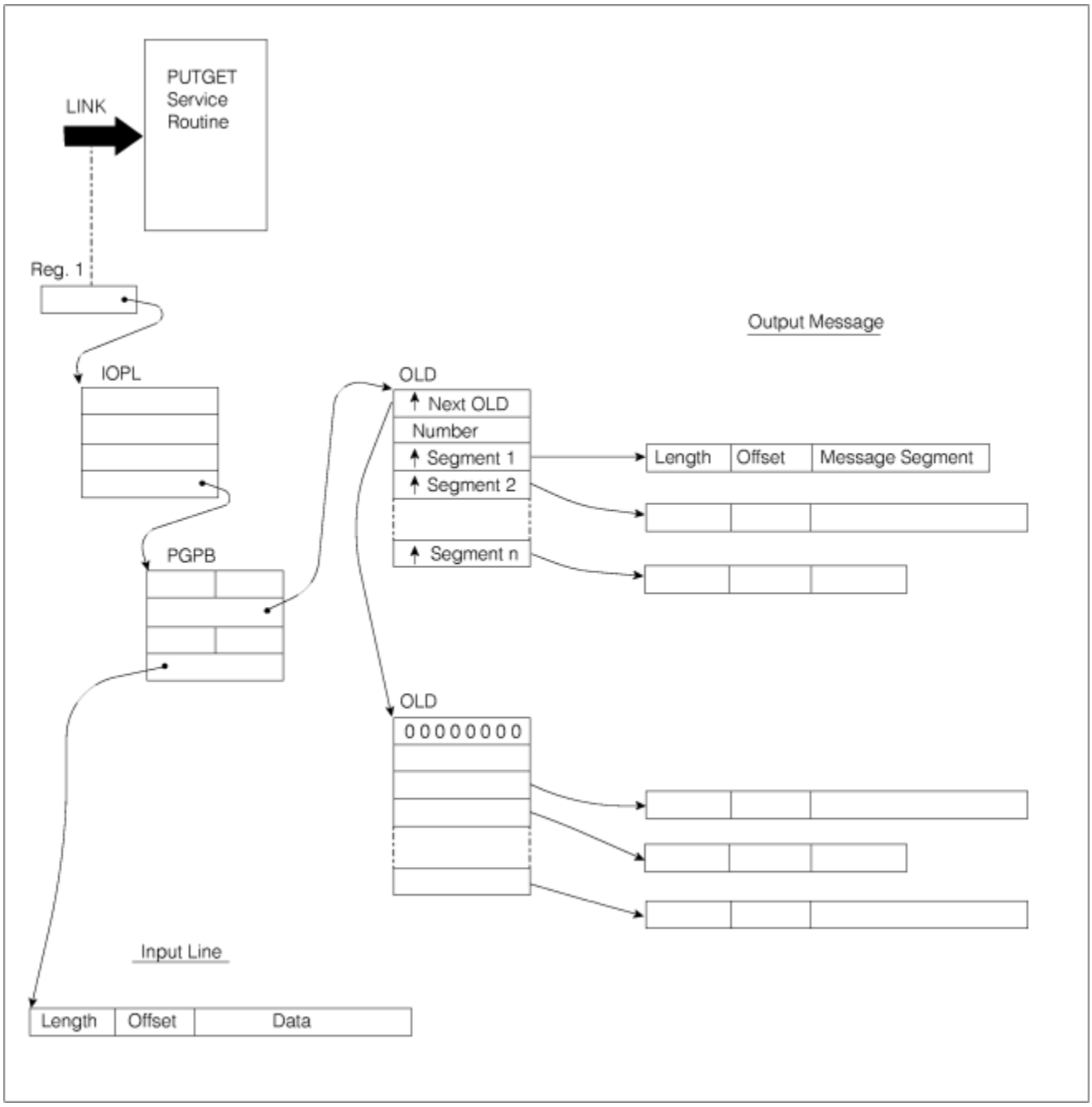


Figure 107. PUTGET Control Block Structure - Input Line Returned

An Example Using PUTGET

Figure 108 on page 254 is an example of the code required to execute the PUTGET macro instruction. The code uses a multilevel PROMPT message as the PUTGET output line. It assumes that a line of input will be returned from the terminal and tests only for a zero return code (PUTGET completed normally).

The execute form of the PUTGET macro instruction builds the I/O parameter list, using the addresses of the user profile table and the environment control table supplied in the Command Processor parameter list. In addition, the I/O parameter list contains the address of an ECB built by the code, and the address of the list form of the PUTGET macro instruction as the PUTGET parameter block address.

Note that the TERMPUT, TERMGET, and ENTRY operands are not coded; the default values are used. Note also that this code is effective only if the top element of the input stack indicates a terminal as the current source of input.

```
* ON ENTRY FROM THE TMP, REGISTER 1 CONTAINS A POINTER TO THE COMMAND
* PROCESSOR PARAMETER LIST (CPPL).
*
*      SET UP ADDRESSABILITY
*      SAVE AREA CHAINING
*
         LR    2,1                   SAVE THE ADDRESS OF THE CPPL.
         USING CPPL,2                ADDRESSABILITY FOR THE CPPL
         L     3,CPPLUPT             PLACE THE ADDRESS IF THE UPT
*                                    INTO A REGISTER
         L     4,CPPLECT             PLACE THE ADDRESS OF THE ECT
*                                    INTO A REGISTER
* ISSUE AN EXECUTE FORM OF THE PUTGET MACRO INSTRUCTION.  THIS
* EXECUTION WRITES A PROMPTING MESSAGE TO THE TERMINAL AND CHAINS
* A SECOND-LEVEL MESSAGE.  IT ALSO FILLS IN THE IOPL.
*
         PUTGET    PARM=APGPB,UPT=(3),ECT=(4),ECB=ECBADS,             X
              OUTPUT=(FIRSTOLD,MULTLVL,PROMPT),MF=(E,IOPLADS)
*
* TEST THE CODE RETURNED BY THE PUTGET SERVICE ROUTINE.  A RETURN CODE
* OF ZERO INDICATES NORMAL COMPLETION.
*
         LTR   15,15                 IS THE RETURN CODE ZERO?
         BNZ   EXIT                  NO - BRANCH TO AN EXIT.
*                                    YES - FALL THROUGH AND OBTAIN
*                                    THE LINE RETURNED FROM THE
*                                    TERMINAL.
         LA    5,APGPB               SET ADDRESSABILITY FOR THE
         USING PGPB,5                PUTGET PARAMETER BLOCK.
         L     1,PGPBIBUF            GET THE ADDRESS OF THE LINE
*                                    RETURNED FROM THE TERMINAL.
```

```
* PROCESS THE INPUT LINE, AND WHEN FINISHED, FREE THE INPUT BUFFER
*
         LH    0,0(1)                PUT THE LENGTH OF THE INPUT
*                                    LINE (INCLUDING THE HEADER)
*                                    INTO REGISTER 0.
         O     0,=X'01000000'
*
         FREEMAIN   R,LV=(0),A=(1)   FREE THE INPUT BUFFER.
*      PROCESSING
*         .
*         .
*         .
*
```

```
EXIT     EXIT ROUTINES
*         .
*         .
*         .
APGPB    PUTGET    MF=L              LIST FORM OF THE PUTGET MACRO
*                                    INSTRUCTION.  IT EXPANDS TO
*                                    BUILD A PUTGET PARAMETER BLOCK.
ECBADS   DC   F'0'                   A FULLWORD OF STORAGE FOR THE
*                                    COMMAND PROCESSOR ECB.
IOPLADS  DC   4F'0'                  FOUR FULLWORDS FOR THE INPUT/
*                                    OUTPUT PARAMETER LIST
*
```

Figure 108. Example of PUTGET Issuing a Multilevel PROMPT Message

Figure 'Example of PUTGET Issuing a Multilevel PROMPT Message' (**Continued**)

```
* BUILD THE CHAIN OF OUTPUT LINE DESCRIPTORS AND OUTPUT MESSAGE
* SEGMENTS.
*
FIRSTOLD  DC   A(NEXTOLD)             POINTER TO THE NEXT OLD.
          DC   F'1'                   INDICATE ONLY ONE SEGMENT.
          DC   A(OUTMSG)              THE ADDRESS OF THE OUTPUT
*                                     MESSAGE.
NEXTOLD   DC   A(0)                   INDICATES THAT THIS IS THE
*                                     LAST OLD ON THE CHAIN.
          DC   F'1'                   INDICATES ONLY ONE SEGMENT.
          DC   A(CHNMSG)              ADDRESS OF THE SECOND LEVEL
*                                     MESSAGE TO BE CHAINED.
*
```

```
* THE PROMPTING MESSAGE AND THE SECOND-LEVEL MESSAGE ARE FORMATTED
* IDENTICALLY.  THE FORMAT IS:  A TWO BYTE LENGTH INDICATOR, A TWO
* BYTE OFFSET FIELD, AND THE VARIABLE-LENGTH TEXT FIELD.
*
OUTMSG    DC   H'31'                  LENGTH OF THE OUTPUT MESSAGE
*                                     INCLUDING THE FOUR BYTE HEADER.
          DC   H'0'                   THE OFFSET FIELD IS SET TO ZERO
*                                     IN THE FIRST SEGMENT OF A
*                                     MESSAGE.
          DC   CL27'PLEASE ENTER DATA SET NAME'
*                                     THIS IS THE MESSAGE TO BE
*                                     WRITTEN TO THE TERMINAL.
```

```
CHNMSG    DC   H'37'                  LENGTH OF THE SECOND LEVEL
*                                     MESSAGE TO BE PLACED ON AN
*                                     INTERNAL CHAIN. THIS LENGTH
*                                     INCLUDES THE FOUR BYTE HEADER.
          DC   H'0'                   THE OFFSET FIELD IS SET TO ZERO
*                                     IN THE FIRST SEGMENT OF A
*                                     MESSAGE.
          DC   CL33'MASTER PARTS CATALOG IS REQUIRED'
*                                     THIS IS THE MESSAGE TO BE
*                                     INTERNALLY CHAINED.
          IKJPGPB                     DSECT FOR THE PUTGET PARAMETER
*                                     BLOCK. IT EXPANDS WITH THE
*                                     SYMBOLIC NAME PGPB.
          IKJCPPL                     DSECT FOR THE COMMAND PROCESSOR
*                                     PARAMETER LIST.
          END
```

Chapter 10. Using the TGET/TPUT/TPG macro instructions for terminal I/O

This chapter describes how to use the TGET, TPUT and TPG macro instructions to process terminal I/O.

Overview of the TGET, TPUT and TPG macro instructions

You can use the TGET, TPUT, and TPG macro instructions in any programs that you write that run under TSO/E. However, when you use TGET, TPUT, or TPG in an application program, the program becomes TSO/E dependent. In a batch environment, the only TPUTs that are processed are those with the ASID, ASIDLOC, or USERIDL keyword referencing an ASID or user ID other than the current one. TGET, TPG, and other types of TPUT macros are ignored.

The TGET, TPUT, and TPG macro instructions do not require that you build control blocks for their use. The operands that you code on each of these macro instructions specify the location and size of the TGET, TPUT, or TPG buffers, and the functions you want performed.

The TGET and TPUT macro instructions have standard, list, execute, and register forms. The TPG macro has standard, list, and execute forms.

The sections that follow discuss the syntax of the TPUT, TPG and TGET macro instructions and the format of the parameters that you must pass.

See *z/OS TSO/E Programming Guide* for information about using the TGET and TPUT macro instructions in a full-screen Command Processor.

Using the TPUT macro instruction to Write a Line to the Terminal

Use the TPUT macro instruction to transmit a line of output to the terminal. You can use the TPUT macro instruction in any application programs to be run under TSO/E. Note, however, that TPUT does not provide message ID stripping, text insertion, or second-level message chaining. If you require these features, use the PUTLINE macro instruction which is described in Chapter 9, “Using the TSO/E I/O service routines for terminal I/O,” on page 173.

The TPUT macro instruction can be issued in 24-bit or 31-bit addressing mode. All input specified on the macro instruction must reside below 16 MB in virtual storage.

Figure 109 on page 258 shows the format of the TPUT macro instruction; the figure combines the standard, register, list, and execute forms. Each of the operands is explained following the figure.

Note: For a discussion of register contents and parameter list expansions for TPUT, see “Parameter Formats for TGET, TPUT, and TPG” on page 266.

```
  [symbol]      TPUT      [buffer address,buffer size                    ]
                          [,EDIT   ]                                     ]
                          [,NOEDIT ]                                     ]
                          [,ASIS   ][,WAIT  ][,NOHOLD]+
[,NOBREAK]         ]
                          [,CONTROL][,NOWAIT][,HOLD  ][,BREAKIN]         ]
                          [,FULLSCR]                                     ]

                          [,R                ][,HIGHP][,ASID=id        ] ]
                          [,MF={L           }][,LOWP ][,ASIDLOC=address] ]
                          [    {(E,ctrl addr)}]       [,USERIDL=address] ]

                          [ [TOKNIN=address]                             ]
```

Figure 109. The Standard, Register, List, and Execute Forms of the TPUT macro instruction

buffer address
Standard form: The address of the buffer that holds your line of output. You can specify any label that is valid in an RX instruction, or place the address of the label in one of the general registers 1–12, and then specify that register within parentheses.

Register form: The register that contains the parameters. When the R format is specified, this operand must be in one of the general registers 1–12, and that register must be specified within parentheses.

buffer size
Standard form: The size of the output buffer in bytes. The allowable range is 0-32767 bytes. A buffer size of 0 results in no data being transmitted to the terminal. You can specify this buffer size directly as a number, or you can place the buffer size into one of the general registers 0, or 2–12, and specify that register within parentheses.

Register form: The register that contains the parameters. When the R format is specified, this operand must be in one of the general registers 0 or 2–12, and that register must be specified within parentheses.

R
indicates that this is the register form of the TPUT macro instruction. You must place the parameters you want passed to TPUT into two registers and specify those registers as the first two operands of the macro instruction.

If the registers you specify as the first and second operands are registers 1 and 0 respectively, the TPUT macro instruction uses those registers. However, if you use registers 2–12, the macro expansion loads registers 1 and 0, respectively, from the registers you specify as the buffer address and buffer size. Therefore, you might find it advantageous to use registers 1 and 0. The expansion of the register form of the TPUT macro instruction destroys the contents of registers 1 and 0.

The R operand and all other optional operands are mutually exclusive. If both R and any other optional operands are coded, the macro will not expand.

MF=L | (E,*ctrl addr*)
indicates the form of the TPUT macro instruction.

L
specifies the list form.

(E,*ctrl addr*)
specifies the execute form and the address of the list form.

EDIT
indicates that in addition to minimal editing (see ASIS), the following TPUT functions are requested:

1. All trailing blanks are removed before the line is written to the terminal. If a blank line is sent, the terminal vertically spaces one line.

2. Control characters are added to the end of the output line to position the cursor to the beginning of the next line.
3. All terminal control characters, except backspace, are replaced with a printable character.
4. Only those characters that appear in USA EBCDIC keyboard layout and code charts are supported. All others are replaced with a printable character. The replacement of characters includes the representation of the keyboard features and the special characters $, #, @ without hexadecimal equivalents of the USA EBCDIC code. For more information about keyboard features character sets, see *IBM 3270 Information Display System: Character Set Reference*.

EDIT is the default value for the EDIT, ASIS, CONTROL, FULLSCR, and NOEDIT operands.

NOEDIT
indicates that, if the terminal is an IBM 3270 display, the message is transmitted completely unedited. It is assumed that a Command Processor that uses this full-screen option has structured the data stream with the necessary commands to perform the display function. For LU_T1 terminals, this option is converted to ASIS.

TSO/VTAM supports 3270 extended data stream functions with the TPUT NOEDIT and NOEDIT modes of input. For information about specifying the NOEDIT mode of input, refer to "STFSMODE - Set Full-Screen Mode" on page 154.

ASIS
indicates that minimal editing is to be performed by TPUT as follows:

1. The line of output is translated from EBCDIC to terminal code. Not valid characters are converted to a printable character to prevent program caused I/O errors. This does not mean that all unprintable characters are eliminated. For example, restore, uppercase, lower case, bypass, and bell ring might be valid but unprintable characters at some terminals. (See CONTROL.)
2. Transmission control characters are added.
3. An EBCDIC NL, placed at the end of the message, indicates to TPUT that the cursor is to be returned at the end of the line. NL is replaced with whatever is necessary to cause the cursor to return for that particular terminal type. This NL processing occurs only if you specify ASIS, and if the NL is the last character in your message.

   If you specify EDIT, NL is handled as described in item 3 under EDIT.

   If the NL is embedded in your message, a semicolon or colon may be substituted for NL and sent to the terminal. No idle characters are added (see item 6 below). This can cause overprinting, particularly on terminals that require a line-feed character to position the carrier on a new line.
4. If you have used backspace in your output message, but the backspace character does not exist on the terminal type to which the message is being routed, the backspace character is removed from the output message.
5. If the output line is longer than the terminal line size, control characters are added as needed to cause the message to display on several lines.
6. A sufficient number of idle characters is added to the end of each output line to prevent the transmission of output to the terminal while the cursor is being returned to the left-hand margin.
7. Including a bypass character, bypass carriage return, or bypass new-line character in the TPUT macro data suppresses printing of the next input entered by the user at the 3270 terminal. VTAM moves the cursor to the next available line, unlocking the keyboard. No more data is sent to the terminal until the terminal user enters data or presses the Enter key. The data entered by the user is not printed at the terminal.

CONTROL
indicates that this line is composed of terminal control characters and does not display or move the cursor on the terminal. This option should be used for transmission of characters such as bypass, restore, or bell ring. See item 7 under ASIS for additional information.

FULLSCR
indicates that, for IBM 3270 display terminals, the message will be transmitted essentially unedited. The FULLSCR option is designed to allow you to use special features of the 3270 system. For any

other terminal type, this option is treated exactly as ASIS. With the FULLSCR option, only the following editing is performed:

1. If the first character in your message is an escape control character (X'27'), the two characters following it are treated as a command code and as a write control character by the 3270. Note that the command code should always be for a remote 3270. If necessary, TPUT will convert the code to that for a local 3270. If the first character is not an escape character, a default write command and a write control character are added to the beginning of the message. Any attachment-dependent characters required for correct transmission of the data stream are provided by the access method.
2. Transmission control characters (SOH, STX, ETX, ETB, EOT, and NAK) and characters having no 3270 equivalent (X'04', X'06', X'14' through X'17', and X'24') are converted to printable colons to prevent program-caused I/O errors.

Lines are not counted when you use this option.

If the OWAITHI value specified in your TSO/E parameters is not large enough to contain your entire message, or if the BUFFERS and BUFFERSIZE parameters are specified so that your message does not fit into all of the system's buffers, the TPUT operation does not proceed, and code X'10' is returned. For a description of OWAITHI, see *z/OS MVS Initialization and Tuning Reference*. Without the FULLSCR option, your TPUT proceeds buffer-by-buffer as buffers become available.

If FULLSCR is specified for a message destined for another terminal, ASIS will be used instead.

WAIT | NOWAIT

WAIT
specifies that control is not returned to the program that issues the TPUT macro instruction until the output line is placed into a terminal output buffer. If no buffers are available for the same ASID TPUT (TPUT without any ASID, ASIDLOC, or USERIDL option - not a cross-memory TPUT), the issuing program is placed into a wait state until buffers become available, and the output line is placed into them. WAIT is the default value for the WAIT and NOWAIT operands.

Note: A cross-memory TPUT with WAIT operand will be rejected with a return code of 20 (X'14') when the buffers are not available. A cross-memory (ASID) TPUT is allowed to exceed the storage value coded for the high buffer threshold. The limit is calculated as:

```
high buffer threshold * 3
```

where high buffer threshold is specified by parameter HIBFREXT in the member TSOKEY00 of SYS1.PARMLIB.

NOWAIT
specifies that control is returned to the program that issues the TPUT macro instruction, whether or not a terminal output buffer is available for the output line. If no buffer is available, TPUT returns a code of 4 in register 15.

NOHOLD | HOLD

NOHOLD
indicates that control is returned to the program that issues the TPUT macro instruction as soon as the output line is placed in terminal output buffers.

NOHOLD is the default value for the NOHOLD and HOLD operands.

HOLD
specifies that the program that issues the TPUT macro instruction cannot continue its processing until this output line is written to the terminal or deleted. The TPUT macro with the HOLD option is not discarded during RESHOW processing.

NOBREAK | BREAKIN

NOBREAK
specifies that if the user starts to enter input, the user is not interrupted. The output message is placed on the output queue and displayed after the user completes the line.

NOBREAK is the default value for the NOBREAK and BREAKIN operands.

BREAKIN
specifies that output has precedence over input. If the user starts to enter input, input is interrupted, and this output line is displayed. Data received before the interruption is displayed following this output line. However, the amount of data that is displayed is unpredictable.

HIGHP | LOWP

HIGHP
specifies that this message must be sent to the terminal, even though the destination terminal does not display messages from other terminals. This operand counters the effect of the interterminal communication bit when the bit is set by the PROFILE command.

The operand is recognized only if your program is authorized (either by system key, supervisor state, or APF). The ASID keyword must also be specified. HIGHP is the default if neither HIGHP nor LOWP is specified, and if the issuing program is authorized.

LOWP
specifies that, if the user of the destination terminal allows interterminal messages, this TPUT will be sent to the terminal. If such messages are not allowed, the message is not displayed, and a code of X'0C' is returned, indicating that the message was not displayed. The LOWP operand is recognized only when ASID is specified. To use this operand, your program must be authorized (either by system key, supervisor state, or APF).

If you specify LOWP, your program should have an alternate method of transmitting the message to the terminal user. For example, a message data set could be used.

ASID | ASIDLOC | USERIDL
specifies the ASID (address space identifier) of the target terminal, the address of that ASID, or the address of a field that contains a user ID. If you specify ASID, you must supply an ASID number. If you use ASIDLOC, you must supply the address of the halfword that contains the ASID. If you use USERIDL, you must supply the address of the 8-byte field that contains the user ID. The user ID must be left-justified and, if necessary, padded with blanks. ASID, ASIDLOC, or USERIDL can be specified in a register (2–12), and must be right-justified. The register number must be enclosed in parentheses. If USERIDL is used, the NOHOLD option is both required and the default if not specified.

ASID, ASIDLOC, and USERIDL are not valid when you specify them with FULLSCR or ASIS parameters.

Note: Normally, a program invokes TPUT to issue a message to the user running that program; that is, ASID, ASIDLOC, and USERIDL are not specified. If that program is run in the background, the TPUT has no effect.

If the TPUT specifies an ASID or user ID, the message is sent to the target terminal. ASID and USERID TPUTs from programs not in supervisor state or not authorized under APF are prefixed with a plus sign (+) to prevent possible counterfeiting of system messages to an operator console.

TOKNIN
specifies the address of a security token to be used by VTAM. The address can be specified as either a register (such as (R4)), or as a label (with no parentheses) that contains the address of the user token.

The operand is recognized only if your program is authorized (either by system key 0 - 7, supervisor state, or APF-authorized).

You may specify the TOKNIN operand only with the execute form of the TPUT macro.

If you specify TOKNIN= with no value, the entire TOKNIN operand is ignored.

Return Codes from TPUT

When TPUT returns control to the program that invoked it, in either the foreground or the background, TPUT supplies one of the following return codes in general register 15:

Table 69. Return codes from TPUT

Return code dec(Hex)	Meaning
0(0)	TPUT completed successfully.
4(4)	NOWAIT was specified and no terminal output buffer was available.
8(8)	An attention interruption occurred while TPUT was processing. The message was not sent.
12(C)	A TPUT macro instruction with an ASID operand was issued but the user, indicated by the ASID, requested that interterminal messages not be printed on the terminal. The message was not sent.
16(10)	Incorrect parameters were passed to TPUT.
20(14)	The terminal was logged off and could not be reached. Cross-memory TPUT could not get buffer or a serious error has occurred in z/OSMF ISPF.
24(18)	The sender is not permitted to send a message to the intended user.
28(1C)	The intended receiver of the message is logged on at a security label too low to receive the message.
32(20)	No storage is available.
36(24)	JESXCF at remote side is downlevel.
40(28)	JESXCF at local side is downlevel.
44(32)	JESXCF function call failed.

For TPUT FULLSCR or NOEDIT with the parameter list, register 0 is supplied with the output buffer number. It also sets the indicator of the output buffer number that is present in register 0 of the parameter list. The output buffer number is 0 through 65535. When it reaches 65535, it is reset to zero.

Using the TPG macro instruction to Write a Line Causing Immediate Response

Use the TPG macro instruction to transmit a line of output to the terminal if that line of output will cause the device to respond immediately with input. You can use the TPG macro instruction in any application programs that you write to run under TSO/E. If a TPG macro is coded in a background program, the TPG is ignored.

The TPG macro is designed for use on any terminal type that supports the Query function. The main use of TPG is to perform the Query function for a user who has included a Read Partition Structured field. TPG NOEDIT creates an outbound request unit with an associated change direction indicator to allow the device to go into send state. This data is not inspected. A TGET macro must be issued to retrieve the query response.

The TPG macro instruction can be invoked in either 24-bit or 31-bit addressing mode. All input specified on the macro must reside below 16 MB in virtual storage.

Figure 110 on page 263 shows the standard, list and execute forms of the TPG macro instruction. The register format cannot be used for the TPG macro. Each of the operands is explained following the figure. For a discussion of parameter list expansions for TPG, see “Parameter Formats for TGET, TPUT, and TPG” on page 266.

```
[symbol]      TPG         buffer address,buffer size
                       [[,NOEDIT] [,WAIT  ][,NOHOLD ]]
                       [          [,NOWAIT][,HOLD   ]]
                       [[,MF={L           }          ]
                       [[    {(E,ctrl addr)}         ]
```

Figure 110. The Standard, List, and Execute Forms of the TPG macro instruction

buffer address
Standard form: The address of the buffer that holds your output data. You can specify any address valid in an RX instruction, or place the address in one of the general registers 1–12, and then specify that register within parentheses.

buffer size
Standard form: The size of the output buffer in bytes. The allowable range is 0-32767 bytes. A buffer size of 0 results in no data being transmitted to the terminal. You can specify this buffer size directly as a number, or you can place the buffer size into one of the general registers 0 or 2–12 and specify that register within parentheses.

NOEDIT
indicates that, if the terminal is an IBM 3270 display, the message is transmitted completely unedited. If your Command Processor uses this option, you must structure the data stream with the necessary commands to perform the display function (by including the command, write control character, structured fields,...). The Command Processor should supply only the data stream. Any attachment-dependent characters (such as X'27' for bisynchronous devices) are provided by the access method. For LU_T1 terminals, this option is treated exactly like the ASIS option of the TPUT macro.

Note: NOEDIT is the default, and is the only mode for the TPG macro. If NOEDIT is omitted, a comma must be used.

WAIT | NOWAIT

WAIT
specifies that control is not returned to the program that issued the TPG macro instruction until the output line is placed into a terminal output buffer. If no buffers are available, the issuing program is placed into a wait state until buffers become available, and the output line is placed into them. WAIT is the default value for the WAIT and NOWAIT operands.

NOWAIT
specifies that control is returned to the program that issued the TPG macro instruction, whether or not a terminal output buffer is available for the output line. If no buffer is available, TPG returns a code of 4 in register 15.

NOHOLD | HOLD

NOHOLD
indicates that control is returned to the program that issued the TPG macro instruction as soon as the output line is placed in terminal output buffers.

NOHOLD is the default value for the NOHOLD and HOLD operands.

HOLD
specifies that the program that issued the TPG macro instruction cannot continue its processing until this output line is written to the terminal or deleted.

MF=L | (E,*ctrl addr*)
indicates the form of the TPG macro instruction.

L
specifies the list form.

(E,*ctrl addr*)
specifies the execute form and the address of the list form.

Return Codes from TPG

When TPG returns control to the program that invoked it, either in the foreground or in the background, TPG supplies one of the following return codes in general register 15:

Table 70. Return codes from TPG

Return code dec(Hex)	Meaning
0(0)	TPG completed successfully.
4(4)	NOWAIT was specified and no terminal output buffer was available.
8(8)	An attention interruption occurred while TPG was processing. The message was not sent.
16(10)	Incorrect parameters were passed to TPG.
20(14)	The terminal was logged off and could not be reached or a serious error has occurred in z/OSMF ISPF.

For TPG with the parameter list, register 0 is supplied with the output buffer number. It also sets the indicator of the output buffer number that is present in register 0 of the parameter list. The output buffer number is 0 through 65535. When it reaches 65535, it is reset to zero.

Using the TGET macro instruction to Get a Line from the Terminal

Use the TGET macro instruction to read a line of input from the terminal. A line of input is defined as all the data between the beginning of the input line and a line-end delimiter. A line-end delimiter is any character or combination of characters that causes the cursor to return to the left-hand margin on a new line, or that terminates transmission from the terminal.

You can use the TGET macro instruction in any application program that is run under TSO/E. Note, however, that TGET does not provide access to in-storage lists, nor does it perform any type of logical line processing on the returned line. If you require these features, use the GETLINE macro instruction, which is discussed in Chapter 9, “Using the TSO/E I/O service routines for terminal I/O,” on page 173.

Each time TGET returns control to your program, register 1 contains the number of bytes of data actually moved from the terminal to your input buffer. If your buffer is smaller than the line of input entered at the terminal, only as much of the input line as can be contained in the input buffer is moved. Return code X'0C' indicates that only part of the line was obtained by TGET. You must then issue as many TGET macro instructions as are required to get the rest of the line of input.

In the full-screen environment, when TGET returns with one byte of data with a RESHOW character to the command processor, the RESHOW indicator tells the command processor to completely restore the screen contents. That is, the command processor must reissue the previous full-screen message. The default RESHOW character is X'6E'. The command processor can specify the RESHOW character in the STFSMODE macro. See *z/OS TSO/E Programming Guide* for additional information under the topic "RESHOW in Full-Screen Message Processing and Restoration of Screen Captures".

The TGET macro instruction can be invoked in 24-bit or 31-bit addressing mode. All input specified on the macro must reside below 16 MB in virtual storage.

Figure 111 on page 265 shows the format of the TGET macro instruction; it combines the standard, register, and list forms. Each of the operands is explained following the figure. For a discussion of register contents and parameter list expansions for TGET, see “Parameter Formats for TGET, TPUT, and TPG” on page 266.

```
                                                [[,EDIT] [,WAIT  ]    ]
[symbol]      TGET       buffer address,buffer size[[,ASIS] [,NOWAIT]    ]
                                                [[,R                  ]
                                                [[,MF={L           }  ]
                                                [[    {(E,ctrl addr)} ]
```

Figure 111. The Standard, Register, List, and Execute Forms of the TGET macro instruction

buffer address
Standard form: The address of the buffer that is to receive the input line. This can be any address valid in an RX instruction, or the address can be placed in one of the general registers 1–12, and that register specified within parentheses.

Register form: The register that contains the parameters. When the R format is specified, this operand must be in one of the general registers 1–12, and that register must be specified within parentheses.

buffer size
Standard form: The size of the input buffer in bytes. The allowable range is 0-32767 bytes. You can specify this buffer size directly as a number, or you can place the buffer size into one of the general registers 0, or 2–12, and specify that register within parentheses. A TGET with a 0-length buffer size will successfully get a null line.

Register form: The register that contains the parameters. When the R format is specified, this operand must be in one of the general registers 0 or 2–12, and that register must be specified within parentheses.

R
indicates that this is the register form of the TGET macro instruction. You must place the parameters you want passed to TGET into two registers and specify those registers as the first two operands of the macro instruction.

Note: If the registers you specify as the first and second operands are registers 1 and 0 respectively, the TGET macro instruction uses those registers. However, if you use registers 2–12, the macro expansion loads registers 1 and 0, respectively, from the registers you specify as the buffer address and buffer size. Therefore, you might find it advantageous to use registers 1 and 0.

The R operand and all other optional operands are mutually exclusive. If both R and any other optional operands are coded, the macro will not expand.

EDIT
specifies that in addition to minimal editing (see ASIS), the following TGET functions are requested:

1. All terminal control characters (nongraphic characters such as bypass, line feed, restore, prefix and the character immediately following it) are removed from the data.
2. When backspace is not used for character deletion, the horizontal tab (HT) and the backspace (BS) characters remain in the data.
3. If the returned input line is shorter than the input buffer length, the buffer is padded with blanks. These blanks are not included in the character count returned in register 1.

EDIT is the default value for the EDIT and ASIS operands.

ASIS
specifies that minimal editing is done as described below:

1. Transmission control characters are removed.
2. The returned input line is translated from terminal code to EBCDIC. Not valid characters are compressed out of the data.
3. Line deletion and character deletion are performed according to the specifications in the terminal status block.

4. New line (NL), cursor return (CR), and line feed (LF) characters, if present at the end of the line, are not included in the data count returned in register 1.
5. After the input message is received, the cursor is returned to the left-hand margin of the next line before any output to the terminal is displayed.

WAIT | NOWAIT

WAIT
specifies that control is not returned to the program that issues the TGET macro instruction until the input line is placed into your input buffer. If an input line is not available from the terminal, the issuing program is placed into a wait state until a line becomes available and is read into your input buffer. WAIT is the default value for the WAIT and NOWAIT operands.

NOWAIT
specifies that, whether or not an input line is available from the terminal, control is returned to the program that issues the TGET macro instruction. If no line is returned, TGET returns a code of X'04' in register 15.

MF=L | (E,*ctrl addr*)
indicates the form of the TGET macro instruction.

L
specifies the list form.

(E,*ctrl addr*)
specifies the execute form and the address of the list form.

Return Codes from TGET

When TGET returns control to the program that invoked it, TGET supplies, in register 1, the length of the message moved into your buffer. In addition, one of the following return codes is supplied in register 15:

Table 71. Return codes from TGET

Return code dec(Hex)	Meaning
0(0)	TGET completed successfully. Register 1 contains the length of the input line read into your input buffer.
4(4)	NOWAIT was specified and no input was available to be read into your input buffer.
8(8)	An attention interruption occurred while TGET was processing. The message was not received.
12(C)	Your input buffer was not large enough to accept the entire line of input entered at the terminal. Subsequent TGET macro instructions will obtain the rest of the input line.
16(10)	Incorrect parameters were passed to TGET.
20(14)	The terminal was logged off and could not be reached or a serious error has occurred in z/OSMF ISPF.
24(18)	TGET completed successfully. Register 1 contains the length of the input line read into your buffer. The data was received in NOEDIT mode.
28(1C)	Your input buffer was not large enough to accept the entire line of input entered at the terminal. Subsequent TGET macro instructions will obtain the rest of the input line. The data was received in NOEDIT mode.

Parameter Formats for TGET, TPUT, and TPG

Register Form of TGET and TPUT

If you use the register form of the TGET or TPUT macro instruction, you must code the parameters into two registers. Specify these two registers, enclosed in parentheses, as the first two operands of the TGET or TPUT macro instruction, followed by the R operand to indicate that you are executing the register form of the macro instruction.

If the registers you specify as the first and second operands of the macro instruction are register 1 and register 0 respectively, the TGET or TPUT macro instruction uses those registers. However, if you specify registers 2–12, the macro expansion loads registers one and zero, respectively, from the registers you specify. For TPUT, the expansion destroys the contents of registers 0 and 1. The R format cannot be used for the TPG macro.

For the TPUT macro, you must format the registers as shown in Figure 112 on page 267.

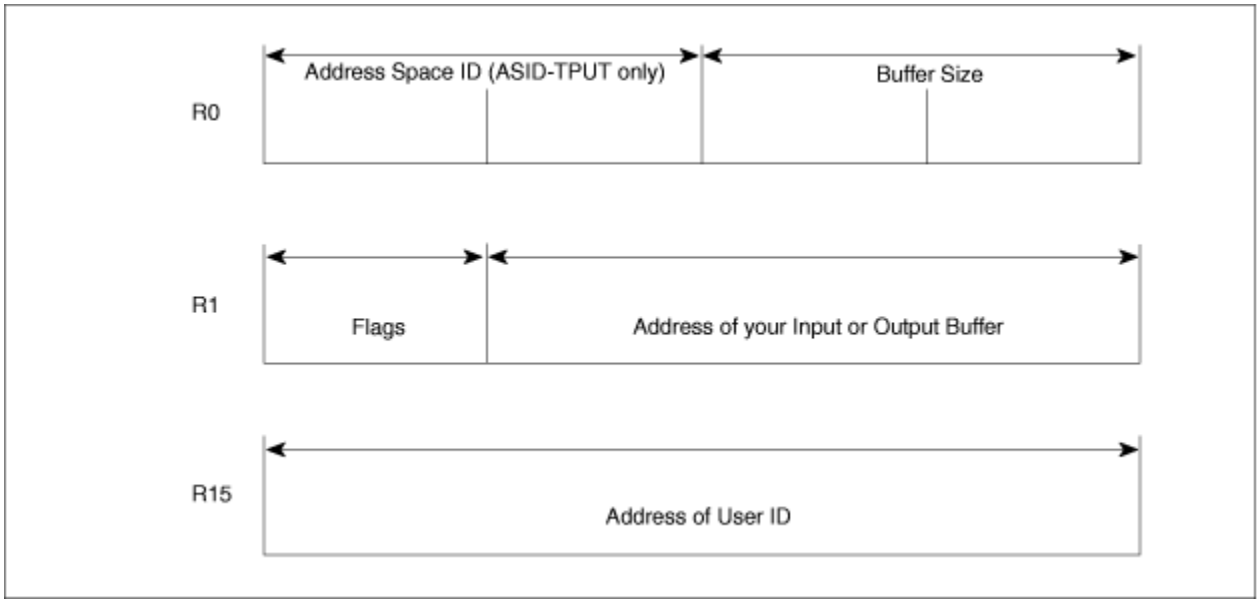


Figure 112. TPUT Parameter Registers

For the TGET macro, you must format the registers as shown in Figure 113 on page 267.

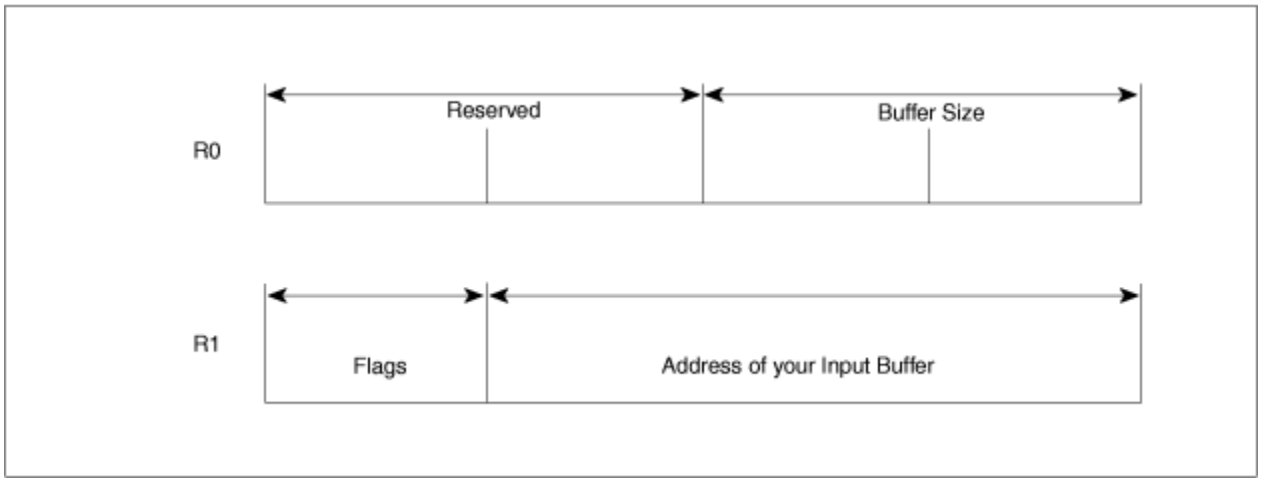


Figure 113. TGET Parameter Registers

For both TPUT and TGET, the high-order byte of register 1 contains flags that indicate what type of processing you want performed. Table 72 on page 268 shows the meanings of these flags.

Table 72. Option flags contained in register 1

Setting	Meaning
`0... ....`	Always set to 0 for TPUT.
`1... ....`	Always set to 1 for TGET.
`.0.. ....`	No user ID.
`.1.. ....`	Register 15 contains address of user ID.
`..0. ....`	HIGHP processing is requested.
`..1. ....`	LOWP processing is requested.
`...0 ....`	WAIT processing is requested.
`...1 ....`	NOWAIT processing is requested.
`.... 0...`	NOHOLD processing is requested.
`.... 1...`	HOLD processing is requested.
`.... .0..`	NOBREAK processing is requested.
`.... .1..`	BREAKIN processing is requested.
`.... ..00`	EDIT processing is requested.
`.... ..01`	ASIS processing is requested.
`.... ..10`	CONTROL processing is requested.
`.... ..11`	FULLSCR processing is requested.

Execute, Standard and List Forms of TPUT

If you use the execute form of the TPUT macro, the coded parameters expand into the parameter list shown in Figure 114 on page 269.

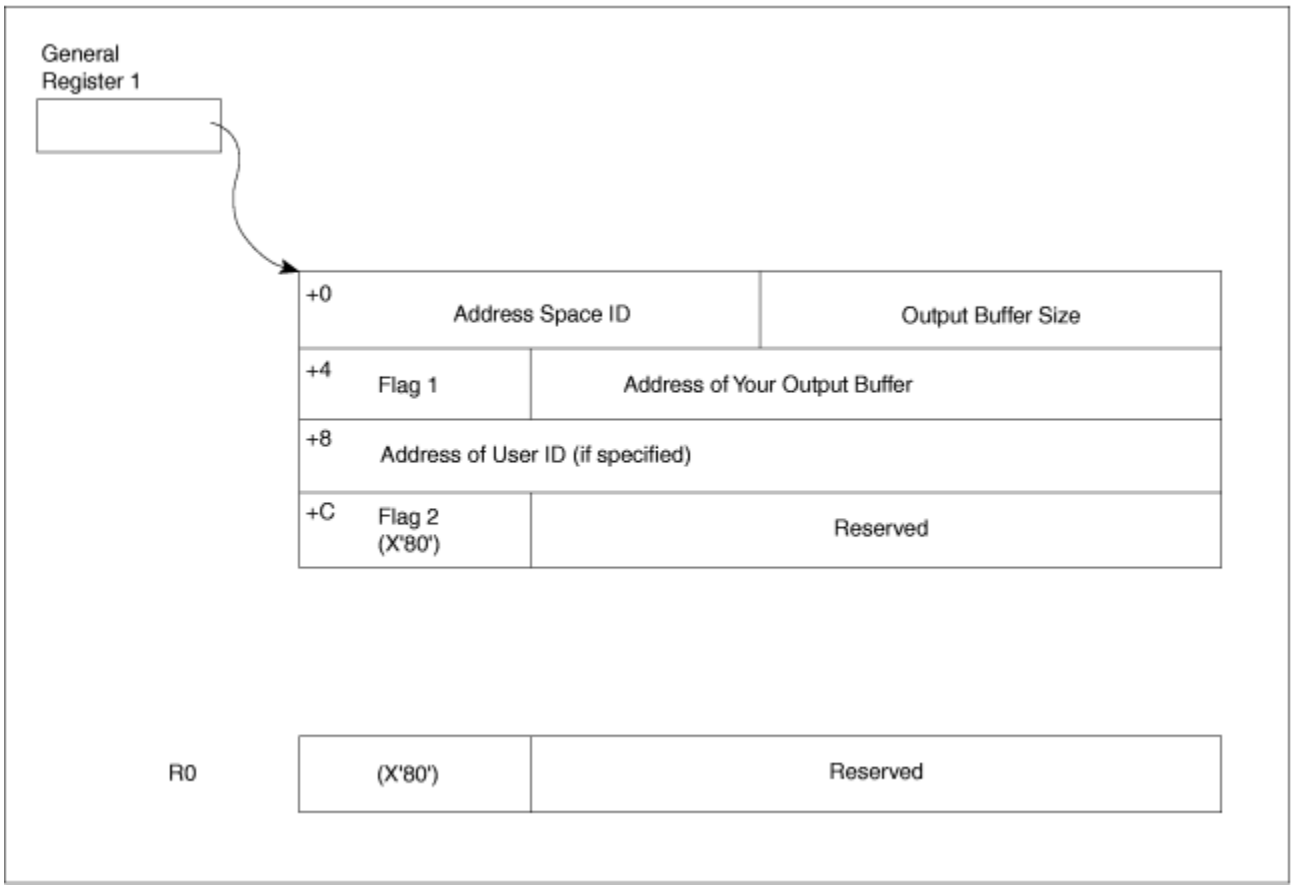


Figure 114. Parameter List Expansion for the Execute Form of TPUT

The possible settings of Flag1 in the parameter list expansion for the execute form of TPUT are the same as those for the high-order byte of register 1 in the register form. Table 72 on page 268 describes the meanings of these flags.

If you use the standard form of the TPUT macro, you can code your parameters using registers or symbols. In this case, the TPUT macro expands to load the parameters into registers 0, 1, and 15 in the format illustrated in Figure 112 on page 267.

If you use the list form of the TPUT macro, the coded parameters expand into the parameter list shown in Figure 115 on page 269.

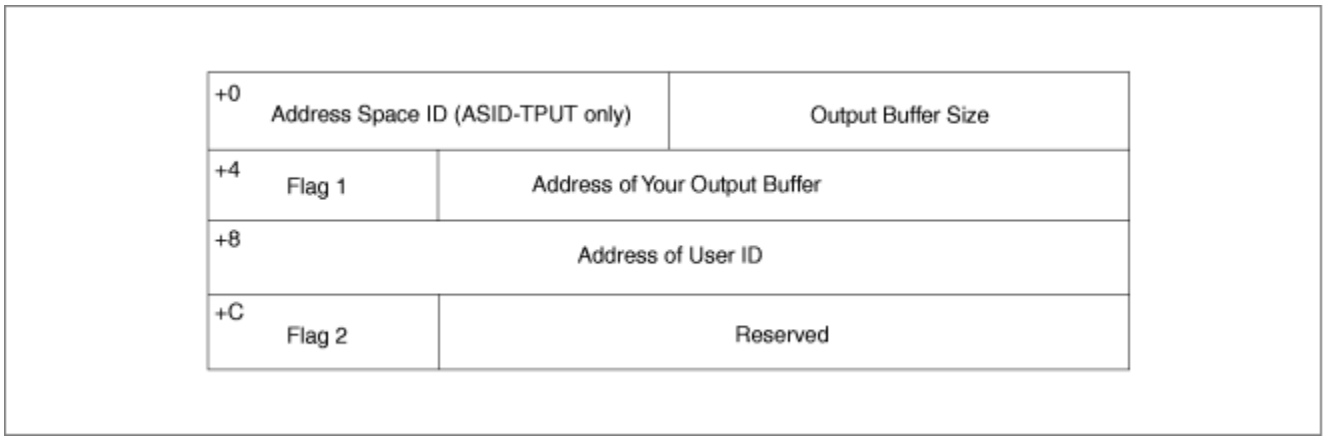


Figure 115. Parameter List Expansion for the List Form of TPUT

The value of Flag2 in the parameter list expansion for the list form of TPUT is X'01', if the NOEDIT option is specified.

The value of Flag2 in the parameter list upon return from TPUT SVC is X'40', if the FULLSCR or NOEDIT is specified. This indicates that register 0 is supplied with the output buffer number.

Execute and List Forms of TPG

If you use the execute form of the TPG macro, the coded parameters expand into the parameter list shown in Figure 116 on page 270.

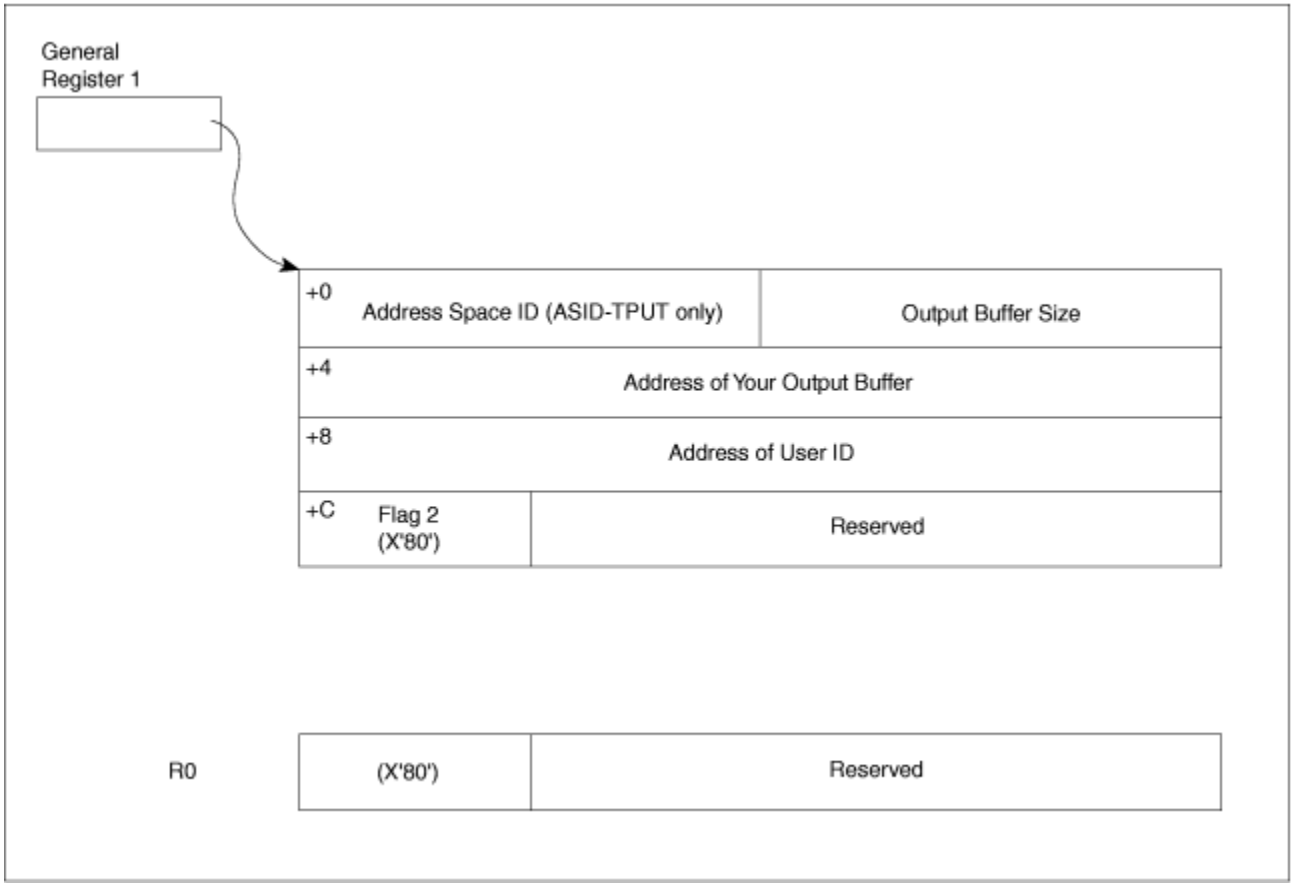


Figure 116. Parameter List Expansion for the Execute Form of TPG

If you use the list form of the TPG macro, the coded parameters expand into the parameter list shown in Figure 117 on page 271.

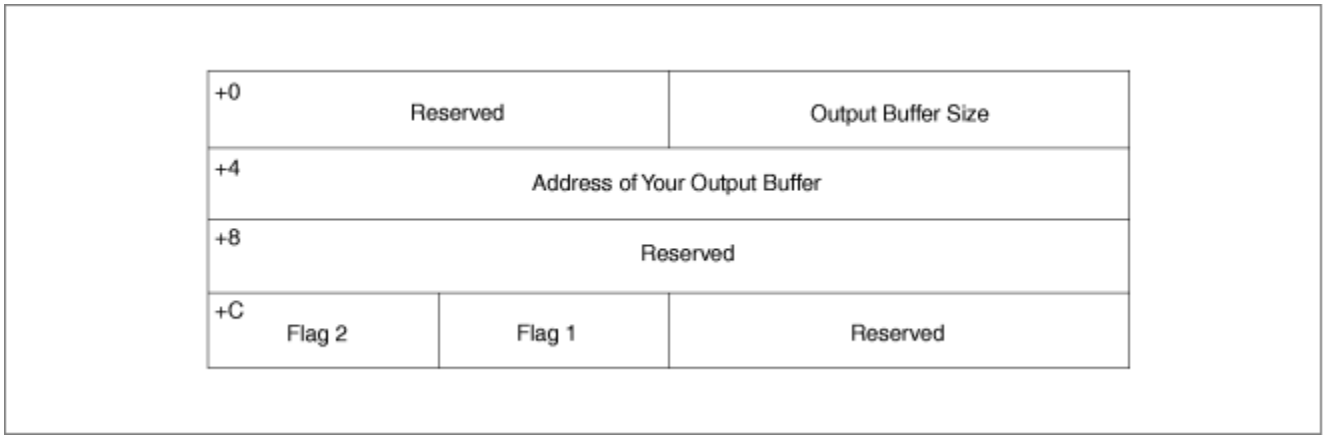


Figure 117. Parameter List Expansion for the List Form of TPG

For Figure 117 on page 271, the possible settings of Flag1 are shown in Table 72 on page 268. Flag2 is X'01' for the NOEDIT option and X'02' for the TPG macro.

The value of Flag2 in the parameter list upon return from TPUT SVC is X'40'. This indicates that register 0 is supplied with the output buffer number.

Standard, List and Execute Forms of TGET

If you use the standard, list, or execute form of the TGET macro, the coded parameters expand into the parameter list shown in Figure 118 on page 271.

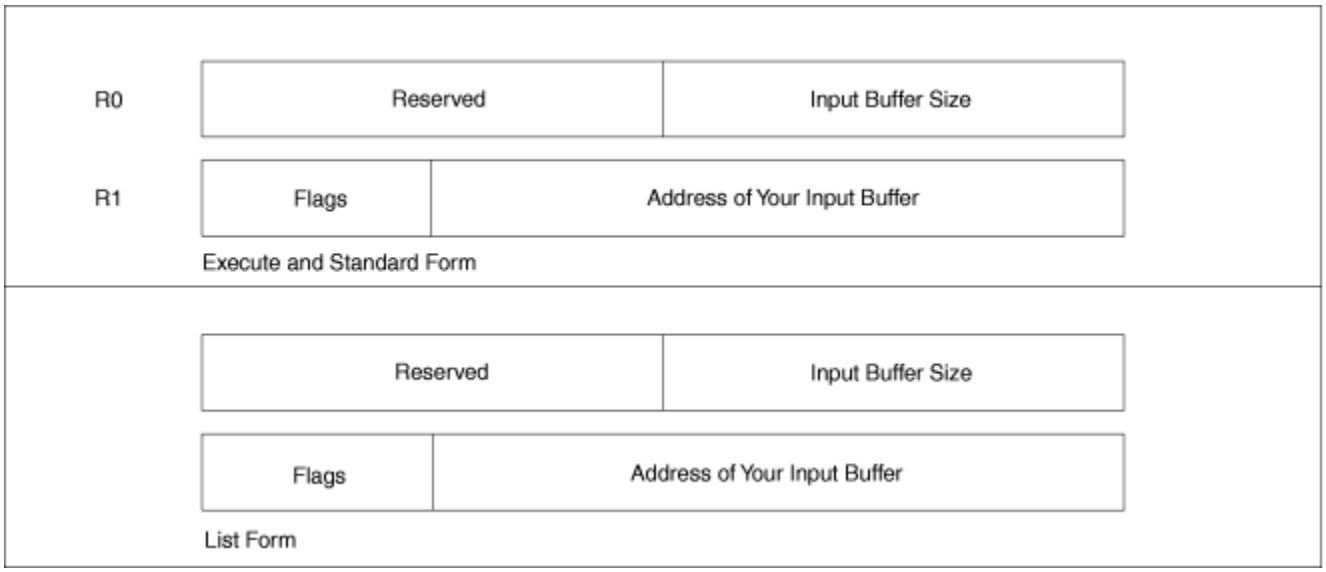


Figure 118. Parameter List Expansion for the Standard, List, and Execute Forms of TGET

In Figure 118 on page 271, the possible settings of Flags are the same as those for the high-order byte of register 1 in the register form. Table 72 on page 268 describes the meanings of these flags.

Examples Using the TGET and TPUT macro instructions

The following coding examples show different ways to use the TGET and TPUT macro instructions.

Example 1: Using the Default Values for TPUT and TGET

Figure 119 on page 272 shows a TPUT and a TGET macro instruction. They both use the default values; that is, the TPUT macro instruction defaults to EDIT, WAIT, NOHOLD, and NOBREAK, and the TGET macro instruction defaults to EDIT and WAIT.

```
*
*      PROCESSING
*
* USE THE TPUT MACRO INSTRUCTION TO WRITE A MESSAGE TO THE TERMINAL.
* USE THE DEFAULT VALUES.
*
         TPUT  MESSAGE1,24          THE BUFFER ADDRESS IS THE SYMBOLIC
*                                   ADDRESS MESSAGE1, AND THE BUFFER
*                                   LENGTH IS 24 BYTES.
         LTR   15,15                TEST RETURN CODE - ZERO INDICATES
*                                   SUCCESSFUL COMPLETION.
         BNZ   ERRTN                IF THE RETURN CODE IS NOT ZERO,
*                                   GO TO AN ERROR ROUTINE.
* USE THE TGET MACRO INSTRUCTION TO OBTAIN AN INPUT LINE FROM THE
* TERMINAL.  TAKE THE DEFAULT VALUES.
*
         TGET  BUFFER,130           THE BUFFER ADDRESS IS THE SYMBOLIC
*                                   ADDRESS, BUFFER, AND THE INPUT
*                                   BUFFER LENGTH IS 130 BYTES.
         LTR   15,15                TEST THE RETURN CODE - ZERO
*                                   INDICATES SUCCESSFUL COMPLETION.
         BNZ   ERRTN                IF THE RETURN CODE IS NOT ZERO,
*                                   BRANCH TO AN ERROR ROUTINE.
*
*      PROCESSING
*
ERRTN     ERROR ROUTINE PROCESSING
*          .
*          .
*          .
* STORAGE DECLARATIONS
*
         DS    0F
MESSAGE1 DC    CL24'THIS IS A TPUT MESSAGE. '
BUFFER   DS    CL130
         END
```

Figure 119. Example 1: TPUT and TGET macro instructions Using the Default Values

The program issuing the TGET macro instruction is not given control until a line of data is returned. The default value is WAIT. If less than 130 characters are entered, the input buffer is padded with blanks. The default is EDIT. Remember that the actual length of the data in the input buffer is returned in register 1.

Example 2: Using TPUT with Buffer Address and Buffer Length in Registers

In the coding example shown in Figure 120 on page 273, the output message buffer address and length are loaded into registers, and those registers are coded as operands in the TPUT macro instruction.

You might want to do this when, for example, the TPUT macro instruction is issued in a subroutine which receives, as parameters, a pointer to the message and the message length.

```
*
*    PROCESSING
*
* PLACE THE BUFFER ADDRESS AND THE BUFFER LENGTH INTO REGISTERS.
*
         LA    0,L'MESSAGE1          LOAD THE BUFFER LENGTH INTO
*                                    REGISTER ZERO. THE LOAD ADDRESS
*                                    INSTRUCTION INSURES THAT THE HIGH
*                                    ORDER BYTE IS ZEROED IN THE
*                                    REGISTER.
         LA    1,MESSAGE1            LOAD ADDRESS OF THE OUTPUT
*                                    BUFFER INTO REGISTER 1.
* ISSUE THE TPUT MACRO INSTRUCTION.
*
       TPUT    (1),(0)
*
       LTR     15,15                 TEST THE RETURN CODE - ZERO
*                                    INDICATES SUCCESSFUL COMPLETION.
       BNZ     ERRTN                 IF THE RETURN CODE IS NOT ZERO,
*                                    GO TO AN ERROR ROUTINE.
*    PROCESSING
*
ERRTN   ERROR PROCESSING
*         .
*         .
*         .
* STORAGE DECLARATIONS
*
         DS    0F
MESSAGE1 DC    C'THIS IS A TPUT MESSAGE.'
*
         END
```

Figure 120. Example 2: TPUT macro instruction with Buffer Address and Buffer Length in Registers

Example 3: Using the Register Format of TGET

Figure 121 on page 274 shows the code necessary to issue a register format TGET macro instruction. The buffer length, buffer address, and the option flags are loaded into registers zero and one. Note that the flag byte in register one is set to binary B'10000001', indicating that this is a TGET macro instruction requesting ASIS processing. This means that only minimal editing is performed on the input line.

```
GETFLGS  EQU   B'10000001'
*
*    PROCESSING
*
* PLACE THE BUFFER SIZE AND THE BUFFER ADDRESS INTO REGISTERS 0 AND 1.
*
         LA    0,L'BUFFER            LOAD THE BUFFER SIZE INTO
*                                    REGISTER ZERO.
         LA    1,BUFFER              LOAD BUFFER ADDRESS INTO
*                                    REGISTER 1.
         LA    4,GETFLGS             THIS WILL BE THE HIGH-ORDER
*                                    BYTE OF REGISTER 1.
         SLL   4,24                  SHIFT THE FLAGS TO THE HIGH-
*                                    ORDER BYTE
         OR    1,4                   MERGE FLAG BYTE INTO REGISTER 1.
*
* ISSUE THE TGET MACRO INSTRUCTION SPECIFYING REGISTER FORMAT (R).
*
         TGET  (1),(0),R
*
         LTR   15,15                 TEST RETURN CODE. IF NOT ZERO,
         BNZ   ERRTN                 GO TO AN ERROR ROUTINE.
*
*    PROCESSING
*
ERRTN   ERROR PROCESSING
*         .
*         .
*         .
* STORAGE DECLARATIONS
*
BUFFER   DS    CL130                 INPUT BUFFER
*
         END
```

Figure 121. Example 3: TGET macro instruction Register Format

Chapter 11. Using the TSO/E message handling routine IKJEFF02

This chapter describes how to use the TSO/E message handling routine (IKJEFF02) in a Command Processor to issue messages.

Overview of Message Handling

There are three types of TSO/E messages:

- Prompting messages
- Mode messages
- Informational messages

Prompting messages begin with ENTER or REENTER, and require a response from the user.

Mode messages are the READY messages sent by the terminal monitor program, and any other similar messages sent by command processors, such as the EDIT mode message sent by the EDIT Command Processor. They inform the user which command is in control and let the user know that the system is waiting for the user to enter a new command or subcommand.

Informational messages do not require an immediate response from the user.

Messages should usually have other messages associated with them that more fully explain the initial message. These messages, called second-level messages, are displayed only if the user specifically requests them by entering a question mark (?) in response to the initial message. Prompting messages can have any number of second-level messages. An informational message can have only one second-level message associated with it. Mode messages cannot have second-level messages.

TSO/E Message Issuer Routine (IKJEFF02)

The TSO/E message issuer routine issues a message using PUTLINE, PUTGET, write-to-operator (WTO), or write-to-programmer (WTP). You can indicate to IKJEFF02 which of these services should be used to issue the message, or you can allow the default, PUTGET, to be used. For prompting and mode messages, you should indicate to IKJEFF02 that PUTGET should be used to issue the message; for informational messages, PUTLINE should be used. If you want to issue the message in the language specified in the user profile table (UPT), you must indicate to IKJEFF02 that PUTGET or PUTLINE should be used.

For more information about providing translated messages, see “PUTLINE Message Line Processing” on page 224.

You can invoke IKJEFF02 just to issue the message to the terminal, both to issue the message and return the requested message to the caller in the caller's buffers, or just to return the message to the caller. This process of returning the message is referred to as extracting the message.

The TSO/E message issuer routine simplifies the issuing of messages with inserts because hexadecimal inserts can be converted to printable characters and the same parameter list can be used to issue any message. It also makes it more convenient to place all messages for a command in a single CSECT or assembly module, which is important when message texts must be modified. Adding or updating a message is simpler when IKJEFF02 is used, rather than PUTLINE or PUTGET.

Passing Control to the TSO/E Message Issuer Routine

Your Command Processor can invoke the TSO/E message issuer routine using either the CALLTSSR or LINK macro instructions, specifying IKJEFF02 as the entry point name. However, you must first create the input parameter list and place its address into register 1. The input parameter list is described in “The Input Parameter List” on page 276.

IKJEFF02 can be invoked in either 24- or 31-bit addressing mode. IKJEFF02 can accept input above or below 16 MB in virtual storage. The caller's parameters must be in the primary address space.

The Input Parameter List

Use the IKJEFFMT macro to map the input parameter list for IKJEFF02. This parameter list identifies the message which is to be issued, describes inserts, if any, for the message, and indicates to IKJEFF02 whether to issue the message using PUTLINE, PUTGET, WTO or WTP. The parameter list also indicates to IKJEFF02 whether the message is to be provided in the language specified in the UPT, and contains the address of a CSECT that contains the text of the message.

This mapping macro allows you to request the standard format, which is the default, or the extended format of the parameter list. The extended format must be used if the message inserts or the extract buffers being passed to IKJEFF02 reside above 16 MB in virtual storage. If they reside below 16 MB, you do not need to use the extended format. However, all 31-bit addresses must be valid; that is the high-order bit must be zero. Your Command Processor must set the MTFMT bit in the input parameter list to reflect the format of the parameter list you are using.

The IKJEFFMT macro, which is provided in SYS1.MACLIB, has several options that your Command Processor can specify:

- Use the MTDSECT=YES option to map the MTDSECTD DSECT, instead of obtaining storage. MTDSECT=NO is the default.
- Use the MTFORMAT=NEW option to request the extended format; specify MTFORMAT=OLD to request the standard format. MTFORMAT=OLD is the default.
- The MTNINST option specifies the number of entries to be inserted into the message that IKJEFF02 issues.

Standard Format of the Input Parameter List

The IKJEFFMT macro generates the standard format input parameter list described below.

Table 73. Standard format of input parameter list

Offset dec(Hex)	Field name	Contents or meaning
0(0)	LISTPTR	Address of message description section of this parameter list. (The message description section begins with the MSGCSECT entry.)
4(4)	MTCPPL	Address of TMP's CPPL control block (required for PUTLINE or PUTGET).
8(8)	ECBPTR	Address of optional communications ECB for PUTLINE or PUTGET.
12(C)		Reserved.
12(C)	MTHIGH	High-order bit of reserved field turned on for standard linkage.
16(10)	MSGCSECT	Address of an assembly module or a CSECT containing IKJTSMSG macros that build message identifications and associated texts.
20(14)	SW	1-byte field of switches.
	MTNOIDSW	**1... ....** Message is printed; no message id is needed.
	MTPUTLSW	**.1.. ....** Message issued as PUTLINE. (Message inserts for a second-level message must be listed before inserts for a first-level message.) If this bit is zero, message issued as a PUTGET, with second-level message required and inserts for second-level messages necessarily following inserts for first-level messages.
	MTWTOSW	**..1. ....** Message issued as a WTO. Default is PUTGET.
	MTHEXSW	**...1 ....** Number translations to printable hexadecimal rather than default of printable decimal.

Table 73. Standard format of input parameter list (continued)

Offset dec(Hex)	Field name	Contents or meaning
	MTKEY1SW	**.... 1...** Modeset from key 1 to key 0 before issuing a PUTLINE or PUTGET message. Default is no modeset.
	MTJOBISW	**.... .1..** Blanks are compressed from inserts in the format of JOBNAME (JOBID). The blanks between (1) the JOBNAME and opening parenthesis and (2) the JOBID and closing parenthesis are removed. The maximum value for the message and insert lengths is 252 characters. Inserts and messages greater than 252 characters are truncated.
	MTWTPSW	**.... ..1.** Message issued as WTO with write-to-programmer routing code. Inserts are handled the same as for PUTLINE. Default is PUTGET.
	MTNHEXSW	**.... ...1** Number translations to printable decimal, even if larger than X'FFFF'. Default is printable hex above X'FFFF'.
21(15)	MTREPLYP	Address of reply from PUTGET. The reply text is preceded by a 2-byte field containing length of text plus header field.
24(18)	SW2	1-byte field of switches.
	MT2OLDSW	**1... ....** Field MTOLDPTR points to second level message already in PUTLINE/PUTGET (Output Line Descriptor) format. Default is IKJTSMSG format.
	MTDOMSW	**.1.. ....** Delete WTP or WTO messages from the display console.
	MTNOXQSW	**..1. ....** Override default of X'' around inserts converted to printable hex.
	MTNPLMSW	**...1 ....** Override default of error message if PUTLINE fails.
	MTPGMSW	**.... 1...** Request an error message if PUTGET fails.
	MTEXTRCN	**.... .1..** Request an extract and a message.
	MTFMT	**.... ..0.** Request standard (24-bit) format of this parameter list.
	MTTRANS	**.... ...1** Issue the message in the language specified in the UPT.
25(19)		Reserved.
28(1C)	MTOLDPTR	Pointer to OLD for second-level message, required if MT2OLDSW bit is on.
32(20)	MTEXTRLN	1-byte field indicating the length of the extract buffer. The caller provides this for the first-level message. When message translation is requested (that is, MTTRANS is ON), the caller provides a four-byte buffer. IKJEFF02 updates the buffer with the address of the translated message buffers that it returns. Therefore, you specify the address of a four-byte buffer in this field. For information about the form of the message buffers that IKJEFF02 returns, see Figure 122 on page 279. When message translation is not requested (that is, MTTRANS is OFF), the caller provides a buffer to contain the entire first-level message. Therefore, you specify the length of the entire buffer you are providing.

Table 73. Standard format of input parameter list (continued)

Offset dec(Hex)	Field name	Contents or meaning

33(21)	MTEXTRBF	A fullword field that points to the extract buffer that the caller provides for the first-level message.
		
		When message translation is requested (that is, MTTRANS is ON), the caller provides a four-byte buffer. IKJEFF02 updates the buffer with the address of the translated message buffers that it returns. Therefore, you specify the address of a four-byte buffer in this field. For information about the form of the message buffers that IKJEFF02 returns, see Figure 122 on page 279.
		
		When message translation is not requested (that is, MTTRANS is OFF), the caller provides a buffer to contain the entire first-level message. Therefore, you specify the address of the buffer you are providing to IKJEFF02. The maximum length of the buffer that the caller can provide is 255 bytes, based on the one-byte length field, MTEXTRLN.
		
		Upon return from IKJEFF02, the buffer contains the first-level message in the form:
		
		`   LL          00       Text`
		`(2 bytes)   (2 bytes)`
		
		where:
		
		LL
		indicates the length in hex of the entire message that was extracted into the caller's buffer, including the 4 byte length of the LL and 00 fields.
		
		00
		indicates a halfword offset containing 2 bytes of X'00'.
		
		Text
		indicates the actual first-level message text.

36(24)	MTEXTRL2	1-byte field indicating the length of the extract buffer the caller provides for the second-level message. When message translation is requested (that is, MTTRANS is ON), the caller provides a four-byte buffer that IKJEFF02 updates with the address of the translated message buffers that it returns. Therefore, you specify a value of 4 in this field. When message translation is not requested (that is, MTTRANS is OFF), the caller provides a buffer to contain the entire second-level message. Therefore, you specify the length of the entire buffer you are providing.

Table 73. Standard format of input parameter list (continued)

Offset dec(Hex)	Field name	Contents or meaning

37(25)	MTEXTRB2	A fullword field that points to the extract buffer that the caller provides for the second-level message.
		
		When message translation is requested (that is, MTTRANS is ON), the caller provides a four-byte buffer. IKJEFF02 updates the buffer with the address of the translated message buffers that it returns. Therefore, you specify the address of a four-byte buffer in this field. For information about the form of the message buffers that IKJEFF02 returns, see Figure 122 on page 279.
		
		When message translation is not requested (that is, MTTRANS is OFF), the caller provides a buffer to contain the entire second-level message. Therefore, you specify the address of the buffer you are providing to IKJEFF02. The maximum length of the buffer that the caller can provide is 255 bytes, based on the one-byte length field, MTEXTRL2.
		
		Upon return from IKJEFF02, the buffer contains the second-level message in the form:
		
		`   LL          00       Text`
		`(2 bytes)   (2 bytes)`
		
		where:
		
		LL
		indicates the length in hex of the entire message that was extracted into the caller's buffer, including the 4 byte length of the LL and 00 fields.
		
		00
		indicates a halfword offset containing 2 bytes of X'00'
		
		Text
		indicates the actual second-level message text.

40(28)	MSGID	Message's identifier in message CSECT, padded with blanks on the right.
44(2C)	MTINSRTS	Insert information for message. The following two fields are supplied for each insert.
44(2C)	MTLEN	Length of an insert for the message.
44(2C)	MTHIGHL	High-order bit is on if necessary to translate the first 1-4 bytes of the insert from hexadecimal to character (printable hexadecimal or decimal depending on whether MTHEXSW is set to ON or OFF).
45(2D)	MTADDR	Address of an insert for the message.

Note: If MTTRANS is on and extraction is requested, IKJEFF02 sets the fullword, pointed to by MTEXTRBF or MTEXTRB2, to the address of the translated text buffers in subpool 78. The user must free the translated text buffers.

The format of a translated text buffer is shown in Figure 122 on page 279. The pointer in the last message text line contains zero to indicate the end of the buffer.

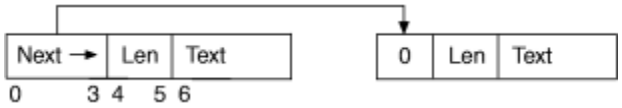


Figure 122. Translated Text Buffer Format

Extended Format of Input Parameter List

The IKJEFFMT macro generates the extended format input parameter list described below.

Table 74. Extended format of input parameter list

Offset dec(Hex)	Field name	Contents or meaning
0(0)	LISTPTR	Address of message description section of this parameter list. (The message description section begins with the MSGCSECT entry.)
4(4)	MTCPPL	Address of TMP's CPPL control block (required for PUTLINE or PUTGET).

Table 74. Extended format of input parameter list (continued)

Offset dec(Hex)	Field name	Contents or meaning
8(8)	ECBPTR	Address of optional communications ECB for PUTLINE or PUTGET.
12(C)		Reserved.
12(C)	MTHIGH	High-order bit of reserved field turned on for standard linkage.
16(10)	MSGCSECT	Address of an assembly module or a CSECT containing IKJTSMSG macros that build message identifications and associated texts.
20(14)	SW	1-byte field of switches.
	MTNOIDSW	**1... ....** Message is printed; no message id is needed.
	MTPUTLSW	**.1.. ....** Message issued as PUTLINE. (Message inserts for a second-level message must be listed before inserts for a first-level message.) If this bit is zero, message issued as a PUTGET, with second-level message required and inserts for second-level messages necessarily following inserts for first-level messages.
	MTWTOSW	**..1. ....** Message issued as a WTO. Default is PUTGET.
	MTHEXSW	**...1 ....** Number translations to printable hexadecimal rather than default of printable decimal.
	MTKEY1SW	**.... 1...** Modeset from key 1 to key 0 before issuing a PUTLINE or PUTGET message. Default is no modeset.
	MTJOBISW	**.... .1..** Blanks are compressed from xx(yy) format inserts. Default is no compression.
	MTWTPSW	**.... ..1.** Message issued as WTO with write-to-programmer routing code. Inserts are handled the same as for PUTLINE. Default is PUTGET.
	MTNHEXSW	**.... ...1** Number translations to printable decimal, even if larger than X'FFFF'. Default is printable hex above X'FFFF'.
21(15)	MTEXTRLN	Length of extract buffer (4 if MTTRANS is on).
22(16)	MTEXTRL2	Length of extract buffer for second-level message (4 if MTTRANS is on).
23(17)		Reserved.
24(18)	SW2	1-byte field of switches.
	MT2OLDSW	**1... ....** Field MTOLDPTR points to second-level message already in PUTLINE/PUTGET (Output Line Descriptor) format. Default is IKJTSMSG format.
	MTDOMSW	**.1.. ....** Delete WTP or WTO messages from the display console.
	MTNOXQSW	**..1. ....** Override default of X'' around inserts converted to printable hex.
	MTNPLMSW	**...1 ....** Override default of error message if PUTLINE fails.
	MTPGMSW	**.... 1...** Request an error message if PUTGET fails.
	MTEXTRCN	**.... .1..** Request an extract and a message.

Table 74. Extended format of input parameter list (continued)

Offset dec(Hex)	Field name	Contents or meaning
	MTFMT	**.... ..1.** Request extended (31-bit) format of this parameter list.
	MTTRANS	**.... ...1** Issue the message in the language specified in the UPT.
25(19)		Reserved.
28(1C)	MTOLDPTR	Pointer to OLD for second-level message, required if MT2OLDSW bit is on.
32(20)	MTEXTRBF	Pointer to extract buffer supplied by caller or pointer to a fullword if MTTRANS is on.
36(24)	MTEXTRB2	Pointer to extract buffer supplied by caller for second-level message or pointer to a fullword if MTTRANS is on.
40(28)	MSGID	Message's identifier in message CSECT, padded with blanks on the right.
44(2C)	MTREPLYP	Address of reply from PUTGET.
48(30)	MTINSRTS	Insert information for message. The following two fields are supplied for each insert.
48(30)	MTLEN	Length of an insert for the message.
48(30)	MTHIGHL	High-order bit is on if necessary to translate the first 1-4 bytes of the insert from hexadecimal to character (printable hexadecimal or decimal depending on whether MTHEXSW is set to ON or OFF).
52(34)	MTADDR	Address of an insert for the message.

Note: If MTTRANS is on and extraction is requested, IKJEFF02 sets the fullword, pointed to by MTEXTRBF or MTEXTRB2, to the address of the translated text buffers in subpool 78. The user must free the translated text buffers.

The format of a translated text buffer is shown in Figure 122 on page 279.

Using IKJTSMSG to Describe Message Text and Insert Locations

Use the IKJTSMSG macro to generate assembler language DC instructions describing the text and locations of inserts for a message which is to be issued by the TSO/E message issuer routine (IKJEFF02). All of the messages which a Command Processor issues should be grouped into an assembly module consisting entirely of IKJTSMSG macros preceded by a CSECT statement and followed by an END statement. The last IKJTSMSG macro in the CSECT must be a dummy entry with no operands.

The IKJTSMSG macro can be issued by a program loaded below or above 16 MB in virtual storage.

Figure 123 on page 281 shows the syntax of the IKJTSMSG macro instruction; each of the operands is explained following the figure.

```
[symbol]      IKJTSMSG ('msgid msgtext'),id1[,id2]
```

Figure 123. The IKJTSMSG macro instruction

msgid
The identifier which will be displayed when the message is issued.

msgtext
The text of the message. If an insert is necessary within the text of a message or at the end of a message, use the following rules:

- Indicate the location of an insert in the middle of a message by a ‘,,’.
- If the insert is to be located at the end of a message, indicate it by a ', following the message text.

id1
The internal identifier of the message. It can be from one to four characters and cannot contain a blank, comma, parenthesis, or an apostrophe. Pass this id to IKJEFF02 in the MSGID field of the parameter list. For a PUTGET message with more than one level, pass the id1 field of the first-level message. For a PUTLINE, WTO or write-to-programmer message with two levels, pass the id1 field of the second-level message.

id2
The internal identifier of a message to be chained to this message. For a PUTGET message, the first-level message would have an *id2* field identifying the second level, and the second-level message could have an *id2* field to identify another second-level, and so on. For a PUTLINE, WTO, or write-to-programmer message, the second-level message would have an *id2* field identifying the first level.

Return Codes from the TSO/E Message Issuer Routine

When the TSO/E message issuer routine returns control to its caller, register 15 contains one of the following return codes:

Table 75. Return codes from the TSO/E message issuer routine

Return code dec(Hex)	Meaning
0(0)	The message was issued successfully.
76(4C)	There was an error in the parameter list. A diagnostic message is also issued.
Other	This is either a PUTLINE or PUTGET return code. See “Return Codes from PUTLINE” on page 229 or “Return Codes from PUTGET” on page 251.

Example Using IKJTSMSG

Figure 124 on page 283 is an example that shows how a message module can be created for a SUBMIT command. The IKJTSMSG macro is used to describe the following:

- Message IKJ56250I is a single level PUTLINE message with one insert.
- Message IKJ56251I is a PUTLINE message with two levels.
- Message IKJ56252A is a PUTGET message with two levels.
- Message IKJ56253I is a PUTLINE message with an insert at the end of the text.
- The IKJTSMSG macro with no operands indicates the end of the message CSECT.

Figure 124 on page 283 shows an example of the IKJTSMSG macro.

```
*
* COMMENTS CAN PRECEDE OR FOLLOW THE MACROS TO LIST MODULES ISSUING
* THE MESSAGES AND GIVE THE MESSAGE DESCRIPTIONS.
*
IKJEFF03 CSECT
         IKJTSMSG ('IKJ56250I JOB',,'SUBMITTED'),00
*
         IKJTSMSG ('IKJ56251I ',,' COMMAND NOT AUTHORIZED+'),R01
         IKJTSMSG ('IKJ56251I YOUR INSTALLATION MUST AUTHORIZE USE OF TX
               HIS COMMAND'),01,R01
*        **  SECOND LEVEL POINTS TO FIRST LEVEL FOR PUTLINE **
*
         IKJTSMSG ('IKJ56252A ENTER JOBNAME CHARACTER+ -'),02,S02
         IKJTSMSG ('IKJ56252A JOBNAME IS CREATED FROM USERID PLUS',    X
               ' ONE ALPHANUMERIC OR SPECIAL CHARACTER'),S02
*        ** FIRST LEVEL POINTS TO SECOND LEVEL FOR PUTGET **
         IKJTSMSG ('IKJ56253I INVALID CHARACTER -',),03
*        ** THE COMMA AFTER THE APOSTROPHE INDICATES A TRAILING INSERT
*
         IKJTSMSG
         END   IKJEFF03
```

Figure 124. An Example Using the IKJTSMSG macro instruction

Chapter 12. Using the STAX service routine to handle attention interrupts

This chapter describes how to use the STAX service routine in your programs to handle attention interruptions.

Use the STAX service routine in your Command Processor or problem program to cause the system to recognize and schedule an attention exit that receives control when an attention interruption occurs. Your program provides the address of an attention exit routine to the system by issuing the STAX macro instruction.

The STAX service routine may be invoked in either 24-, 31-, or 64-bit addressing mode. Your attention exit routine receives control in the same addressing mode in which the corresponding STAX macro is issued.

For information on writing attention exit routines, see *z/OS TSO/E Programming Guide*.

The STAX macro instruction

Use the STAX macro instruction to specify the address of an attention exit routine that is to be given control asynchronously when a user presses the attention key or when a simulated attention is specified. (See "STATTN - Set Attention Simulation" on page 160 for a description of the simulated attention function.)

For attention interruptions that occur while a CLIST with a CLIST attention exit is processing, control passes to the last attention exit established with the CLSTATTN operand on the STAX macro (attention exits are searched in LIFO order until one is found that was established with the CLSTATTN operand). If no attention exit was established with the CLSTATTN operand, control passes to the first attention exit established.

The STAX macro instruction can also be used to cancel the last attention exit routine established by the task. To do this, specify the STAX macro instruction without any operands.

The STAX macro instruction is used only in a time sharing environment. When a task other than a TSO/E user issues the STAX macro, no action is taken. In addition, attention exits can be established only for time sharing tasks operating in the foreground.

Note: If the PSW key at the time of the STAX is zero, the PSW key when the exit is driven is zero. However, if the PSW key at the time of the STAX is not zero, the PSW key when the exit is driven is that of the job key.

The system routines that process attention handling require that the STAX parameter list remain unchanged for the duration of the program. Because the expansion of the STAX parameter list is usually located in an area that is reusable by the active program, you should either code the necessary protection to prevent overlays or you should make a copy of the parameter list in an area that is non-reusable.

Issue the STAX macro instruction to provide the information required by the STAX service routine. The STAX macro may be issued in either 24-, 31-, or 64-bit addressing mode, unless you are using the LINKAGE=BRANCH option. (See the LINKAGE operand description below for programming restrictions.) An attention exit routine receives control in the same addressing mode in which the STAX macro is issued.

The STAX macro instruction has a list, an execute, and a standard form.

The list form of the STAX macro instruction (MF=L) generates a STAX parameter list. The execute form of the STAX macro instruction (MF=E,address) completes or modifies that list and passes its address to the STAX service routine. The standard form does not require you to specify MF=L or MF=E.

Figure 125 on page 286 shows the format of the STAX macro instruction; each of the operands is explained following the figure.

```
[symbol]      STAX    [ exit address [,OBUF=(output buffer address,size)]]
                      [              [,IBUF=(input buffer address,size)] ]
                                     [,USADDR=user address]

                                     [,REPLACE={YES} ]
                                     [         {NO } ]

                                     [,DEFER={YES} ]
                                     [       {NO } ]

                                     [,LINKAGE={SVC   } ]
                                     [         {BRANCH} ]

                                     [,CLSTATTN={YES} ]
                                     [          {NO } ]

                                     [,IGNORE={YES} ]
                                     [        {NO } ]

                                     [,TOPLEVL={YES} ]
                                     [         {NO } ]

                                     {,MF=L           }
                                     {,MF=(E,address) }
```

Figure 125. Forms of the STAX macro instruction

Note: When the STAX macro is issued in 31- or 64-bit addressing mode, *exit address* and USADDR can reside above 16 MB in virtual storage. All other input must reside below 16 MB.

exit address
Specify the entry point of the routine to be given control when an attention interruption is received. You must specify the exit address in both the list and the execute forms of the STAX macro instruction when you are establishing an attention interruption handling exit.

You do not need to specify an exit address if you are using the DEFER operand as long as you code no other operands, except the MF operand. If you exclude the exit address and code no other operands, the STAX service routine cancels the previous attention exit established by the task issuing this STAX macro instruction.

OBUF=(*output buffer address*,*output buffer size*)

Output buffer address
Supply the address of a below-16M buffer you have obtained and initialized with the message to be put out to the terminal user who enters the attention interruption. This message can identify the exit routine and request information from the terminal user. It is sent to the terminal before the attention exit routine is given control.

Output buffer size
Indicate the number of characters in the output buffer. The size can range from 0 to 32,767 (2^{15}-1) inclusive.

IBUF=(*input buffer address*,*input buffer size*)

Input buffer address
Supply the address of a below-16M buffer you have obtained to receive responses from the terminal user. The attention exit routine is not given control until the STAX service routine has placed the terminal user's reply into this buffer.

Input buffer size
Indicate the number of bytes you have provided as an input buffer. The size can range from 0 to 32,767 (2^{15}-1) inclusive.

USADDR=(*user address*)
The user address is a 24-, 31-, or 64-bit address that points to any information you want passed to your attention handling exit routine when it is given control. When the attention exit gains control, register 1 points to the attention exit parameter list described in Table 76 on page 287.

Table 76. The attention exit parameter Llist

Number of bytes	Field name	Contents or meaning
4		The address of the terminal attention interrupt element (TAIE).
4		The address of the input buffer you specified as the IBUF operand of the STAX macro instruction. This field is zero if you did not include the IBUF operand in the STAX macro instruction.
4		The address of the user parameter information you specified as the USADDR operand of the STAX macro instruction. This field is zero if you did not include the USADDR operand in the STAX macro instruction.

REPLACE=YES | NO

YES
indicates that the attention exit specified by this STAX macro instruction replaces any attention exit specified by a STAX macro instruction previously issued by this task. YES is the default value. REPLACE implies establishing a new attention exit routine for the task, if no previous attention exit has been established.

NO
indicates that this attention exit be established as a new attention exit for this task, in addition to any that have been previously established for this task.

DEFER=YES | NO
The DEFER operand is optional. If the DEFER operand is coded in the STAX macro instruction, the option you request (YES or NO) applies to all tasks within the task chain in which the macro instruction was issued. Any task can issue the STAX macro instruction to specify DEFER=YES or NO; it is not necessary for the issuing task itself to have provided an attention exit routine. If the DEFER operand is not coded in the macro instruction, no action is taken by the STAX service routine regarding the deferral of attention exits.

YES
indicates that any attention interruptions received are to be queued and are not to be processed until:

- Another STAX macro instruction is executed specifying DEFER=NO
- The request block of the program that issued STAX DEFER=YES terminates and there is no other request block on the chain with attention interruptions to be deferred.

NO
indicates that the defer option is being canceled. Any attention interruptions received while the defer option was in effect are processed, unless the task is still not eligible for attention interruptions. If the DEFER operand is omitted, the control program leaves the deferral status unchanged.

Be aware that if a program issues a STAX macro instruction specifying DEFER=YES, the program can get into a situation where an attention interruption cannot be received from the terminal. If your program enters a loop or an unending wait before it has issued a STAX macro instruction specifying DEFER=NO, you cannot regain control at the terminal by entering an attention interruption.

You do not need to specify an exit address in a STAX macro instruction issued only to change deferral status.

LINKAGE=SVC | BRANCH (For MVS/ESA SP 4.2.2 or higher)
The LINKAGE operand is optional and is valid only when used with the DEFER operand. You cannot use any STAX operands other than DEFER when you use LINKAGE=BRANCH. It may be specified only on the standard form of the macro.

SVC
specifies that the STAX macro will generate an SVC to link from the calling program to the STAX SVC service routine. SVC is the default value.

BRANCH
specifies that the STAX macro will generate a branch instruction to link to the DEFER service of the STAX service routine. Because SVCs are not valid in cross-memory mode, this option allows you to defer attention exits in cross-memory mode.

The LINKAGE=BRANCH option requires that your program be in the following states:

Authorization
Supervisor

State
Key 0

Amode
31-bit

Rmode
Any

Interrupt Status
For DEFER=NO,LINKAGE=BRANCH, the caller must be enabled and unlocked.
For DEFER=YES,LINKAGE=BRANCH, the caller may be either enabled or disabled.

Serialization
None.

CLSTATTN=YES | NO
The CLSTATTN operand is optional. Code it only when you are establishing an attention exit. If you code the CLSTATTN keyword, you must provide an exit address.

YES
indicates that the attention exit being established can receive control for normal attention interruptions and for attention interruptions that occur while a CLIST with a CLIST attention exit is processing. When an attention interruption occurs while a CLIST with a CLIST attention exit is processing, the last attention exit established with the CLSTATTN=YES operand gains control to process the CLIST attention exit or pass control to the CLIST attention facility.

NO
indicates that the attention exit being established cannot process a CLIST that has a CLIST attention exit. No is the default value for the CLSTATTN operand.

IGNORE=YES | NO
The IGNORE operand is optional and is effective only when used in conjunction with the CLSTATTN=YES operand. Code the IGNORE operand only when attention interruptions are to be ignored or reestablished.

YES
When coded with the CLSTATTN operand (and an exit address) to establish an attention exit, indicates that attention interruptions are to be ignored when the attention exit being established receives control. Attention interruptions are reestablished when the attention exit returns control or issues the IGNORE=NO operand.

When coded within an attention exit established with the CLSTATTN=YES operand, IGNORE=YES also indicates that attention interruptions are to be ignored until the attention exit currently in control returns control or issues the IGNORE=NO operand. However, when coded within an attention exit, IGNORE=YES must be the only operand on the STAX macro instruction.

NO
indicates that attention interruptions are to be reestablished. An attention exit established with the CLSTATTN=YES operand can issue IGNORE=NO to indicate that attention interruptions are to be reestablished. The IGNORE=NO operand must be the only operand on the STAX macro instruction. IGNORE, without the CLSTATTN operand and an exit address, can only be issued by an attention exit that was established with the CLSTATTN=YES operand.

TOPLEVL=YES | NO
The TOPLEVL operand is optional. Code it only when you are establishing an attention exit.

If you code the TOPLEVL operand, you must provide an exit address.

If the ATTENTION key is pressed once, the first-level attention exit is given control; if pressed twice, the second-level attention exit is given control, and so forth. Use the TOPLEVL operand to control processing when the terminal user presses the attention key multiple times.

YES
indicates that when the attention key is pressed multiple times, control is *not* to be passed to higher-level attention exits than the exit being established. When an attention exit established with the TOPLEVL=YES operand receives control, the user cannot terminate execution of the exit by pressing the attention key. Therefore, the user cannot terminate execution of the program, and possibly see a TSO/E READY mode message.

NO
indicates that higher-level attention exits than the one being established can receive control when the attention key is pressed multiple times. For example, when the user presses the attention key two times, the second-level attention exit is given control. NO is the default value for the TOPLEVL operand.

MF=L | (E,*address*)
specifies the form of the STAX macro instruction.

L
specifies the list form of the STAX macro instruction. It generates a STAX parameter list.

(E,*address*)
specifies the execute form of the STAX macro instruction. It completes or modifies the STAX parameter list and passes the address of the parameter list to the STAX service routine. Place the address of the STAX parameter list (the address of the list form of the STAX macro instruction) into a register and specify that register number within parentheses.

You can place each of the required address and size parameters into registers and specify those registers, within parentheses, in the STAX macro instruction. Figure 126 on page 289 shows how an execute form of the STAX macro instruction might look if you load all the required parameters into registers.

```
STAX     (2),IBUF=((3),(4)),OBUF=((5),(6)),USADDR=(7),MF=(E,(1))
```

Figure 126. Using Registers in the STAX macro instruction

Return Codes from the STAX Service Routine

When your program issues the STAX macro instruction, control is returned to the instruction following the STAX macro instruction. When control is returned, register 1 either contains the address of the user parameter list provided for the previous exit for this task or it contains zero. Register 1 contains zero if this is the first STAX issued for this task, the STAX was issued with a cancel option, or the STAX was issued with only the DEFER option. If an error was detected (return code 8, 12, or 16), then the contents of register 1 is the same as it was at entry.

Register 15 contains one of the following return codes:

Table 77. Return codes from the STAX service routine

Return code dec(Hex)	Meaning
0(0)	The STAX service routine successfully completed the function you requested.
4(4)	Deferral of attention exits has already been requested and is presently in effect. Any other operands you specified in the STAX macro instruction have been processed successfully.

Table 77. Return codes from the STAX service routine (continued)

Return code dec(Hex)	Meaning
8(8)	The user of the DEFER option is not valid (asynchronous exit routine).
12(C)	The STAX macro has already been issued with the IGNORE=YES operand.
16(10)	The STAX macro has already been issued with the IGNORE=NO operand.
20(14)	A branch entry STAX DEFER=NO was requested, but attentions are not being deferred.
24(18)	A branch entry STAX DEFER=NO was requested, but the task is still not eligible to receive attention interruptions.

Note: If the STAX macro instruction is issued by a task that is not executing in a TSO/E user's address space, a return code of zero is passed to the caller in register 15. The contents of register 1 are not altered.

If a combination of parameters or the parameters themselves are not valid, an abend code of X'260' will be issued. The following types of errors cause an abend:

- Both DEFER=YES and DEFER=NO are specified.
- The input buffer address is not valid because the storage is not in same key as user's TCB.
- The input or output buffer size is not valid.
- A routine that is not a CLIST attention exit issued the STAX macro with the IGNORE parameter.
- A parameter list address is not valid.
- The format number of the parameter list is not valid.

Example Using the STAX macro instruction

The coding example shown in Figure 127 on page 291 uses the list and the execute forms of the STAX macro instruction to set up an attention handling exit. The OBUF operand provides a message to be written to the terminal when the attention interruption is received, and the IBUF operand provides space for an input buffer. Because the REPLACE operand is not coded on the macro instruction, the default value of YES is used. The attention handling exit established by this execution of the STAX macro instruction replaces the previous attention handling exit established for this task.

```
* THIS CODING EXAMPLE ISSUES A STAX MACRO INSTRUCTION TO SET UP AN
* ATTENTION EXIT.
*
*   PROCESSING
*      .
*      .
*      .
*
         LA    3,STAXLIST
* ISSUE THE EXECUTE FORM OF THE STAX MACRO INSTRUCTION
*
         STAX  ATTNEXIT,OBUF=(OUTBUF,31),IBUF=(INBUF,140),MF=(E,(3))
*
* CHECK THE RETURN CODE FROM THE STAX SERVICE ROUTINE. A ZERO RETURN
* CODE INDICATES SUCCESSFUL COMPLETION.
*
         LTR   15,15
         BNZ   ERRTN
*
*   PROCESSING
*
ERRTN      ERROR HANDLING ROUTINE
*            .
*            .
*            .
ATTNEXIT   ATTENTION EXIT ROUTINE
*            .
*            .
*            .
*
*
*   STORAGE DECLARATIONS
*
STAXLIST STAX  ATTNEXIT,MF=L             THIS LIST FORM OF THE STAX
*                                        MACRO INSTRUCTION EXPANDS AND
*                                        PROVIDES SPACE FOR THE STAX
*                                        PARAMETER LIST
*
OUTBUF   DC    C'THIS IS A SAMPLE ATTENTION EXIT'
         DS    0F
INBUF    DC    CL140'0'                  INITIALIZE 140 BYTES TO ZERO
*                                        AS THE INPUT BUFFER
*
         END
```

Figure 127. Example Using the STAX macro instruction

Chapter 13. Using the CLIST attention facility

This chapter describes how to use the CLIST attention facility to process a CLIST's attention routine.

Overview of the CLIST Attention Facility

If a program processes a CLIST with a CLIST attention routine, and an attention interruption occurs, the program's attention routine can process the CLIST's attention routine through the CLIST attention facility.

When an attention interruption occurs while a CLIST with an attention routine is processing, the attention routine established with the CLSTATTN operand receives control. The routine can then invoke the CLIST attention facility to process the CLIST's attention routine.

Note: If the program does not establish an attention routine with the CLSTATTN operand, control passes to the next highest-level attention routine established with the CLSTATTN operand coded on the STAX macro.

Figure 128 on page 293 shows the flow of control between a program and the CLIST attention facility.

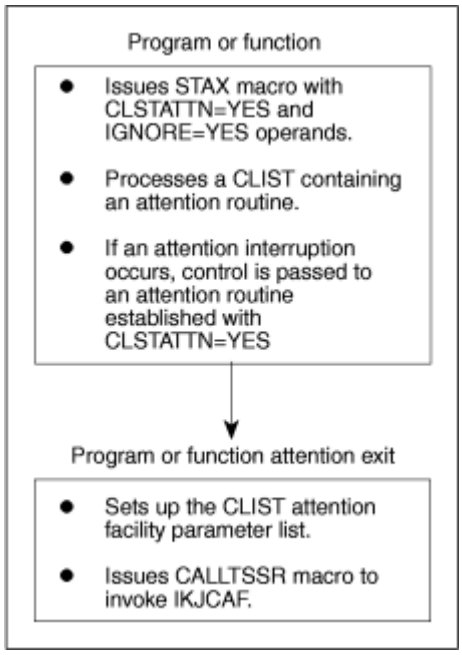


Figure 128. Flow of Control Between a Caller and the CLIST Attention Facility

Invoking the CLIST Attention Facility

To invoke the CLIST attention facility, the calling program must:

1. Establish an attention routine that specifies the parameters on the STAX macro to:
   - Receive control when an interruption occurs while processing a CLIST that contains a CLIST attention routine.
   - Prevent attention interruptions while processing the CLIST attention routine.
2. Build the CLIST attention facility parameter list (IKJCAFPL) and place its address in register 1.
3. Issue the CALLTSSR macro to pass control to the CLIST attention facility.

Establishing the Exit that Invokes IKJCAF

You must include the CLSTATTN operand on the STAX macro that establishes the caller's attention exit. When an attention interruption occurs, control passes to the last attention routine established with the CLSTATTN=YES operand.

You must also include the IGNORE=YES operand on the STAX macro that establishes the caller's attention routine. The IGNORE=YES operand indicates that attention interruptions are to be ignored while the routine is processing a CLIST's attention routine. If attention interruptions are not ignored, an abend can result.

See “The STAX macro instruction” on page 285 for restrictions governing the use of the CLSTATTN and IGNORE operands.

Passing Parameters to IKJCAF

The caller's attention routine must store the address of the CLIST attention facility parameter list (IKJCAFPL) in register 1. A caller executing below 16 MB in virtual storage must make sure the parameters it passes to IKJCAF are valid in a 31-bit environment. That is, the high-order byte of each address must be zero. If the high-order byte of each address is not zero, the CLIST attention facility returns to the caller with a return code of 28 (decimal).

You can use the IKJCAFPL DSECT, which is provided in SYS1.MACLIB, to map the fields of the parameter list. Table 78 on page 294 shows the format of the CLIST attention facility parameter list.

Table 78. The CLIST attention facility parameter list (IKJCAFPL)

Offset dec(Hex)	Number of bytes	Field name	Contents or meaning
0(0)	4	CAFCAF	Parameter list identifier, which is the value C‘CAF’.
4(4)	1	CAFLEV	Version number.
5(5)	3		Reserved.
8(8)	4	CAFTAIE	Address of the terminal attention interruption element (TAIE).
12(C)	4	CAFIOPL	Address of the input/output (IOPL) parameter list. The calling program must fill in all fields of the IOPL except IOPLIOPB.
16(10)	4	CAFPGPB	Address of the PUTGET parameter block (PGPB). The calling program must provide the space for the PGPB, which is defined by the IKJPGPB DSECT.
20(14)	4	CAFSTPB	Address of the STACK parameter block (STPB). The calling program must provide the space for the stack parameter block, which is defined by the IKJSTPB DSECT. IKJCAF fills in the STPB.
24(18)	4	CAFABEND	Abend code. If no abend, this field will be blanks.
28(1C)	4	CAFRSNCD	Abend reason code. If no abend, this field will be zeros.
32(20)	8		Reserved.

Passing Control to IKJCAF

Issue the CALLTSSR macro instruction in your program's attention routine to pass control to IKJCAF.

You must invoke IKJCAF in 31-bit addressing mode, after setting its addressing mode in bit 0 of register 14. The parameters you pass to IKJCAF must be in the primary address space. IKJCAF returns control in 31-bit addressing mode.

The following example shows the assembler code you can use to invoke the CLIST attention facility:

```
*
  R1=PARMADDR                       *Address of the
                                    *parameter list
*
```

```
  CALLTSSR EP=IKJCAF                *Passes control to the
*                                   *CLIST attention facility
```

Note: Any routine that uses the CALLTSSR macro instruction to invoke IKJCAF must include the CVT mapping macro (CVT), which is found in SYS1.MACLIB and the TSVT mapping macro (IKJTSVT), which is found in SYS1.MACLIB.

Returning from the CLIST Attention Facility

Output from the CLIST attention facility consists of a return code in register 15 and, if the return code is zero, a buffer. The buffer contains the TSO/E command coded in the CLIST attention routine.

When the CLIST attention facility returns control to your attention routine, your routine should do the following:

1. Check the return code in general register 15. Table 79 on page 295 shows the possible return codes from the CLIST attention facility.

Table 79. Return codes from the CLIST attention facility

Return code dec(Hex)	Meaning
0(0)	Normal completion.
8(8)	The current attention interruption is not for a CLIST with an attention routine.
16(10)	The CLIST attention facility issued an abend and retried. The reason code that accompanies the abend is the return code from the failing TSO/E service routine.
20(14)	The CLIST attention facility could not establish an ESTAE exit. No processing occurred.
24(18)	The CLIST attention facility received incorrect parameters from the calling program.
28(1C)	A 24-bit caller passed a parameter list that was not valid in the CLIST attention facility's 31-bit environment. (The high-order byte of each address was not zero.)

2. If attention interruptions have not been reestablished, issue the STAX macro with the IGNORE=NO operand.

   If you included the IGNORE=YES operand on the STAX macro that established the caller's attention exit, or the caller's exit issued the macro before invoking IKJCAF, the caller's exit receives control from IKJCAF in the IGNORE=YES state. The caller's exit must issue the STAX macro with the IGNORE=NO operand to reestablish attention interruptions. However, if the caller's exit does not reestablish attention interruptions, interruptions are automatically reestablished when the exit ends.

   If you did not include the IGNORE=YES operand on the STAX macro that established the caller's attention exit, or the caller's exit did not issue the macro before invoking IKJCAF, IKJCAF changes the IGNORE=NO state before it returns control to the caller's exit. When the caller's exit receives control, attention interruptions are reestablished.

   See “The STAX macro instruction” on page 285 for restrictions governing the use of the IGNORE operand.

3. Issue the FREEMAIN macro to free the storage for the input buffer. (If the caller's attention exit does not free the storage for the input buffer, the caller's mainline routine should free the storage.)

Chapter 14. Obtaining a List of data set names

This chapter describes how to use the TSO/E program ICQGCL00 in an application program to obtain a list of data set names that match specified criteria.

A valid ISPF environment must exist for an application to be able to invoke ICQGCL00.

ICQGCL00 lets application users search user catalogs for data set names that adhere to the criteria specified. Using the information returned by ICQGCL00, application programs can display those data set names to the user, who can view them or select them for further processing.

Operation of ICQGCL00

An application that uses ICQGCL00 specifies, as input parameters, the criteria to be used in searching the user catalog. The list of the names of the data sets that match the searched criteria are returned to the application in an ISPF table. If the table specified by the application does not exist, ICQGCL00 creates a temporary table. If the table does exist, and is sorted, the data sets are added to the table in sorted order. However, if the existing table is not sorted, ICQGCL00 adds the data set names to the bottom of the table. Figure 129 on page 297 shows the interaction between an application program and ICQGCL00.

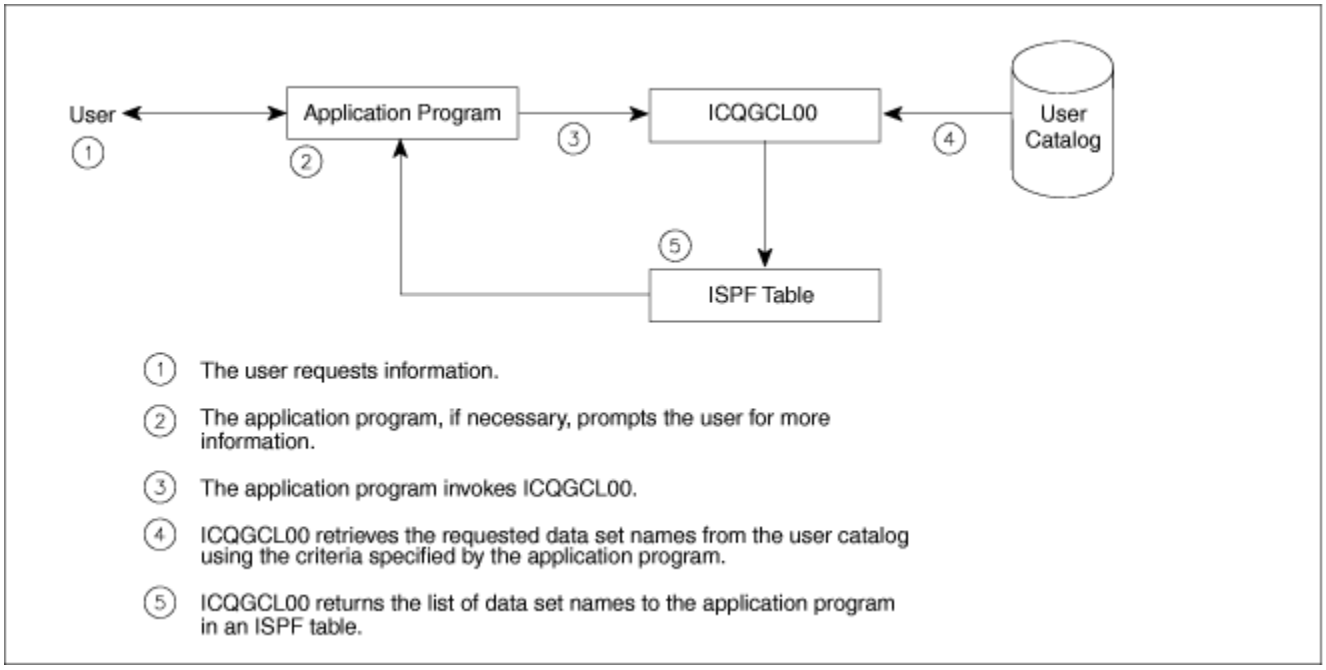


Figure 129. Using ICQGCL00 to Return a List of Data Set Names

A fully-qualified data set name has three fields: a user ID (or prefix), a first-level qualifier (or data set name) and a second-level qualifier (or descriptive qualifier). Search criteria can be specified for each field of a data set name. For example, an application can specify that all data set names with a user ID of MYDATA, a first-level qualifier beginning with the characters ICQ, and a second-level qualifier of CLIST be returned.

The input parameters limit the search. For example, they can specify that only those data set names that have exactly two qualifiers following the user ID be returned.

Invoking ICQGCL00

Applications invoke ICQGCL00 with the following syntax. The parameters PREFIX and TABLE are required; the others are optional keyword parameters.

```
ICQGCL00 +
           PREFIX(user ID) +
           QUAL1(first-level-qualifier) +
           QUAL2(second-level-qualifier) +
           EXACT(Y|N) +
           TABLE(table-name)
```

PREFIX(*user ID*)
specifies the user ID or prefix to be used as search criteria.

QUAL1(*first-level qualifier*)
specifies the first-level qualifier to be used as search criteria. An asterisk (*) can be specified to indicate that all data set names meet the search criteria for the first-level qualifier. Also, an asterisk can be used as a suffix of the first-level qualifier to indicate that all data set names that match the prefix characters meet the search criteria for the first-level qualifier.

QUAL2(*second-level qualifier*)
specifies the second-level qualifier to be used as search criteria. An asterisk (*) can be specified to indicate that all data set names meet the search criteria for the second-level qualifier. Also, an asterisk can be used as a suffix of the second-level qualifier to indicate that all data set names that match the prefix characters meet the search criteria for the second-level qualifier.

EXACT(Y | N)
specifies whether to return those data set names that match the specified qualifiers and have exactly that number of qualifiers. The default is Y, to return only those data set names that have exactly the number of specified qualifiers.

For example, to search for all data sets that have a user ID of MYDATA and a first-level qualifier beginning with ICQ, specify

```
ICQGCL00  PREFIX(MYDATA) QUAL1(ICQ*) EXACT(N) +
          TABLE(table-name)
```

To search for all data sets with a user ID of MYDATA that have exactly two levels of qualification after the prefix, specify:

```
ICQGCL00  PREFIX(MYDATA) QUAL1(*) QUAL2(*) EXACT(Y) +
          TABLE(table-name)
```

TABLE(*table-name*)
specifies the name of the ISPF table in which ICQGCL00 returns the names of the data sets that meet the search criteria. If the specified table does not exist, ICQGCL00 creates a temporary table. The following section describes the variables that ICQGCL00 creates in the table.

Output Table Variables

ICQGCL00 returns an ISPF table containing the names of the data sets that meet the search criteria. If the table does not exist, ICQGCL00 creates a temporary table; if the table does exist, ICQGCL00 adds rows to it at the bottom. Each row in the output table contains the following variables.

QCLPREF
indicates the user ID (or prefix) portion of the data set name.

QCLDSN1
indicates the first-level qualifier part of the data set name.

QCLDSN2
indicates the second-level qualifier part of the data set name.

QCLDSN
indicates the fully-qualified data set name, without quotes.

Return Codes from ICQGCL00

ICQGCL00 sets the return code in variable &LASTCC and the reason code in shared pool variable &QCLRESC. Table 80 on page 299 lists the return codes set by ICQGCL00.

Table 80. ICQGCL00 return codes

Return code	Meaning
0	ICQGCL00 completed successfully.
12	No data set names were found to match the search criteria.
16	An ISPF services TBADD error occurred. &QCLRESC contains the TBADD return code.
20	A prefix was not specified.
24	An error occurred. &QCLRESC contains the LOCATE return code.
28	ISPLINK module was not found.
32	A parameter was either missing or not valid.
36	An error occurred in the Parse Service Routine.
40	The application specified a table that does not exist and ICQGCL00 could not create it.

Example Using ICQGCL00

The CLIST in Figure 130 on page 300 is a sample application that invokes ISPF dialog management services to display input and output panels. The application displays an input panel on which user enters the information to be used as criteria for searching the user catalog. The input panel is shown in Figure 131 on page 301. The application invokes ICQGCL00 using the search criteria that the user specified. The application displays the results on an output panel (Figure 132 on page 301), which lists the data sets matching the search criteria.

```
/*****************************************************************/
/*                                                               */
/* THIS SAMPLE APPLICATION SEARCHES THE USER CATALOG FOR DATA SET  */
/* NAMES THAT MATCH THE CRITERIA SPECIFIED BY THE USER.  IT       */
/* RETURNS A LIST OF DATA SET NAMES THAT ARE DISPLAYED ON AN      */
/* OUTPUT PANEL.                                                 */
/*                                                               */
/*****************************************************************/
PROC 0
CONTROL END(ENDO)

/*  PROCESSING
/*      .
/*      .
/*      .
SET LOOP = YES
ISPEXEC DISPLAY PANEL(PANEL1)     /* DISPLAY PANEL TO USER
IF &LASTCC = 8 THEN +
  SET LOOP = NO                   /* END PRESSED, DON'T CONTINUE
DO WHILE &LOOP = YES              /* REPEAT UNTIL END PRESSED
  ISPEXEC TBCREATE QCLDSNTB +
      NAMES(QCLPREF QCLDSN1 QCLDSN2 QCLDSN QCLACT) +
           NOWRITE REPLACE        /* CREATE TABLE TO DISPLAY
      SET PARM1 = PREFIX(&NRSTR(&USER))    /* SET UP PARAMETERS
      SET PARM2 = QUAL1(&NRSTR(&QCLDSN1))
      SET PARM3 = QUAL2(&NRSTR(&QCLDSN2))
      SET PARM4 = EXACT(Y)
      SET PARM5 = TABLE(QCLDSNTB)
      ICQGCL00 &PARM1 &PARM2 &PARM3 &PARM4 &PARM5  /* CALL ICQGCL00+
                                    TO GET LIST OF DATA SET NAMES
      SET LCC = &LASTCC
      IF &LCC = 0 THEN +
        DO                        /* MATCH FOUND
          ISPEXEC TBTOP QCLDSNTB  /* GO TO TOP OF TABLE
          ISPEXEC TBDISPL QCLDSNTB PANEL(PANEL2)
          IF &LASTCC ¬= 8 THEN +
            DO
               .
               .                /* PROCESS SELECTIONS
               .
            ENDO
        ENDO
      ELSE +
        DO
          .
          .                       /* ISSUE APPROPRIATE MESSAGE
          .
        ENDO
  ISPEXEC DISPLAY PANEL(PANEL1)  /* DISPLAY PANEL TO USER
  IF &LASTCC = 8 THEN +
    SET LOOP = NO                 /* END PRESSED, DON'T CONTINUE
ENDO
/*
/*  PROCESSING
/*    .
/*    .
/*    .
```

Figure 130. A Sample Application Using ICQGCL00

```
)ATTR
       %   TYPE(TEXT) INTENS(HIGH)
       +   TYPE(TEXT) INTENS(LOW)
       _   TYPE(INPUT) INTENS(HIGH) CAPS(ON) JUST(LEFT)
       ¬   TYPE(INPUT) INTENS(HIGH) PAD(_) CAPS(ON)
)BODY
+               List of Data Set Names - Specification
+COMMAND%===>_ZCMD                                        +
+
+Specify criteria to use in searching for data set names below.
+
+ISPF Library:
+   Project          %===>_USER    +
+   Group            %===>_Q2      +
+   Type             %===>_Q3      +
+
+
+
)PROC
 VER (&USER,NB)               /* ensure PROJECT is specified
 IF (&Q2 = ' ')               /* if second qualifier is blank
   &Q2 = '*'                  /* default it to asterisk
 IF (&Q3 = ' ')               /* if third qualifier is blank
   &Q3 = '*'                  /* default it to asterisk
 &QCLPREF = &USER
 &QCLDSN1 = &Q2
 &QCLDSN2 = &Q3
)END
```

Figure 131. Sample Application Input Panel Definition (PANEL1)

```
)ATTR DEFAULT(%+_)
        % TYPE(TEXT) INTENS(HIGH)
        + TYPE(TEXT) INTENS(LOW)
        _ TYPE(INPUT) CAPS(ON) JUST(LEFT) PAD(_) INTENS(HIGH)
        @ TYPE(INPUT) CAPS(ON) JUST(LEFT) INTENS(HIGH)
        $ TYPE(OUTPUT) CAPS(ON) INTENS(HIGH)
)BODY
+                          Data Set List
+Command ===>@Z                                        %SCROLL ===>@Z   +
+
+Type action to be performed to the right of the data set name.
+
)MODEL
$QCLDSN                                        +@QCLACT
)INIT
 .ZVARS = '(ZCMD ZSCML)'
 IF (&ZSCML = ' ')             /* if scroll is blank,
   &ZSCML = 'PAGE'             /* default it to PAGE
 .CURSOR = QCLACT              /* set cursor at first action field
)PROC
)END
```

Figure 132. Sample Application Output Panel Definition (PANEL2)

Chapter 15. Using the space management CLIST ICQSPC00

CLIST ICQSPC00 enables an application program to access the TSO/E space management service. The space management service ensures that a specified data set has adequate free space for additional data. If the specified data set does not exist, ICQSPC00 can allocate it. You can invoke ICQSPC00 from within a program that runs under ISPF, or by typing "`tso %icqspc00 dsname optional parameters`" on the command line in an ISPF environment.

Functions of ICQSPC00

When you invoke ICQSPC00 and give it the name of a data set, it checks to see if the data set exists and how full the data set is. Based on the invocation parameters you use, ICQSPC00 determines whether more space is needed. If it is, ICQSPC00 tries to compress or reallocate the data set. If reallocation is needed, ICQSPC00 can check for and preserve RACF® protection.

One optional invocation parameter lets you indicate that, if the specified data set does not exist, it should be allocated. If you use this option, you must specify the allocation parameters.

Applications

Using ICQSPC00, you can:

- Manage any data set that will be edited or that will have data added to it in any other way.
- Periodically make sure your data sets have enough space left for future operations.
- Use an ISPF edit macro (SAVE or END) to invoke the space manager to make sure there is enough space left to save the data set being edited, and compress it if necessary.

  Note: Space management cannot enlarge a data set that is being edited. Therefore, the edit macro must invoke the space manager with the REALLOCATENEW(NO) parameter.
- Use the invocation parameters SPACEFULL and DIRFULL to make sure the data set is never more than a specified percentage full.
- Use the invocation parameters KBYTESFREE or DIRBLOCKSFREE to make sure there is a specific amount of space left in a data set before adding data.
- Allocate a data set, reallocate a data set, RACF protect a data set, check that a data set exists, or obtain allocation, protection, and directory information provided by the LISTDSI statement.
- Invoke the space management service from a CLIST as an error routine to allocate more space when an error occurs in processing a data set because the data set does not have enough space.

Considerations for Using ICQSPC00

- ISPF must be active to support the space management information panels.
- RACF accounting data is lost when ICQSPC00 reallocates a data set.
- If a data set is RACF protected by a discrete profile, but its profile has been deleted, ICQSPC00 cannot copy the protection to the new data set.
- If RACF is not installed, space management cannot copy the RACF universal access (UACC) from the old data set profile to the new data set profile. Instead, space management gives the new data set the default UACC, which might not match that of the old data set. For any release of RACF, however, space management copies the RACF access list from the old profile to the new profile.

- When a user enlarges a RACF-protected data set that has a high-level qualifier not equal to the user's user ID, the user needs the authority to create a RACF profile for the temporary data set used during reallocation.
- If the old data set is password-protected, and password-protected data sets are allowed, the data set is no longer password-protected if space management must reallocate the data set. (Whether password-protected data sets are allowed is specified when space management is invoked).
- Space management can only ensure that the data set has a specified percentage or amount of space free. If, after invoking ICQSPC00, an application or user adds more data to the data set than there is room for, the addition fails. However, an application could invoke ICQSPC00 to recover from the shortage.

Invoking ICQSPC00

The invocation syntax of ICQSPC00 is as follows. Only the dsname parameter is required; the others are optional keyword parameters.

```
%ICQSPC00  dsname  SPACEFULL(percent) +
                   SPACEINCREASE(percent) +
                   KBYTESFREE(nn) +
                   DIRFULL(percent) +
                   DIRINCREASE(percent) +
                   DIRBLOCKSFREE(nn) +
                   RECALL(yes/no) +
                   PROTECTNEW(yes/no) +
                   RACFUACC(none/read/update/alter) +
                   ALLOWPASSWORDS(yes/no) +
                   INFOPANEL(panelid) +
                   ASKPANEL(panelid) +
                   VERIFYPARMS(yes/no) +
                   COMPRESS +
                   TRACE +
                   REALLOCATENEW(yes/no) +
                   ALLOCATENEW(yes/no/ask) +
                     PRIMSPACE(nn) +
                     SECSPACE(nn) +
                     UNITS(tracks/cylinders) +
                     DIRBLOCKS(nn) +
                     BLKSIZE(nn) +
                     LRECL(nn) +
                     RECFM(string) +
                     LIKE(dsname)
```

DSNAME
specifies the name of the data set to be managed. If the data set name appears within single quotes, the CLIST treats it as fully qualified. If the data set name appears without quotes, the CLIST adds the prefix.

SPACEFULL(*percent*)
specifies the percentage that the data set can be full before ICQSPC00 should compress or reallocate it. Specify a number from 0 to 100. The default is 80, which means that when the data set is greater than 80% full, it will be compressed or reallocated.

SPACEINCREASE(*percent*)
specifies the percentage of increase for the data set's primary extent that ICQSPC00 is to use when reallocating the data set. Specify an integer. The default is 50.

KBYTESFREE(*nn*)
specifies the minimum amount of space (in kilobytes) that the data set must have free. If the data set does not have this much space free after ICQSPC00 compresses it, ICQSPC00 then reallocates the data set to force this much free space. There is no default.

DIRFULL(*percent*)
specifies the percentage that the directory for a partitioned data set can be full before it should be compressed or reallocated. Specify a number from 0 to 100. The default is 80, which means that when the directory is greater than 80% full, it will be compressed or reallocated.

DIRINCREASE(*percent*)
specifies the percentage that a partitioned data set's directory should be increased in size when being reallocated. Specify an integer. The default is 50.

DIRBLOCKSFREE(*nn*)
specifies the minimum number of directory blocks (in positive integers) that a partitioned data set must have free. If the data set does not have this many blocks free after ICQSPC00 compresses it, ICQSPC00 then reallocates it to force this many directory blocks to be free. There is no default.

RECALL(YES | NO | null)
specifies whether ICQSPC00 is to recall a data set that has been migrated by the Data Facility Hierarchical Storage Manager (DFHSM). Null specifies recalling a migrated data set from DASD. YES specifies recalling a data set from any medium, including tape. NO specifies that no data sets be recalled. The default is null.

PROTECTNEW(YES | NO)
specifies whether ICQSPC00 should RACF-protect a new data set. If you specify YES, ICQSPC00 protects the new data set with the value in RACFUACC and copies the list of users who have permission to access the data set. The default is NO.

RACFUACC(NONE | READ | UPDATE | ALTER)
specifies the universal RACF access for a new data set or an enlarged data set. When the data set has a generic profile, this parameter also protects the temporary data set used during reallocation. If the data set has a discrete RACF profile, ICQSPC00 ignores this parameter and copies the existing RACFUACC value and the list of users who have permission to access the data set. The default is NONE.

ALLOWPASSWORDS(YES | NO)
specifies whether ICQSPC00 should manage a password-protected data set. If you specify NO and the data set is password-protected, ICQSPC00 does not manage the data set and sets a non-zero return code. If you specify YES, ICQSPC00 executes normally, and the system prompts the user to enter the password when required. The default is NO.

INFOPANEL(*panel ID*)
specifies the ID of a panel that will override the default space management panel displayed during allocation. The default panel ID is ICQSPE00.

```
ICQSPE00      INFORMATION CENTER FACILITY - SPACE MANAGEMENT



     The data set specified below is running out of storage space.



     Please wait a few minutes while the storage space for
     this data set is automatically increased.

     None of your work will be lost.

     This panel is simply to inform you of this activity while
     it is taking place, because it can take a few moments.

     You will be returned to where you were interrupted when this
     processing is complete.

     You can then continue where you left off.
```

Figure 133. Default Panel for Space Management Allocation (ICQSPE00)

ASKPANEL(*panel ID*)
specifies the ID of a panel that will override the default space management panel displayed when a data set does not exist. The default panel ID is ICQSPE01.

```
ICQSPE01      INFORMATION CENTER FACILITY - SPACE MANAGEMENT
COMMAND ===>



   The data set specified below does not exist.



   To continue normally and have the data set created, press ENTER.
   To cancel, press END.
```

Figure 134. Default Panel for Space Management When a Data Set Does Not Exist (ICQSPE01)

VERIFYPARMS(YES | NO)
specifies whether ICQSPC00 is to check the syntax of the input parameters. Specify YES to check the parameter syntax. Specifying NO can improve performance. The default is YES.

COMPRESS
specifies that ICQSPC00 is to compress the data set. After compressing the data set, ICQSPC00 then checks to see if there is enough space, and if not, it reallocates the data set.

TRACE
specifies that space management should trace ICQSPC00 to the terminal. It sets the same trace level as TRACE2 in other Information Center Facility functions.

Note: If you invoke ICQSPC00 from an ISPF selection panel, it also supports the Information Center Facility trace options 'TRACE1', 'TRACE2', 'TRACE3.ICQSPC00', and 'TRACEOFF'.

REALLOCATENEW(YES | NO)
specifies whether to enlarge (reallocate) a data set if it is running out of space. Specify YES or NO. If you specify NO, ICQSPC00 does not reallocate the data set. The default is YES.

ALLOCATENEW(YES |NO | ASK)
specifies whether ICQSPC00 is to allocate a new data set if the one specified does not exist. If you specify YES, ICQSPC00 automatically allocates the data set. If you specify NO, ICQSPC00 does not allocate a new data set and sets a return code indicating that the data set was not found. If you specify ASK, the panel specified in the ASKPANEL parameter (or the default, ICQSPE01) is displayed, asking whether the file should be allocated. The default is ASK.

The following optional allocation parameters must be used when allocating a new data set:

PRIMSPACE(*nn*)
specifies the number of primary space units to be allocated to the data set. There is no default.

SECSPACE(*nn*)
specifies the number of secondary space units to be allocated to the data set. There is no default.

UNITS(TRACKS | CYLINDERS)
specifies the space units for the data set. It can be TRACKS or CYLINDERS. There is no default.

DIRBLOCKS(*nn*)
specifies the number of directory blocks to be allocated to a partitioned data set. For a sequential data set, you must specify DIRBLOCKS(0). There is no default.

BLKSIZE(*nn*)
specifies the block size of the data set. There is no default.

LRECL(*nn*)
specifies the logical record length of the data set. The length can range from 1 to 32756. There is no default.

RECFM(*string*)
specifies the record format of the data set. If more than one characteristic is specified, do not separate them with blanks or commas. For example, specify RECFM(FB) to indicate that the records are blocked and fixed-length.

LIKE(*dsname*)
specifies a data set that the space manager is to use as a model data set. There is no default.

To specify a fully-qualified data set name, enclose it in three sets of single quotes. For example, to use 'userid.MODEL.CLIST' as a model data set, specify:

```
LIKE('userid.MODEL.CLIST'')
```

Return and Reason Codes from ICQSPC00

This section describes the return codes and reason codes that ICQSPC00 returns. ICQSPC00 sets the return code in variable &LASTCC and the reason code in a shared pool variable, &QSPRC. The return code tells you whether the function completed successfully, and the reason code gives more detail about what ICQSPC00 did. Table 81 on page 307 and Table 82 on page 307 lists the return and reason codes set by ICQSPC00.

The return and reason codes have related messages. ICQSPC00 sets the return code messages in the shared pool variable &QSPCCMSG and the reason code messages in the shared pool variable &QSPRCMSG.

Table 81. ICQSPC00 return and reason codes

Return code &LASTCC	Meaning	Possible reason codes &QSPRC	Message ID &QSPCCMSG
0	The data set was managed successfully.	1 - 6	ICQSP000
8	An input parameter was not valid.	11 - 37	ICQSP010
20	The data set could not be managed.	41, 58, 431 - 435	ICQSP040

Table 82. ICQSPC00 reason codes

Reason code	Meaning	Message ID &QSPRCMSG
1	The data set did not need more space.	ICQSP001
2	The data set was compressed.	ICQSP002
3	The data set's directory was enlarged.	ICQSP003
4	The data set's primary quantity was enlarged.	ICQSP004
5	The data set's directory and primary quantity were enlarged.	ICQSP005
6	The data set did not exist, but was created.	ICQSP006
11	dsname is not valid.	ICQSP011
12	SPACEFULL is not valid. It must be an integer, 0 - 100.	ICQSP012
13	SPACEINCREASE is not valid. It must be an integer.	ICQSP013
14	KBYTESFREE is not valid. It must be an integer or blank.	ICQSP014
15	DIRFULL is not valid. It must be an integer, 0 - 100.	ICQSP015
16	DIRINCREASE is not valid. It must be an integer.	ICQSP016
17	DIRBLOCKSFREE is not valid. It must be an integer or blank.	ICQSP017
18	RECALL is not valid. It must be null, YES, or NO.	ICQSP018
19	PROTECTNEW is not valid. It must be YES or NO.	ICQSP019

Table 82. ICQSPC00 reason codes (continued)

Reason code	Meaning	Message ID &QSPRCMSG
21	RACFUACC is not valid. It must be NONE, READ, or ALTER.	ICQSP021
22	ALLOWPASSWORDS is not valid. It must be YES or NO.	ICQSP022
23	REALLOCATENEW is not valid. It must be YES or NO.	ICQSP023
24	ALLOCATENEW is not valid. It must be YES, NO, or ASK.	ICQSP024
25	PRIMSPACE is not valid. It must be an integer or blank.	ICQSP025
26	SECSPACE is not valid. It must be an integer or blank.	ICQSP026
27	UNITS not valid. It must be TRACKS, CYLINDERS, or blank.	ICQSP027
28	DIRBLOCKS is not valid. It must be an integer or blank.	ICQSP028
29	BLKSIZE is not valid. It must be an integer, 1 - 32760, or blank.	ICQSP029
30	LRECL is not valid. It must be an integer, 1 - 32756, or blank.	ICQSP030
31	RECFM is not valid. It must be alphabetic or blank.	ICQSP031
32	LIKE is not valid. It must be a valid data set name or blank.	ICQSP032
35	INFOPANEL is not valid. It must be non-blank.	ICQSP035
36	ASKPANEL is not valid. It must be non-blank.	ICQSP036
37	VERIFYPARMS is not valid. It must be YES or NO.	ICQSP037
38	ALLOCATION parameters are missing.	ICQSP038
41	Data set could not be managed: it is password-protected.	ICQSP041
42	Data set could not be managed: it is not sequential or partitioned.	ICQSP042
43	LISTDSI error occurred.	ICQSP043
44	Data set does not exist and could not be created.	ICQSP044
45	Data set was not created: insufficient authority.	ICQSP045
46	Data set was not created: allocation error.	ICQSP046
47	Data set was not created: ADDSD return code = &QSPRACFA.	ICQSP047
48	Data set was not created: user request.	ICQSP048
49	Data set was not compressed: IEBCOPY return code = &QSPCOPY.	ICQSP049
50	Data set was not enlarged: application request.	ICQSP050
51	Data set was not enlarged: insufficient authority.	ICQSP051
52	Data set was not enlarged: allocation error.	ICQSP052
53	Data set was not enlarged: ADDSD return code = &QSPRACFA.	ICQSP053
54	Data set was not enlarged: PERMIT return code = &QSPRACFP.	ICQSP054
55	Data set was not enlarged: IEBCOPY return code = &QSPCOPY.	ICQSP055
56	Data set was not enlarged: DELETE return code = &QSPDELET.	ICQSP056
57	Data set was not enlarged but renamed: an enlarge error occurred.	ICQSP057
58	Data set was not enlarged: IEBGENER return code = &QSPGENER.	ICQSP058

Table 82. ICQSPC00 reason codes (continued)		
Reason code	**Meaning**	**Message ID &QSPRCMSG**
431	You are not authorized to access the data set.	ICQSP431
432	Data set not available: migrated and not recalled.	ICQSP432
433	Data set not available: DFHSM migrated to a non-DASD device.	ICQSP433
434	Data set not available: it is not on a DASD device.	ICQSP434
435	Data set not available: volume containing it is not mounted.	ICQSP435

Examples Using ICQSPC00

The following CLISTs illustrate sample applications for the space management service.

Example 1: The SPACE MANAGER CLIST

The SPACE MANAGER CLIST in Figure 135 on page 309 receives a data set name as input using a positional parameter called DATA SET. The CLIST then invokes ICQSPC00, which checks to see if the data set is 75% full. If it is 75% or more full, ICQSPC00 tries to compress the data set. If compression is not possible, ICQSPC00 reallocates the data set, preserving the RACF protection. If the data set does not exist, ICQSPC00 can allocate a new data set with the specified name.

```
/*********************************************************************/
/* This CLIST invokes the space manager function to ensure that the  */
/* specified data set is no more than 75% full. If it is too full,   */
/* space management will enlarge (compress or reallocate) the data   */
/* set by 50%.  If the data set does not exist, space manager will   */
/* ask the user if it should be created. Data set protection will    */
/* be maintained if the data set is reallocated.                     */
/*                                                                   */
/*********************************************************************/
PROC 1 DATASET
%ICQSPC00 &DATASET           /* Invoke space manager to...           */+
          SPACEFULL(75)      /* If the data set is over 75% full,    */+
          SPACEINCREASE(50)  /*   increase it by 50%                 */+
          DIRFULL(75)        /* If the dir blocks are over 75% full, */+
          DIRINCREASE(50)    /*   increase them by 50%               */+
          ALLOCATENEW(ASK)   /* If the data set doesn't exist, ask     +
                                the user if it should be created,      +
                                using the following ALLOC parameters */+
            PRIMSPACE(10)    /*   Primary space of 10 tracks,        */+
            SECSPACE(5)      /*   Secondary space of 5 tracks,       */+
            UNITS(TRACKS)    /*   Allocate in tracks,                */+
            DIRBLOCKS(10)    /*   10 directory blocks,               */+
            BLKSIZE(800)     /*   Block size of 800                  */+
            LRECL(80)        /*   Logical record length of 80        */+
            RECFM(FB)        /*   Fixed, blocked record format       */
SET RCODE = &LASTCC          /* Save the return code                 */
ISPEXEC VGET (QSPRCMSG)      /* Get reason code message & display it */
ISPEXEC GETMSG MSG(&QSPRCMSG) LONGMSG(MESSAGE)
CONTROL ASIS
WRITE &MESSAGE
EXIT CODE(&RCODE)
```

Figure 135. Example 1: The SPACE MANAGER CLIST

Example 2: The SPACE ENLARGER CLIST

The SPACE ENLARGER CLIST in Figure 136 on page 310 invokes ICQSPC00 to automatically enlarge the specified data set by 50%. The SPACE ENLARGER CLIST displays a message if the data set was enlarged successfully or, if not, it displays an error message from ICQSPC00.

```
/*********************************************************************
/* This CLIST invokes ICQSPC00 to enlarge the specified data set    */
/* by 50%.                                                          */
/*********************************************************************/
PROC 1 DATASET
%ICQSPC00 &DATASET           /* Invoke space manager to...           */+
          SPACEFULL(0)       /* Force the space to be enlarged       */+
          DIRFULL(0)         /* Force the dir blocks to be enlarged  */+
          SPACEINCREASE(50)  /* Increase primary extent by 50%       */+
          DIRINCREASE(50)    /* Increase size of directory by 50%    */+
          ALLOCATENEW(NO)    /* Don't create the data set, if it       +
                                doesn't exist                        */
SET RCODE = &LASTCC          /* Save the return code                 */
CONTROL ASIS
IF &RCODE = 0 THEN +
  WRITE The data set was enlarged successfully.
ELSE +
 DO                          /* Error enlarging data set             */
  ISPEXEC VGET (QSPRCMSG)    /* Get reason code message & display it */
  ISPEXEC GETMSG MSG(&QSPRCMSG) LONGMSG(MESSAGE)
  WRITE &MESSAGE
 END
EXIT CODE(&RCODE)
```

Figure 136. Example 2: The SPACE ENLARGER CLIST

Chapter 16. Using IKJADTAB to change alternative library environments

This chapter describes how an application program can use the alternative library interface routine to create and remove alternative library environments and to modify alternative library definitions.

Functions of IKJADTAB

Use the alternative library interface routine (IKJADTAB) to create and remove alternative library environments and to modify alternative library definitions for CLIST and REXX libraries.

An environment application is a program that must disregard system level definitions and previous alternative definitions established by the user or by application programs. Therefore, before an application program invokes an environment application, the program must establish a new alternative library environment by creating an alternative library definition table. Creating an alternative library definition table is necessary if the environment being invoked uses alternative libraries for CLISTs and REXX execs. Conversely, when the environment application completes, the invoking program must remove the alternative library definition table that was created, and re-establish the previous environment.

Execs written in REXX can establish alternative load libraries that determine where the system searches for TSO/E commands issued from the exec. For execs that run in an ISPF environment, use IKJADTAB in an application program to reset DCB addresses for these alternative load module libraries.

An application program can use IKJADTAB to perform the following functions:

- Create a new alternative library definition table. Optionally, you can provide a model table whose contents are copied into the new table.
- Create an alternative library definition table if one does not exist. Add to the table the address of the DCB for an alternative load module library.
- Remove one or more alternative library definition tables.
- Remove all alternative library definition tables.

Passing Control to IKJADTAB

Your program can invoke the alternative library interface routine by using either:

- The CALLTSSR macro instruction, specifying IKJADTB as the entry point name
- The LINK macro instruction, specifying IKJADTAB as the entry point name.

However, you must first create the IKJADTAB parameter list and place its address into general register 1.

IKJADTAB must receive control in 31-bit addressing mode. The parameters you pass to IKJADTAB must be in the primary address space. This routine accepts input above or below 16 MB in virtual storage. Alternative library definition tables created by IKJADTAB reside above 16 MB in virtual storage.

The IKJADTAB Parameter List

On entry to IKJADTAB, register 1 must point to an IKJADTAB parameter list that you have built.

Figure 137 on page 312 shows the standard parameter list structure for IKJADTAB.

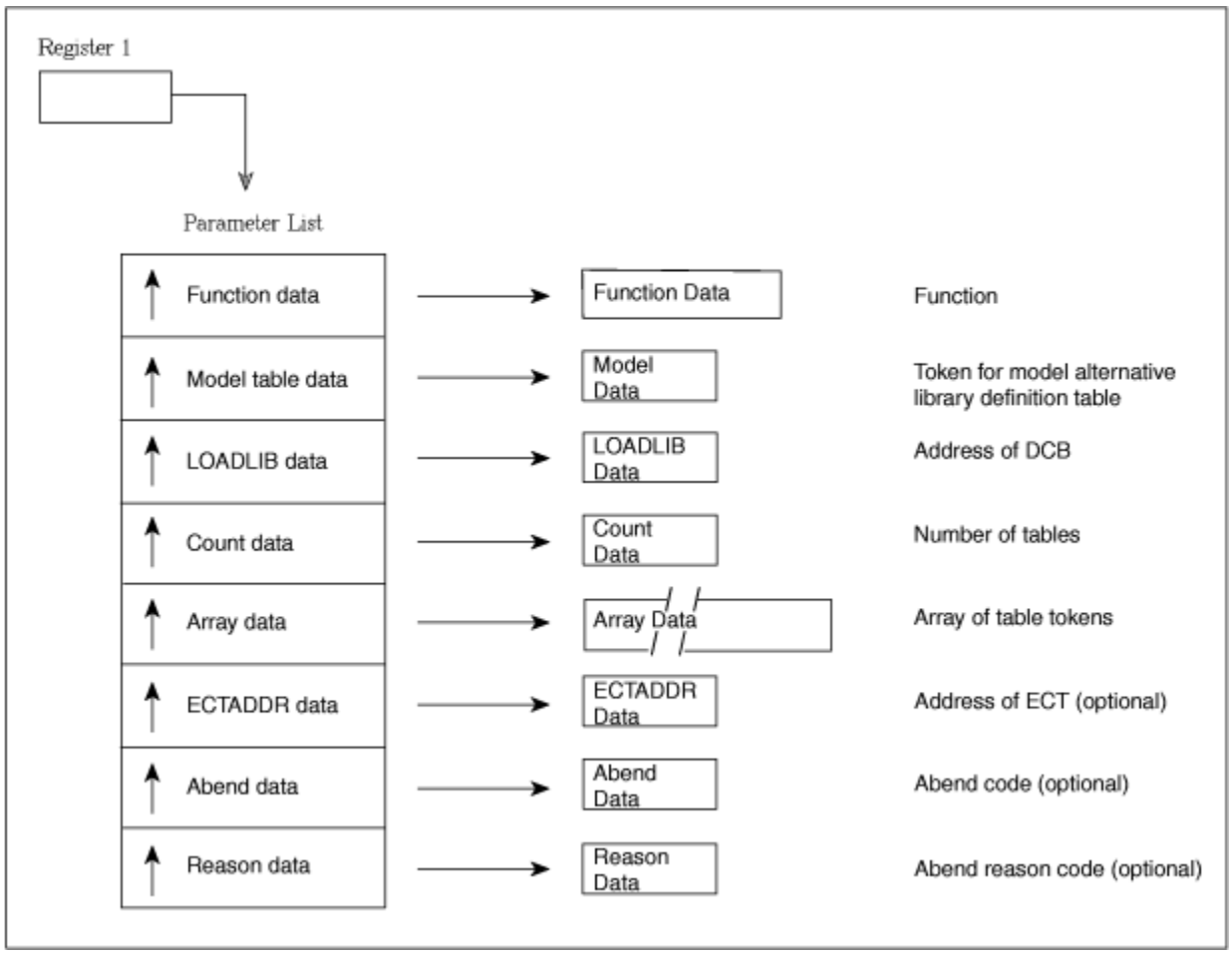


Figure 137. Parameter List Structure for IKJADTAB

You must turn on the high-order bit of the last address in the parameter list to indicate the end of the list.

Use the IKJADFMT mapping macro, which is provided in SYS1.MACLIB, to map the parameter list for IKJADTAB. IKJADFMT has the following syntax:

```
    IKJADFMT    [ADMAXCNT=xx]
```

ADMAXCNT=*xx*
specifies the number of elements in the array of table tokens. ADMAXCNT=1 is the default.

Table 83 on page 313 shows the names and descriptions of the IKJADTAB parameters.

Table 83. The parameters for IKJADTAB

Parameter	Function
ADTAB_FUNCTION	An 8-byte character string that indicates the function to be performed. Set the contents of the 8-byte field to one of the following EBCDIC values: **Value** **Function** **NEWTABLE** Create a new table. If your program uses the ADTAB_LIKE parameter to specify a model table, IKJADTAB copies the contents of the model table into the new table. Otherwise, IKJADTAB initializes a new table. **ADD_LOAD** If a table to contain alternative libraries does not exist, create one. IKJADTAB adds the DCB address, which you specify in the ADTAB_LOADLIB parameter, to the table. **ENDTABLE** Remove one or more tables. **ALLTABLS** Remove *all* tables. IKJADTAB removes tables created by your program and by callers of your program.
ADTAB_LIKE	A fullword containing the token for a model alternative library table. Use this parameter to pass alternative library definitions to a new environment. Specify the ADTAB_LIKE parameter with the NEWTABLE function to copy the contents of the model table into the new table being created. The calling program must set the token for the model table to zero if: • You use the NEWTABLE function to initialize a new alternative library table. • You specify a function other than NEWTABLE.
ADTAB_LOADLIB	A fullword containing the address of the DCB for an alternative load module library. Use this parameter with the ADD_LOAD function to place the DCB address for a load module library in an existing or newly created alternative library table. If you specify a function other than ADD_LOAD, you must set the address of the DCB to zero.
ADTAB_COUNT	A fullword containing the number of tables to be freed. Use this parameter with the ENDTABLE function to specify the *number* of tables to be freed. Use the ADTAB_ARRAY parameter to specify the *tokens* for the tables to be freed. If you specify a function other than ENDTABLE, you must set the number of tables to zero.
ADTAB_ARRAY	An array of table tokens. Each element of the array is a fullword containing the token for a table to be freed. Use this parameter with the ENDTABLE function to specify the *tokens* for the tables to be freed. Use the ADTAB_COUNT parameter to specify the *number* of tables to be freed. When IKJADTAB successfully frees a table, it sets the corresponding token in the parameter list to zero. If you specify a function other than ENDTABLE, you must set the value of the first array element to zero.
ADTAB_ECTADDR	A fullword containing the address of the current ECT under which the ALTLIB environment is to be created or removed. This parameter is optional. If you do not specify this parameter, or if you specify a binary zero as the ECT address, IKJADTAB uses the system ECT.

Table 83. The parameters for IKJADTAB (continued)	
Parameter	**Function**
ADTAB_ABEND	A fullword containing the hexadecimal abend code issued by IKJADTAB. IKJADTAB sets this parameter only when it sets a return code of 100 (decimal) or 104 (decimal). A return code of 100 indicates that a parameter is in inaccessible storage. A return code of 104 indicates an internal IKJADTAB error. This parameter is optional. If you do not specify this parameter, IKJADTAB will not return an abend code when it sets a return code of 100 or 104.
ADTAB_REASON_	A fullword containing the hexadecimal abend reason code issued by IKJADTAB. IKJADTAB sets this parameter only when it sets a return code of 100 (decimal) or 104 (decimal). A return code of 100 indicates that a parameter is in inaccessible storage. A return code of 104 indicates an internal IKJADTAB error. This parameter is optional. If you do not specify this parameter, IKJADTAB will not return an abend reason code when it sets a return code of 100 or 104.

Notes: IKJADFMT provides a "bounded" parameter list, which means that you must indicate the number of tables to be freed at the time the parameter list is built. If your program cannot determine the number of tables to be freed before building the parameter list, perform the following steps:

1. Use the IKJADFMT mapping macro specifying ADMAXCNT=1.
2. Obtain sufficient virtual storage for the array of table tokens.
3. Modify the ADTAB_COUNT parameter to reflect the number of tables to be freed.
4. In the parameter list, set the pointer to the array data (ADTAB_ARRAY@ field) to the address of the virtual storage obtained for the array.

Output from IKJADTAB

The alternative library interface routine passes a return code to the calling program in general register 15. Also, if IKJADTAB determines that the input parameters are valid, general register 0 contains the token for an alternative library definition table when:

- Either the NEWTABLE or ADD_LOAD functions are specified, and IKJADTAB issues a return code of 0 or 4.
- The ENDTABLE function is specified, and IKJADTAB issues a return code of 20. In this case, register 0 contains the token for the first alternative library definition table that could not be freed.

Return Codes from IKJADTAB

When IKJADTAB returns control to its caller, general register 15 contains one of the following return codes:

Table 84. Return codes from IKJADTAB

Return code dec(Hex)	Meaning
0(0)	Successful completion. Additional information, listed by function, follows. **Function** **Meaning** **NEWTABLE** A previous table did not exist and a new table was created. Register 0 contains the token for the new table. **ADD_LOAD** An alternative library definition table exists and was updated. Register 0 contains the token for the table. **ENDTABLE** IKJADTAB freed all tables requested and set the corresponding table tokens in the parameter list to zero. All associated ddnames are also freed. **ALLTABLS** IKJADTAB freed all tables created by your program and by callers of your program.
4(4)	Successful completion. Additional information, listed by function, follows. **Function** **Meaning** **NEWTABLE** A previous table existed, and a new table was created to replace it. Register 0 contains the token for the new table. **ADD_LOAD** An alternative library definition table did not exist, but IKJADTAB created and updated a new table. Register 0 contains the token for the new table. **ENDTABLE** IKJADTAB freed all alternative library definition tables and set the corresponding table tokens in the parameter list to zero. However, errors occurred when deallocating ddnames. IKJADTAB issues appropriate messages.
16(10)	Unsuccessful completion. Additional information, listed by function, follows: **Function** **Meaning** **NEWTABLE** The calling program specified a non-zero value for the token for the model table (ADTAB_LIKE parameter), but a table containing alternative libraries was not found. Register 0 is unchanged. IKJADTAB issues an appropriate message. **ADD_LOAD** The system previously created an environment that it expected to use for the current request. IKJADTAB determined that this environment is in error. Register 0 is unchanged. IKJADTAB issues an appropriate message.

Table 84. Return codes from IKJADTAB (continued)

Return code dec(Hex)	Meaning
20(14)	Unsuccessful completion. Additional information, listed by function, follows. **Function** **Meaning** **NEWTABLE** IKJADTAB was unable to obtain a table to contain alternative library definitions. Register 0 is unchanged. IKJADTAB issues an appropriate message. **ADD_LOAD** IKJADTAB was unable to obtain an alternative library definition table. Register 0 is unchanged. IKJADTAB issues an appropriate message. **ENDTABLE** At least one of the alternative library definition tables could not be freed. Register 0 contains the token for the first table that could not be freed. For tables that were freed successfully, IKJADTAB set the corresponding table tokens in the parameter list to zero. However, errors occurred when deallocating ddnames. IKJADTAB issues an appropriate message. **ALLTABLS** IKJADTAB could not free at least one of the tables.
28(1C)	Unsuccessful completion. IKJADTAB was unable to establish a recovery environment. IKJADTAB issues an appropriate message.
40(28)	Unsuccessful completion. A not valid function was specified (neither NEWTABLE, ADD_LOAD, ENDTABLE, nor ALLTABLS). IKJADTAB issues an appropriate message.
44(2C)	Unsuccessful completion. The NEWTABLE function was requested, but a non-zero value was specified for the ADTAB_LOADLIB parameter. IKJADTAB issues an appropriate message.
48(30)	Unsuccessful completion. The NEWTABLE function was requested, but a non-zero value was specified for the ADTAB_COUNT parameter. IKJADTAB issues an appropriate message.
52(34)	Unsuccessful completion. The NEWTABLE function was requested, but a non-zero value was specified for the ADTAB_ARRAY parameter. IKJADTAB issues an appropriate message.
56(38)	Unsuccessful completion. The ADD_LOAD function was requested, but a non-zero value was specified for the ADTAB_LIKE parameter. IKJADTAB issues an appropriate message.
60(3C)	Unsuccessful completion. The ADD_LOAD function was requested, but a non-zero value was specified for the ADTAB_COUNT parameter. IKJADTAB issues an appropriate message.
64(40)	Unsuccessful completion. The ADD_LOAD function was requested, but a non-zero value was specified for the ADTAB_ARRAY parameter. IKJADTAB issues an appropriate message.
68(44)	Unsuccessful completion. The ENDTABLE function was requested, but a non-zero value was specified for the ADTAB_LIKE parameter. IKJADTAB issues an appropriate message.
72(48)	Unsuccessful completion. The ENDTABLE function was requested, but a non-zero value was specified for the ADTAB_LOADLIB parameter. IKJADTAB issues an appropriate message.

Table 84. Return codes from IKJADTAB (continued)

Return code dec(Hex)	Meaning
76(4C)	Unsuccessful completion. The ALLTABLS function was requested, but a non-zero value was specified for the ADTAB_LIKE parameter. IKJADTAB issues an appropriate message.
80(50)	Unsuccessful completion. The ALLTABLS function was requested, but a non-zero value was specified for the ADTAB_LOADLIB parameter. IKJADTAB issues an appropriate message.
84(54)	Unsuccessful completion. The ALLTABLS function was requested, but a non-zero value was specified for the ADTAB_COUNT parameter. IKJADTAB issues an appropriate message.
88(58)	Unsuccessful completion. The ALLTABLS function was requested, but a non-zero value was specified for the ADTAB_ARRAY parameter. IKJADTAB issues an appropriate message.
100(64)	Unsuccessful completion. Parameters are in storage that cannot be accessed.
104(68)	Unsuccessful completion. An internal processing error occurred.
108(6C)	Unsuccessful completion. IKJADTAB was not invoked in a TSO/E environment.
112(70)	Unsuccessful completion. IKJADTAB was invoked in an authorized TSO/E environment.

Example Using IKJADTAB

Figure 138 on page 318 is an example showing how to invoke IKJADTAB to create and initialize a new alternative library definition table. This new table is created before the program invokes a new environment application.

The segment of assembler code shown sets up the parameter list for IKJADTAB and invokes IKJADTAB using the CALLTSSR macro instruction.

```
*********************************************************************
*                                                                   *
* ENTRY FROM THE TMP - REGISTER ONE CONTAINS A POINTER TO THE CPPL   *
*                                                                   *
*********************************************************************
         LR    R2,R1                    SAVE THE ADDRESS OF THE CPPL
         L     R3,12(R2)                PLACE THE ECT ADDRESS INTO A
                                        REGISTER
*********************************************************************
*                                                                   *
* SET UP THE PARAMETER LIST FOR IKJADTAB.  THE FUNCTION PARAMETER IS *
* SET TO 'NEWTABLE'.  THE ECTADDR PARAMETER IS SET TO THE ECT       *
* ADDRESS OF THE ECT PASSED AS INPUT TO THE COMMAND PROCESSOR.      *
* A VALUE OF ZERO IS PASSED FOR ALL OTHER PARAMETERS.               *
*                                                                   *
*********************************************************************

         XC    IKJADFMT(24),IKJADFMT        INITIALIZE PARAMETER VALUES
         MVC   ADTAB_FUNCTION(8),NEWTABLE   REQUEST NEWTABLE FUNCTION

         LA    R2,ADTAB_FUNCTION            PLACE ADDRESS OF FUNCTION
         ST    R2,ADTAB_FUNCTION@           DATA IN PARAMETER LIST

         LA    R2,ADTAB_LIKE                PLACE ADDRESS OF MODEL TABLE
         ST    R2,ADTAB_LIKE@               DATA IN PARAMETER LIST

         LA    R2,ADTAB_LOADLIB             PLACE ADDRESS OF LOADLIB
         ST    R2,ADTAB_LOADLIB@            DATA IN PARAMETER LIST

         LA    R2,ADTAB_COUNT               PLACE ADDRESS OF COUNT DATA
         ST    R2,ADTAB_COUNT@              IN PARAMETER LIST

         LA    R2,ADTAB_ARRAY               PLACE ADDRESS OF ARRAY DATA
         ST    R2,ADTAB_ARRAY@              IN PARAMETER LIST

         LA    R2,ADTAB_ECTADDR             PLACE ADDRESS OF ECTADDR DATA
         ST    R2,ADTAB_ECTADDR@            IN PARAMETER LIST

         LA    R2,ADTAB_ABEND               PLACE ADDRESS OF ABEND DATA
         ST    R2,ADTAB_ABEND@              IN PARAMETER LIST

         LA    R2,ADTAB_REASON_WORD         PLACE ADDRESS OF REASON_WORD
         ST    R2,ADTAB_REASON_WORD@        DATA IN PARAMETER LIST

         OI    ADTAB_REASON_WORD,B'10000000'
                                            TURN ON HIGH-ORDER BIT

         LA    R1,IKJADFMT_PLIST            REG 1 POINTS TO PARM LIST

         CALLTSSR EP=IKJADTB                INVOKE IKJADTAB, SPECIFYING
*                                           ENTRY POINT IKJADTB.

         ST    R15,IKJADTAB_RC              SAVE RETURN CODE
```

Figure 138. A Sample Program Using IKJADTAB

Figure of 'A Sample Program Using IKJADTAB' (**Continued**)

```
*********************************************************************
*                                                                   *
*  CHECK THE RETURN CODE FROM IKJADTAB.                             *
*                                                                   *
*********************************************************************

         LA    R3,4                        DETERMINE IF THE RETURN
         CR    R15,R3                      CODE IS 4 OR LESS
         BL    NO_ERROR                    BRANCH IF RETURN CODE IS
*                                          0 OR 4.
         B     ERROR                       BRANCH IF RETURN CODE IS
*                                          GREATER THAN 4.
NO_ERROR EQU   *
         ST    R0,USER_TOKEN               SAVE TOKEN FOR ALTERNATE
*                                          LIBRARY DEFINITION TABLE
```

```
*********************************************************************
*                                                                   *
*  IKJADTAB HAS COMPLETED SUCCESSFULLY.                             *
*  INVOKE THE ENVIRONMENT APPLICATION.                              *
*              .                                                    *
*              .                                                    *
*              .                                                    *
*              .                                                    *
*********************************************************************
         IKJADFMT ADMAXCNT=1               IKJADTAB PARAMETER LIST -
*                                          SPECIFY ONE ARRAY ELEMENT
NEWTABLE    DC    C'NEWTABLE'             CONSTANT FOR FUNCTION
IKJADTAB_RC DS    F                       SAVE AREA FOR RETURN CODE
USER_TOKEN  DS    F                       SAVE AREA FOR TABLE TOKEN
R1          EQU   1                       GENERAL REGISTER 1
R2          EQU   2                       GENERAL REGISTER 2
R3          EQU   3                       GENERAL REGISTER 3
R15         EQU   15                      GENERAL REGISTER 15
```

Chapter 17. Using the dynamic allocation interface routine DAIR

This chapter describes how to use the dynamic allocation interface routine (DAIR) in a Command Processor to allocate, free, concatenate and deconcatenate data sets during program execution.

Functions of the Dynamic Allocation Interface Routine

Dynamic allocation routines allocate, free, concatenate, and deconcatenate data sets dynamically, that is, during problem program execution. In the TSO/E environment, dynamic allocation permits the terminal monitor program, command processors, and other problem programs executing in the foreground region to allocate data sets after LOGON and free them before LOGOFF.

Programs that execute in the TSO/E environment can access dynamic allocation directly, using SVC 99, or through the dynamic allocation interface routine (DAIR). Though its use is not suggested because of reduced functions and additional system overhead, DAIR is documented in this book to provide compatibility for existing programs that use it. DAIR can be used to obtain information about a data set and, if necessary, invoke dynamic allocation routines to perform the requested function.

You can use DAIR to perform the following functions:

- Obtain the current status of a data set
- Allocate a data set
- Free a data set
- Concatenate data sets
- Deconcatenate data sets
- Build a list of attributes (DCB parameters) to be assigned to data sets
- Delete a list of attributes.

For a complete discussion of dynamic allocation, see *z/OS MVS Programming: Authorized Assembler Services Guide*.

Passing Control to DAIR

Your program can invoke DAIR by using the CALLTSSR macro instruction, specifying IKJDAIR as the entry point name. However, you must first create the DAIR parameter list (DAPL) and place its address into register 1. The DAPL is described in "The DAIR Parameter List (DAPL)" on page 321.

The DAIR service routine can be invoked in either 24- or 31-bit addressing mode. The caller's parameters must be in the primary address space. When invoked in 31-bit addressing mode, DAIR can be passed input above 16 MB in virtual storage.

The DAIR Parameter List (DAPL)

At entry to DAIR, register 1 must point to a DAIR parameter list that you have built. The addresses of the user profile table, environment control table, and protected step control block can be obtained from the Information Center Facility (CPPL) that the TMP passes to your Command Processor. Additional information on the address and creation of the user profile table, environment control table, and protected step control block is shown in Table 4 on page 15.

You can use the IKJDAPL DSECT, which is provided in SYS1.MACLIB to map the fields in the DAPL. Table 85 on page 322 shows the format of the DAPL.

Table 85. The DAIR parameter list (DAPL)

Offset dec(Hex)	Number of bytes	Field name	Contents or meaning
0(0)	4	DAPLUPT	The address of the user profile table.
4(4)	4	DAPLECT	The address of the environment control table.
8(8)	4	DAPLECB	The address of the calling program's event control block. The ECB is one word of real storage declared and initialized to zero by the calling routine.
12(C)	4	DAPLPSCB	The address of the protected step control block.
16(10)	4	DAPLDAPB	The address of the DAIR parameter block, created by the calling routine.

The DAIR Parameter Block (DAPB)

The fifth word of the DAIR parameter list must contain a pointer to a DAIR parameter block built by the calling routine.

It is a variable-size parameter block that contains, in the first two bytes, an entry code that defines the operation requested by the calling routine. The remaining bytes contain other information required by DAIR to perform the requested function. You must initialize the DAIR parameter block before calling DAIR. Unused fields should be set to zeros, or to blanks for character items. Table 86 on page 322 lists the DAIR entry codes and the functions requested by those codes.

Table 86. DAIR entry codes and their functions

Entry code	Function performed by DAIR
X'00'	Test if a given dsname or ddname is currently allocated to the caller.
X'04'	Test if a given dsname is currently allocated to the caller, or is in system catalog.
X'08'	Allocate a data set by dsname.
X'0C'	Concatenate data sets by ddname.
X'10'	Deconcatenate data sets by ddname.
X'14'	Search the system catalog for all qualifiers for a dsname. (The dsname alone represents an unqualified index entry.)
X'18'	Free a data set.
X'1C'	Allocate a ddname to a terminal.
X'24'	Allocate a data set by ddname or dsname.
X'28'	Perform a list of operations.
X'2C'	Mark data sets as not in use.
X'30'	Allocate a SYSOUT data set.
X'34'	Associate DCB parameter with a specified name for use with subsequent allocations.

The DAIR parameter blocks have the formats shown in the following tables. The formats of the blocks depend upon the function requested by the calling routine.

Determining if Ddname or Dsname is Allocated (Entry Code X’00’)

Build the DAIR parameter block shown in Table 87 on page 323 to request that DAIR determine whether the specified dsname or ddname is allocated. Use the IKJDAP00 mapping macro, which is provided in SYS1.MACLIB, to map this DAIR parameter block.

Table 87. DAIR parameter block for entry code X'00'

Number of bytes	Field name	Contents or meaning
2	DA00CD	Entry code X'0000'
2	DA00FLG	A flag field set by DAIR before returning to the calling routine. The flags have the following meaning: Byte 1: **0000 ....** Reserved. Set to zero. **.... 1...** Dsname or ddname is permanently allocated. **.... .1..** Ddname is a DYNAM. **.... ..1.** The dsname is currently allocated. **.... ...1** The ddname is currently allocated to the terminal. Byte 2: **0000 0000** Reserved. Set to zero.
4	DA00PDSN	Place in this field the address of the dsname buffer. The dsname buffer is a 46-byte field with the following format: The first two bytes contain the length, in bytes of the dsname; the next 44 bytes contain the dsname, left justified, and padded to the right with blanks.
8	DA00DDN	Contains the ddname for the requested data set. If a dsname is present, the DAIR service routine ignores the contents of this field.
1	DA00CTL	A flag field: **00.0 0000** Reserved bits. Set to zero. **..1. ....** Prefix user ID to dsname.
2		Reserved. Set these bytes to zero.
1	DA00DSO	A flag field. These flags describe the organization of the data. They are returned to the calling routine by the DAIR service routine. **1... ....** Indexed sequential organization **.1.. ....** Physical sequential organization **..1. ....** Direct organization **...1 ....** BTAM or QTAM line group **.... 1...** QTAM direct access message queue **.... .1..** QTAM problem program message queue **.... ..1.** Partitioned organization **.... ...1** Unmovable

After DAIR searches the data set entry for the fully-qualified data set name, register 15 contains one of the following DAIR return codes: 0, 4, or 52. See "Return Codes from DAIR" on page 340 for return code meanings.

Determining if Dsname is Allocated or is in the System Catalog (Entry Code X'04')

Build the DAIR parameter block shown in Table 88 on page 324 to request that DAIR determine whether the specified dsname is allocated. DAIR also searches the system catalog to find an entry for the dsname. Use the IKJDAP04 mapping macro, which is provided in SYS1.MACLIB, to map this DAIR parameter block.

Table 88. DAIR parameter block for entry code X'04'

Number of bytes	Field name	Contents or meaning
2	DA04CD	Entry code X'0004'
2	DA04FLG	A flag field set by DAIR before returning to the calling routine. The flags have the following meaning: Byte 1: **0000 0..0** Reserved. Set these bits to zero. **.... .1..** DAIR found the dsname in the catalog. **.... ..1.** The dsname is currently allocated. Byte 2: **0000 0000** Reserved. Set to zero.
2		Reserved. Set to zero.
2	DA04CTRC	These two bytes will contain an error code from the catalog management routines if an error was encountered by catalog management.
4	DA04PDSN	Place in this field the address of the dsname buffer. The dsname buffer is a 46-byte field with the following format: The first two bytes contain the length, in bytes, of the dsname; the next 44 bytes contain the dsname, left justified, and padded to the right with blanks.
1	DA04CTL	A flag field: **00.0 0000** Reserved. Set these bits to zero. **..1. ....** Prefix user ID to dsname.
2		Reserved. Set these bytes to zero.
1	DA04DSO	A flag field. These flags are set by the DAIR service routine; they describe the organization of the data set to the calling routine. These flags are returned only if the data set is currently allocated. **1... ....** Indexed sequential organization **.1.. ....** Physical sequential organization **..1. ....** Direct organization **...1 ....** BTAM or QTAM line group **.... 1...** QTAM direct access message queue **.... .1..** QTAM problem program message queue **.... ..1.** Partitioned organization **.... ...1** Unmovable

After attempting the requested function, DAIR returns, in register 15, one of the following codes: 0, 4, 8, or 52. See “Return Codes from DAIR” on page 340 for return code meanings.

Allocating a Data Set by Dsname (Entry Code X’08’)

Build the DAIR parameter block shown in Table 89 on page 325 to request that DAIR allocate a data set. Use the IKJDAP08 mapping macro, which is provided in SYS1.MACLIB, to map this DAIR parameter block. The exact action taken by DAIR depends upon the presence of the optional fields and the setting of bits in the control byte.

If the data set is new and you specify DSNAME, (NEW, CATLG) the data set is cataloged upon successful allocation. This is the only time a data set will be cataloged at allocation time. If the catalog attempt is unsuccessful, the data set is freed. If the proper indices are not present, the indices are built.

To allocate a utility data set use DAIR code X'08' and use a dsname of the form &name. If the &name is already allocated, that data set is used. If the &name is not found, a new data set is allocated.

To supply DCB information, provide the name of an attribute list that has been defined previously by a X'34' entry into DAIR.

When setting disposition in a parameter list, only one bit should be on.

For partitioned data sets, specifying the data set name and the member name for DAIR entry code X'08' causes the data set to be allocated, but no check is done to see if the member exists.

For example, to verify that the member really exists:

1. Allocate the data set with the member name using DAIR entry code X'08'.
2. Open the data set with DSORG=PO, MACRF=R.
3. Issue BLDL for the member. (The BLDL return code will indicate whether the member is there or not.)
4. Close the data set.
5. If BLDL indicates that the member does not exist, deallocate the data set using ddname and DAIR entry code X'18'.

The DAIR parameter block required for entry code X'08' has the format shown in Table 89 on page 325.

Table 89. DAIR parameter block for entry code X’08’

Number of bytes	Field name	Contents or meaning
2	DA08CD	Entry code X'0008'
2	DA08FLG	A flag field set by DAIR before returning to the calling routine. The flags have the following meaning: Byte 1: **1... ....** The data set is allocated but a secondary error occurred. Register 15 contains an error code. **.000 0000** Reserved. Set these bits to zero. Byte 2: Reserved. Set to zero.
2	DA08DARC	This field contains the error code, if any, returned from the dynamic allocation routines. (See “Reason Codes from Dynamic Allocation” on page 341.)
2	DA08CTRC	This field contains the error code, if any, returned from catalog management routines. (See “Return Codes from DAIR” on page 340.)

Table 89. DAIR parameter block for entry code X'08' (continued)

Number of bytes	Field name	Contents or meaning
4	DA08PDSN	Place in this field the address of the dsname buffer. The dsname buffer is a 46-byte field with the following format: The first two bytes contain the length, in bytes, of the dsname; the next 44 bytes contain the dsname, left justified and padded to the right with blanks. If this field (DA08PDSN) is zero, the system generates a data set name unless bit 5 in DA08CTL is on, in which case a DUMMY data set is allocated. The system also generates a name if the DA08PDSN field points to a dsname buffer which has a length of 44, is initialized to blanks, and bit 5 in DA08CTL is off.
8	DA08DDN	This field contains the ddname for the data set. If a specific ddname is not required, fill this field with eight blanks; DAIR will place in this field the ddname to which the data set is allocated.
8	DA08UNIT	
8	DA08SER	Serial number desired. Only the first six bytes are significant. If the serial number is less than six bytes, it must be padded to the right with blanks. If the serial number is omitted, the entire field must contain blanks. In this case the following is done: if the data set is a new data set, the system determines the volume to be used for the data set based on the unit information. If the data set already exists, volume and unit information are obtained from the catalog. If the information is not found in the catalog, the allocation request is denied.
4	DA08BLK	This is a 4-byte field used as follows: if the data set is a new data set and bit 0 in DA08CTL is off and bit 1 in DA08CTL is on, this field is used with DA08PQTY to determine the amount of direct access space to be allocated for the data set. If bit 6 of DA08CTL is off, the field is also used as DCB blocksize specification. The value for blocksize must be placed in the two low-order bytes, and the high-order bytes must be zero.
4	DA08PQTY	Primary space quantity desired. The high-order byte must be set to zero and the three low-order bytes should contain the space quantity required. If the quantity is omitted, the entire field must be set to zero. In the case of new direct access data sets, primary and secondary space and type of space are defaulted. Directory quantity is used if specified in DA08DQTY.
4	DA08SQTY	Secondary space quantity desired. The high-order byte must be set to zero; the three low-order bytes should contain the secondary space quantity required. If the quantity is omitted, the entire field must be set to zero.
4	DA08DQTY	Directory quantity required. The high-order byte must be set to zero; the three low-order bytes contain the number of directory blocks desired. If the quantity is omitted, the entire field must be set to zero.
8	DA08MNM	Contains a member name of a partitioned data set. If the name has less than eight characters, pad it to the right with blanks. If the name is omitted, the entire field must contain blanks.
8	DA08PSWD	Contains the password for the data set. If the password has less than eight characters, pad it to the right with blanks. If the password is omitted, the entire field must contain blanks.
1	DA08DSP1	Flag byte. Set the following bits to indicate the status of the data set: **0000 ....** Reserved. Set these bits to zero. **.... 1...** SHR **.... .1..** NEW **.... ..1.** MOD **.... ...1** OLD If this byte is zero, OLD is assumed. NEW or MOD is required if dsname is omitted.

Table 89. DAIR parameter block for entry code X'08' (continued)

Number of bytes	Field name	Contents or meaning
1	DA08DPS2	Flag byte. Set the following bits to indicate the normal disposition of the data set: **0000 ....** Reserved. Set these bits to zero. **.... 1...** KEEP **.... .1..** DELETE **.... ..1.** CATLG **.... ...1** UNCATLG If this byte is zero, it is defaulted as follows: if DA08DSP1 is NEW, DELETE is used; otherwise, KEEP is used.
1	DA08DPS3	Flag byte. Set the following bits to indicate the abnormal disposition of the data set: **0000 ....** Reserved. Set these bits to zero. **.... 1...** KEEP **.... .1..** DELETE **.... ..1.** CATLG **.... ...1** UNCATLG If this byte is zero, DA08DPS2 will be used.
1	DA08CTL	Flag byte. These flags indicate to the DAIR service routine what operations are to be performed: **xx.. ....** Indicate the type of units desired for the space parameters, as follows: **01.. ....** Units are in average block length. **10.. ....** Units are in tracks (TRKS). **11.. ....** Units are in cylinders (CYLS). **..1. ....** Prefix user ID to dsname. **...1 ....** RLSE is desired. **.... 1...** The data set is to be permanently allocated; it is not to be freed until specifically requested. **.... .1..** A DUMMY data set is desired. **.... ..1.** Attribute list name supplied. **.... ...0** Reserved. Set this bit to zero.
3		Reserved. Set these bytes to zero.

Table 89. DAIR parameter block for entry code X'08' (continued)

Number of bytes	Field name	Contents or meaning
1	DA08DSO	A flag field. These flags are set by the DAIR service routine; they describe the organization of the data set to the calling routine. **1... ....** Indexed sequential organization **.1.. ....** Physical sequential organization **..1. ....** Direct organization **...1 ....** BTAM or QTAM line group **.... 1...** QTAM direct access message queue **.... .1..** QTAM problem program message queue **.... ..1.** Partitioned organization **.... ...1** Unmovable
8	DA08ALN	Attribute list name, or a ddname from which DCB attributes should be copied (as in a JCL DCB reference). If the name is less than 8 characters, it should be padded to the right with blanks.

After attempting the requested function, DAIR returns, in register 15, one of the following codes: 0, 4, 8, 12, 16, 20, 28, 32, 44, or 52. See "Return Codes from DAIR" on page 340 for return code meanings.

Concatenating the Specified Ddnames (Entry Code X'0C')

Build the DAIR parameter block shown in Table 90 on page 328 to request that DAIR concatenate data sets. Use the IKJDAP0C mapping macro, which is provided in SYS1.MACLIB, to map this DAIR parameter block.

The ddnames listed in the DAIR parameter block are concatenated in the order in which they appear. All data sets listed by ddname in the DAIR parameter block must be currently allocated.

Table 90. DAIR parameter block for entry code X'0C'

Number of bytes	Field name	Contents or meaning
2	DA0CCD	Entry code X'000C'
2	DA0CDARC	This field contains the error code, if any, returned from the dynamic allocation routines. (See "Reason Codes from Dynamic Allocation" on page 341.)
2		Reserved field. Set this field to zero.
2	DA0CNUMB	Place in this field the number of data sets to be concatenated.
2		Reserved. Set this field to zero.
8	DA0CDDN	Place in this field the ddname of the first data set to be concatenated. This field is repeated for each ddname to be concatenated.

After attempting the requested function, DAIR returns, in register 15, one of the following codes: 0, 4, 12, or 52. See "Return Codes from DAIR" on page 340 for return code meanings.

Deconcatenating the Indicated Ddname (Entry Code X'10')

Build the DAIR parameter block shown in Table 91 on page 329 to request that DAIR deconcatenate a data set. The ddname specified within the DAIR parameter block must be concatenated previously, and is now to be deconcatenated.

Use the IKJDAP10 mapping macro, which is provided in SYS1.MACLIB, to map this DAIR parameter block.

Table 91. DAIR parameter block for entry code X'10'

Number of bytes	Field name	Contents or meaning
2	DA10CD	Entry code X'0010'
2	DA10DARC	This field contains the error code, if any, returned from the dynamic allocation routines. (See "Reason Codes from Dynamic Allocation" on page 341.)
2		Reserved field. Set this field to zero.
8	DA10DDN	Place in this field the ddname of the data set to be deconcatenated.

After attempting the requested function, DAIR returns, in register 15, one of the following codes: 0, 4, 12, or 52. See "Return Codes from DAIR" on page 340 for return code meanings.

Returning Qualifiers for the Specified Dsname (Entry Code X'14')

Build the DAIR parameter block shown in Table 92 on page 329 to request that DAIR return all qualifiers for the dsname specified. Use the IKJDAP14 mapping macro, which is provided in SYS1.MACLIB, to map this DAIR parameter block.

You must also provide the return area pointed to by the DA14PRET field in the DAIR parameter block. If the area you provide is larger than what is needed for all returned information, the remaining bytes in the area are set to zero by DAIR. If the area is smaller than the required size, it is filled to its limit, and the return code indicates this condition.

Table 92. DAIR parameter block for entry code X'14'

Number of bytes	Field name	Contents or meaning
2	DA14CD	Entry code X'0014'
4	DA14PDSN	Place in this field the address of the dsname buffer. The dsname buffer is a 46-byte field with the following format: The first two bytes contain the length, in bytes, of the dsname; the next 44 bytes contain the dsname, left justified and padded to the right with blanks. dsname alone represents an unqualified index entry.
4	DA14PRET	Place in this field the address of the return area in which DAIR is to place the qualifiers found for the dsname. Place the length of the return area in the first two bytes of the return area. Set the next two bytes in the return area to zero. DAIR returns each of the qualifiers it finds in two fullwords of storage beginning at the first word (offset 0) within the return area.
1	DA14CTL	A flag field: **00.0 0000** Reserved. Set these bits to zero. **..1. ....** Prefix user ID to dsname.
3		Reserved bytes. Set this field to zero.

After attempting the requested function, DAIR returns, in register 15, one of the following codes: 0, 4, 36, or 40. See "Return Codes from DAIR" on page 340 for return code meanings.

Freeing the Specified Data Set (Entry Code X'18')

Build the DAIR parameter block shown in Table 93 on page 330 to request that DAIR free a data set. Use the IKJDAP18 mapping macro, which is provided in SYS1.MACLIB, to map this DAIR parameter block.

The data set name represented by dsname is to be freed. If no dsname is given, the data set associated with the ddname is freed. If both ddname and dsname are given, DAIR ignores the ddname.

If the specified dsname is allocated several times to the user, all such allocations are freed.

When setting disposition in a parameter list, only one bit should be on.

Table 93. DAIR parameter block for entry code X'18'

Number of bytes	Field name	Contents or meaning
2	DA18CD	Entry code X'0018'
2	DA18FLG	A flag field set by DAIR before returning to the calling routine. The flags have the following meanings: Byte 1: **1... ....** The data set is freed but a secondary error occurred. Register 15 contains zero and the error information is in DA18DARC. **.000 0000** Reserved bits. Set to zero. Byte 2: Reserved. Set to zero.
2	DA18DARC	This field contains the error code, if any, returned from the dynamic allocation routines. (See "Reason Codes from Dynamic Allocation" on page 341.)
2	DA18CTRC	This field contains the error code, if any, returned from catalog management routines. (See "Return Codes from DAIR" on page 340.)
4	DA18PDSN	Place in this field the address of the dsname buffer. The dsname buffer is a 46-byte field with the following format: The first two bytes contain the length, in bytes, of the dsname; the next 44 bytes contain the dsname, left justified and padded to the right with blanks. This field is zero if the dsname is not specified.
8	DA18DDN	Place in this field the ddname of the data set to be freed, or blanks. If dsname is specified, this field is ignored.
8	DA18MNM	Contains the member name of a partitioned data set. If the name has less than eight characters, pad it to the right with blanks. If the name is omitted, the entire field must contain blanks.
2	DA18SCLS	SYSOUT class. The output class can be A-Z or 0-9 in the first byte. The second byte in the field is ignored. If SYSOUT is not specified, the first byte of this field must contain zeros or blanks.
1	DA18DPS2	Flag byte. Set the following bits to override the normal disposition of the data set: **0000 ....** Reserved bits. Set them to zero. **.... 1...** KEEP **.... .1..** DELETE **.... ..1.** CATLG **.... ...1** UNCATLG If the disposition specified at allocation is to be used, this field must contain zero.
1	DA18CTL	Flag byte. These flags indicate to the DAIR service routine what operations are to be performed: **..1. ....** Prefix user ID to dsname (requires DA18PDSN data be available). **00.. 0000** Reserved bits; set them to zero. **...1 ....** If this bit is on, permanently allocated data sets are deallocated. If the bit is off, the data set will be marked "not in use," if it is permanently allocated.
8		Reserved bytes; set this field to hexadecimal zeros.

After attempting the requested function, DAIR returns, in register 15, one of the following codes: 0, 4, 12, 24, 28, or 52 See “Return Codes from DAIR” on page 340 for return code meanings.

Allocating the Specified Ddname to the Terminal (Entry Code X’1C’)

Build the DAIR parameter block shown in Table 94 on page 331 to request that DAIR allocate a ddname to the terminal. Use the IKJDAP1C mapping macro, which is provided in SYS1.MACLIB, to map this DAIR parameter block.

If the DDNAME field is left blank, DAIR returns the allocated ddname in that field. To supply DCB information, provide the name of an attribute list that has been defined previously by a X'34' entry into DAIR, or the ddname of a currently allocated data set from which DCB attributes can be copied (as in a JCL DCB reference).

Table 94. DAIR parameter block for entry code X’1C’

Number of bytes	Field name	Contents or meaning
2	DA1CCD	Entry code X'001C'
2	DA1CDARC	This field contains the error code, if any, returned from the dynamic allocation routines. (See “Reason Codes from Dynamic Allocation” on page 341.)
1		Reserved field; set it to zero.
1	DA1CCTL	Control byte: **.... 1...** The data set is to be permanently allocated; it is not to be freed until specifically requested. **.... ..1.** Attribute list name supplied. **0000 .0.0** Reserved; set to zero.
8	DA1CDDN	Place in this field the ddname for the data set to be allocated to the terminal or blanks if the allocated ddname should be returned in this field.
8	DA1CALN	Attribute list name that has been defined previously by a X'34' entry into DAIR, or a ddname of a currently allocated data set from which DCB attributes can be copied. This field is used only if Bit 6 of DA1CCTL is set to one.

After attempting the requested function, DAIR returns, in register 15, one of the following codes: 0, 4, 12, 16, 20, 28, or 52. See “Return Codes from DAIR” on page 340 for return code meanings.

Allocating a Data Set by Ddname (Entry Code X’24’)

Build the DAIR parameter block shown in Table 95 on page 331 to request that DAIR allocate a data set by ddname. Use the IKJDAP24 mapping macro, which is provided in SYS1.MACLIB, to map this DAIR parameter block.

If DAIR locates the ddname you specify and a dsname is currently associated with it, the associated dsname is allocated overriding the dsname pointed to by the third word of your DAIR parameter block. The ddname might be found associated with a DUMMY, and if so an indicator is returned but no allocation takes place.

If DAIR cannot allocate by ddname, it will perform processing for code X'08' to allocate by dsname and generate a new ddname.

When setting disposition in a parameter list, only one bit should be on.

Table 95. DAIR parameter block for entry code X’24’

Number of bytes	Field name	Contents or meaning
2	DA24CD	Entry code X'0024'

Table 95. DAIR parameter block for entry code X'24' (continued)

Number of bytes	Field name	Contents or meaning
2	DA24FLG	A flag field set by DAIR before returning to the calling routine. The flags have the following meaning: Byte 1: **1... ....** The data set is allocated but a secondary error occurred. **.... 1...** Ddname requested is allocated as DUMMY. **.000 .000** Reserved bits. Set to zero. Byte 2: Reserved. Set to zero.
2	DA24DARC	This field contains the error code, if any, returned from the dynamic allocation routines. (See "Reason Codes from Dynamic Allocation" on page 341.)
2	DA24CTRC	This field contains the error code, if any, returned from catalog management routines. (See "Return Codes from DAIR" on page 340.)
4	DA24PDSN	Place in this field the address of the dsname buffer. The dsname buffer is a 46-byte field with the following format: The first two bytes contain the length, in bytes, of the dsname; the next 44 bytes contain the dsname, left justified and padded to the right with blanks. If the specified ddname is used, this field (DA24PDSN) is ignored.
8	DA24DDN	Place here the ddname for the data set to be allocated. This ddname is required. If the specified ddname is not allocated, then a generated ddname will be used with the dsname and the generated ddname will be returned in this field.
8	DA24UNIT	
8	DA24SER	Serial number desired. Only the first six bytes are significant. If the serial number is less than six bytes, it must be padded to the right with blanks. If the serial number is omitted, the entire field must contain blanks. In this case, the following is done: If the data set is a new data set, the system determines the volume to be used for the data set based on the unit information. If the data set already exists, volume and unit information are obtained from the catalog. If the information is not found in the catalog, the allocation request is denied.
4	DA24BLK	This is a 4-byte field used as follows: If the data set is a new data set and CONTROL bit 0 is off and bit 1 is on (see below), this field is used with PRIMARY SPACE QUANTITY to determine the amount of direct access space to be allocated for the data set. If CONTROL bit 6 is off, the field is also used as a DCB blocksize specification. The value for BLOCKSIZE must be placed in the two low-order bytes. The high-order byte must be zero.
4	DA24PQTY	Primary space quantity desired. The high-order byte must be set to zero; the three low-order bytes should contain the space quantity required. If the quantity is omitted, the entire field must be set to zero. In this case for new direct access data sets primary and secondary space, and type of space will be defaulted. Directory quantity will be used if specified in DA24DQTY.
4	DA24SQTY	Secondary space quantity desired. The high-order byte must be set to zero; the three low-order bytes should contain the secondary space quantity required. If the quantity is omitted, the entire field must be set to zero.
4	DA24DQTY	Directory quantity required. The high-order byte must be set to zero; the three low-order bytes contain the number of directory blocks desired. If the quantity is omitted, the entire field must be set to zero.
8	DA24MNM	Contains a member name of a partitioned data set. If the name has less than eight characters, pad it to the right with blanks. If the name is omitted, the entire field must contain blanks.
8	DA24PSWD	Contains the password for the data set. If the password has less than eight characters, pad it to the right with blanks. If the password is omitted, the entire field must contain blanks.

Table 95. DAIR parameter block for entry code X'24' (continued)

Number of bytes	Field name	Contents or meaning
1	DA24DSP1	Flag byte. Set the following bits to indicate the status of the data set: **0000 ....** Reserved. Set these bits to zero. **.... 1...** SHR **.... .1..** NEW **.... ..1.** MOD **.... ...1** OLD If this byte is zero, OLD is assumed.
1	DA24DPS2	Flag byte. Set the following bits to indicate the normal disposition of the data set: **0000 ....** Reserved bits. Set them to zero. **.... 1...** KEEP **.... .1..** DELETE **.... ..1.** CATLG **.... ...1** UNCATLG If this byte is zero, it is defaulted as follows: if DA24DSP1 is new, DELETE is used; otherwise KEEP is used.
1	DA24DPS3	Flag byte. Set the following bits to indicate the abnormal disposition of the data set: **0000 ....** Reserved bits. Set them to zero. **.... 1...** KEEP **.... .1..** DELETE **.... ..1.** CATLG **.... ...1** UNCATLG If this byte is omitted (set to zero), DA24DPS2 will be used.

Table 95. DAIR parameter block for entry code X'24' (continued)

Number of bytes	Field name	Contents or meaning
1	DA24CTL	Flag byte. These flags indicate to the DAIR service routine what operations are to be performed: **xx.. ....** Indicate the type of units desired for the space parameters, as follows: **01.. ....** Units are in average block length. **10.. ....** Units are in tracks (TRKS). **11.. ....** Units are in cylinders (CYLS). **..1. ....** Prefix user ID to dsname. **...1 ....** RLSE is desired. **.... 1...** The data set is to be permanently allocated; it is not be freed until specifically requested. **.... .1..** A DUMMY data set is desired. **.... ..1.** Attribute list name supplied. **.... ...0** Reserved bit; set to zero.
3		Reserved bytes; set them to zero.
1	DA24DSO	A flag field. These flags are set by the DAIR service routine; they describe the organization of the data set to the calling routine. **1... ....** Indexed sequential organization. **.1.. ....** Physical sequential organization. **..1. ....** Direct organization. **...1 ....** BTAM or QTAM line group. **.... 1...** QTAM direct access message queue. **.... .1..** QTAM problem program message queue. **.... ..1.** Partitioned organization. **.... ...1** Unmovable.
8	DA24ALN	Attribute list name, or a ddname from which DCB attributes should be copied (as in a JCL DCB reference). If the name is less than eight characters, it should be padded to the right with blanks.

After attempting the requested function, DAIR returns, in register 15, one of the following codes: 0, 4, 8, 12, 16, 20, or 52. See "Return Codes from DAIR" on page 340 for return code meanings.

Performing a List of DAIR Operations (Entry Code X'28')

Build the DAIR parameter block shown in Table 96 on page 335 to request that DAIR perform a list of operations. Use the IKJDAP28 mapping macro, which is provided in SYS1.MACLIB, to map this DAIR parameter block. This DAIR parameter block points to other DAPBs which request the operations to be performed.

All valid DAIR functions are acceptable; however, code X'14' or another code X'28' are ignored.

DAIR processes the requested operations in the order they are requested. DAIR processing stops with the first operation that fails.

Table 96. DAIR parameter block for entry code X'28'

Number of bytes	Field name	Contents or meaning
2	DA28CD	Entry code X'0028'
2	DA28NOP	Place in this field the number of operations to be performed.
4	DA28PFOP	DAIR fills this field with the address of the DAIR parameter block for the first operation that failed. If all operations are successful, this field will contain zero upon return from the DAIR service routine. If this field contains an address, register fifteen contains a return code.
4	DA28OPTR	Place in this field the address of the DAIR parameter block for the first operation you want performed. Repeat this field, filling it with the addresses of the DAPBs, for each of the operations to be performed.

After attempting the requested function, DAIR returns, in register 15, one of the following codes: 0, 4, 8, 12, 16, 20, 24, 28, 32, 44, or 52. For return code meanings see "Return Codes from DAIR" on page 340.

Marking Data Sets as Not in Use (Entry Code X'2C')

Build the DAIR parameter block shown in Table 97 on page 335 to request that DAIR mark data sets associated with a task control block as not in use. This allows data set entries to be reused.

This code should be issued by any Command Processor that attaches another Command Processor and detaches that Command Processor directly.

Use the IKJDAP2C mapping macro, which is provided in SYS1.MACLIB, to map this DAIR parameter block.

Table 97. DAIR parameter block for entry code X'2C'

Number of bytes	Field name	Contents or meaning
2	DA2CCD	Entry code X'002C'
2	DA2CFLG	A flag field. Set the bits to indicate to the DAIR service routine which data sets you want marked 'not in use'. **Hex Setting** **Meaning** **X'0000'** Mark all data sets of the indicated TCB 'not in use'. **X'0001'** Mark the specified ddname 'not in use'. **X'0002'** Mark all data sets associated with lower tasks 'not in use'.
4	DA2CTCB	Place in this field the address of the TCB for the task whose data sets are to be marked 'not in use'. DA2CFLG must be set to X'0000'.
8	DA2CDDN	Place in this field the ddname to be marked 'not in use'. DA2CFLG must be set to X'0001'.

After attempting the requested function, DAIR returns, in register 15, one of the following codes: 0, 4, or 52. For return code meanings see "Return Codes from DAIR" on page 340.

Allocating a SYSOUT Data Set to the Message Class (Entry Code X'30')

Build the DAIR parameter block shown in Table 98 on page 336 to request that DAIR allocate a SYSOUT data set to the message class. Use the IKJDAP30 mapping macro, which is provided in SYS1.MACLIB, to map this DAIR parameter block.

The action taken by DAIR is dependent upon the presence of the optional fields and the setting of bits in the control byte. To supply DCB information, provide the name of an attribute list that has been defined previously by a X'34' entry into DAIR, or the ddname of a currently allocated data set from which DCB attributes can be copied (as in a JCL DCB reference).

To place a SYSOUT data set in a class other than the message class, use DAIR entry code X'30' and when the output has been written, specify the desired class either by using DAIR entry code X'18', or execute the FREE command, after the program has completed processing.

When setting disposition in a parameter list, only one bit should be on.

Table 98. DAIR parameter block for entry code X'30'

Number of bytes	Field name	Contents or meaning
2	DA30CD	Entry code X'0030'
2	DA30FLG	A flag field set by DAIR before returning to the calling routine. The flags have the following meaning: Byte 1: **1... ....** The data set is allocated but a secondary error occurred. Register 15 contains an error code. **.000 0000** Reserved bits. Set to zero. Byte 2: Reserved. Set to zero.
2	DA30DARC	This field contains the error code, if any, returned from the dynamic allocation routines. (See "Reason Codes from Dynamic Allocation" on page 341.)
2		Reserved. Set this field to zero.
4	DA30PDSN	Place in this field the address of the dsname buffer or zeros. The dsname buffer is a 46-byte field which must appear as follows: The first two bytes must contain 44 (X'2C'); the next 44 bytes contain blanks.
8	DA30DDN	This field contains the ddname for the data set. If a specific ddname is not required, fill this field with eight blanks; DAIR will place in this field the ddname to which the data set is allocated.
8	DA30UNIT	This is an 8-byte field containing an esoteric group name, a generic group name, or a specific device number (in EBCDIC). If the unit information is less than eight characters, it must be padded to the right with blanks. If no information is to be provided, the field must be blank. In this case, DAIR will use a default set up at the time your TSO/E session is created. Since MVS/ESA SP 5.1 device numbers can be up to four digits long for increased addressability of I/O devices. If the string representing a device number is longer than three hexadecimal characters (for example, X'1ABC' or X'3390'), it *must* be preceded by a slash (/). A device number *may* be preceded by a slash even if it less than four characters long. This distinguishes numeric-only device numbers from generic device types that contain only four-character numerics. For example, a four-digit device number of X'1ABC', preceded by a slash, is represented in the 8-byte field as /1ABC..., where periods (.) represent blanks.
8	DA30SER	Serial number desired. Only the first six bytes are significant. If the serial number is less than six bytes, it must be padded to the right with blanks. If no volume serial number is specified, the field must be blank. In this case, the following is done: If the data set is a new data set, the system determines the volume to be used for the data set based on the unit information. If the data set already exists, volume and unit information are obtained from the catalog. If the information is not found in the catalog, the allocation request is denied.
4	DA30BLK	Block size requested. This figure represents the average record length desired.
4	DA30PQTY	Primary space quantity desired. The high-order byte must be set to zero; the three low-order bytes should contain the space quantity required. If the quantity is omitted, the entire field must be set to zero. In this case for new direct access data sets primary and secondary space, and type of space will be defaulted.

Table 98. DAIR parameter block for entry code X'30' (continued)

Number of bytes	Field name	Contents or meaning
4	DA30SQTY	Secondary space quantity desired. The high-order byte must be set to zero; the three low-order bytes should contain the secondary space quantity required. If the quantity is omitted, the entire field must be set to zero.
8	DA30PGNM	Place in this field the member name of a special user program to handle SYSOUT operations. Fill this field with blanks if you do not provide a program name.
4	DA30FORM	Form number. This form number indicates that the output should be printed or punched on a specific output form. It is a four character number. This field must be filled with blanks if this parameter is omitted.
2	DA30OCLS	SYSOUT class. The data set will be allocated to the message class, regardless of the class you specify here. To place a SYSOUT data set in a class other than the message class, use DAIR entry code X'30' and when the output has been written, specify the desired class by using DAIR entry code X'18'.
1		Reserved. Set this field to zero.
1	DA30CTL	Flag byte. These flags indicate to the DAIR service routine what operations are to be performed: **xx.. ....** Indicate the type of units desired for the space parameters, as follows: **01.. ....** Units are in average block length. **10.. ....** Units are in tracks (TRKS). **11.. ....** Units are in cylinders (CYLS). **..1. ....** Prefix user ID to dsname. **...1 ....** RLSE is desired. **.... 1...** The data set is to be permanently allocated; it is not to be freed until specifically requested. **.... .1..** A DUMMY data set is desired. **.... ..1.** Attribute list name specified. **.... ...0** Reserved bit; set to zero.
8	DA30ALN	Attribute list name, or a ddname from which DCB attributes should be copied (as in a JCL DCB reference). If the name is less than eight characters, it should be padded to the right with blanks.

After attempting the requested function, DAIR returns, in register 15, one of the following codes: 0, 4, 12, 16, 20, 28, or 52. See "Return Codes from DAIR" on page 340 for return code meanings.

Associating DCB Parameters with a Specified Name (Entry Code X'34')

Build the DAIR parameter block shown in Table 99 on page 338 to request that DCB parameters to be used with subsequent allocations are associated with a specified attribute name. Use the IKJDAP34 mapping macro, which is provided in SYS1.MACLIB, to map this DAIR parameter block.

The following functions related to attribute names are available using code X'34':

- Associate a set of DCB parameters to be used in subsequent allocations.
- Search on the attribute name.
- Delete the attribute name.

Note: When you request that DAIR associate DCB parameters with a specified name, you must also build a DAIR attribute control block (DAIRACB).

Table 99. DAIR parameter block for entry code X'34'

Number of bytes	Field name	Contents or meaning
2	DA34CD	Entry code X'0034'
2	DA34FLG	A flag field set by DAIR before returning to the calling routine. The flags have the following meaning:
	DA34FIND	Byte 1: **1... ....** An attribute list name was found. **0... ....** An attribute list name was not found. **.000 0000** Reserved bits. Set to zero. Byte 2: Reserved. Set to zero.
2	DA34DARC	This field contains the code returned from the dynamic allocation routines. (See "Reason Codes from Dynamic Allocation" on page 341.)
1	DA34CTRL	Flag byte. These flags indicate to DAIR what operations are to be performed:
	DA34SRCH	**1... ....** Search for the attribute list name specified in field DA34NAME.
	DA34CHN	**.1.. ....** Build and chain an attribute list.
	DA34UNCH	**..1. ....** Delete an attribute list name. **...0 0000** Reserved bits. Set to zero.
1		Reserved. Set to zero.
8	DA34NAME	This field contains the name for the list of attributes.
4	DA34ADDR	This field contains the address of the DAIR attribute control block (DAIRACB). This field need only be specified if bit 1 of DA34CTRL is on.

After attempting the requested function, DAIR returns, in register 15, one of the following codes: 0, 4, 12, or 52. See "Return Codes from DAIR" on page 340 for return code meanings.

The DAIR Attribute Control Block (DAIRACB)

Build the DAIRACB shown in Table 100 on page 338 when you request that DAIR construct an attribute list. Place the address of the DAIRACB into the DA34ADDR field of the code X'34' DAIR parameter block shown in Table 99 on page 338. Use the IKJDACB mapping macro, which is provided in SYS1.MACLIB, to map the DAIRACB.

Table 100. DAIR attribute control block (DAIRACB)

Number of bytes	Field name	Contents or meaning
8		Reserved.
8	DAIMASK	First 6 bytes and eighth byte are reserved.
	DAILABEL	Seventh-byte flags. These flags indicate the INOUT/OUTIN options of the OPEN macro.

Table 100. DAIR attribute control block (DAIRACB) (continued)

Number of bytes	Field name	Contents or meaning
	DAIINOUT	**1... ....** Use the INOUT option. **.1.. ....** Use the OUTIN option. **..00 0000** Reserved bits. Should be set to zero.
3		Reserved. Should be set to zero.
3	DAIEXPDT	This field contains a data set expiration date specified in binary.
	DAIYEAR	The first byte contains the expiration year.
	DAIDAY	The next 2 bytes contain the expiration day, left justified. For example, the date 99352 is specified ‘630160’B.
2		Reserved. Should be set to zero.
1	DAIBUFNO	This field contains the number of buffers required.
1	DAIBFTEK	This field contains the buffer type and alignment. **.1.. ....** Simple buffering (S). **.11. ....** Automatic record area construction (A). **..1. ....** Record buffering (R). **...1 ....** Exchange buffering (E). **.... ..1.** Doubleword boundary (D). **.... ...1** Fullword boundary (F). **0... 00..** Reserved bits. Should be set to zero.
2	DAIBUFL	This field contains the buffer length.
1	DAIEROPT	This field indicates the error options: **1... ....** Accept error record. **.1.. ....** Skip error record. **..1. ....** Abnormal EOT. **...0 0000** Reserved bits. Should be set to zero.
1	DAIEKYLE	This field contains the key length.
6		Reserved. Should be set to zero.

Table 100. DAIR attribute control block (DAIRACB) (continued)

Number of bytes	Field name	Contents or meaning
1	DAIRECFM	This field indicates the record format: **1... ....** Fixed (F) **.1.. ....** Variable (V). **11.. ....** Undefined (U). **..1. ....** Track overflow (T). **...1 ....** Blocked (B). **.... 1...** Standard blocks (S). **.... .1..** ASCII printer characters (A). **.... ..1.** Machine control characters (M). **.... ...0** Reserved bit. Should be set to zero.
1	DAIOPTCD	This field contains the error option codes: **1... ....** Write validity check (W). **..1. ....** Chained scheduling (C). **.... 1...** ASCII translate (Q). **.... .1..** User totaling (T). **.0.0 .0.0** Reserved bits. Should be set to zero.
2	DAIBLKSI	This field contains the maximum block size.
2	DAILRECL	This field contains the logical record length.
1	DAINCP	This field contains the maximum number of READ or WRITE channel programs before check.
4		Reserved. Should be set to zero.

The fields that you do not use must be initialized to zero.

Return Codes from DAIR

DAIR returns a code in general register 15 to the calling routine. In addition, further return code information is in the DAxxCTRC field in the DAIR parameter block if the return code is 8, or in the DAxxDARC field if the return code is 12.

The DAIR return codes have the following meaning:

Table 101. Return codes from DAIR

Return code dec(Hex)	Meaning
0(0)	DAIR completed successfully.
4(4)	The parameter list passed to DAIR was not valid.

Table 101. Return codes from DAIR (continued)	
Return code dec(Hex)	**Meaning**
8(8)	An error occurred in a catalog management routine; the catalog management error code is stored in the CTRC field of the DAIR parameter block.
12(C)	An error occurred in dynamic allocation; the dynamic allocation error code is stored in the DARC field of the DAIR parameter block.
16(10)	No TIOT entries were available for use.
20(14)	The ddname requested is unavailable.
24(18)	The dsname requested is a member of a concatenated group.
28(1C)	The ddname or dsname specified is not currently allocated, or the attribute list name specified was not found.
32(20)	The requested data set was previously permanently allocated, or was allocated with a disposition of new, and was not deleted. DISP=NEW cannot now be specified.
36(24)	An error occurred in a catalog information routine (IKJEHCIR).
40(28)	The return area you provided for qualifiers was exhausted and more index blocks exist. If you require more qualifiers, provide a larger return area.
44(2C)	The previous allocation specified a disposition of DELETE for this non-permanently allocated data set. Request specified OLD, MOD, or SHR with no volume serial number.
52(34)	Request denied by installation exit.

The return codes from catalog management, which are found in the DAxxCTRC field if the register 15 return code is 8, are documented in *z/OS MVS Programming: Authorized Assembler Services Guide*.

Reason Codes from Dynamic Allocation

When a DAIR return code of 12 is returned, the codes returned in the DAxxDARC field of the DAIR parameter block are the dynamic allocation error reason codes. See *z/OS MVS Programming: Authorized Assembler Services Guide* for explanations of dynamic allocation error reason codes. In addition to those codes, which are converted from dynamic allocation codes back to the same codes which were used in previous releases, the following reason codes can also be returned:

Table 102. Reason codes from dynamic allocation	
Reason code (hexadecimal)	**Meaning**
0304	The ddname was not specified by the calling routine.
0308	The ddname specified by the calling routine was not found.
0314	Restoring ddnames, as per this request, would have resulted in duplicate ddnames. Duplicate ddnames are not permitted.
0318	Incorrect characters are present in the ddname provided by the caller.
031C	Incorrect characters are present in the membername provided by the caller.
0320	Incorrect characters are present in the dsname provided by the caller.

Table 102. Reason codes from dynamic allocation (continued)

Reason code (hexadecimal)	Meaning
0324	Incorrect characters are present in the SYSOUT program name provided by the caller.
0328	Incorrect characters are present in the SYSOUT form number provided by the caller.
032C	An incorrect SYSOUT class was specified by the caller.
0330	A membername was specified but the data set is not a partitioned data set.
0334	The supplied data set name exceeded 44 characters in length.
0338	The data set disposition specified by the caller is not valid.

Chapter 18. Using IKJEHCIR to retrieve system catalog information

This chapter describes how to use the catalog information routine (IKJEHCIR) to retrieve information from the system catalog.

Functions of the Catalog Information Routine

Use the catalog information routine to retrieve information from the system catalog. This information can include data set name, index name, control volume address, or volume serial number. The information can be requested from a specific user catalog, or, if no catalog is specified, the system default catalog search is used. The following kinds of information can be requested:

- The next-level qualifiers for a name
- All names having the same name as the high-level qualifier and the data set type associated with each name
- The volume serial numbers and device types associated with a name.

You can also ask for combinations of the information above.

Passing Control to the Catalog Information Routine

Your program can invoke the catalog information routine by using either the CALLTSSR or LINK macro instructions, specifying IKJEHCIR as the entry point name. However, you must first create the catalog information routine parameter list (CIRPARM) and place its address into register 1. Register 13 must contain the address of an 18-word save area.

IKJEHCIR can be invoked in either 24- or 31-bit addressing mode. However, all input passed to IKJEHCIR must reside below 16 MB in virtual storage. The caller's parameters must be in the primary address space. IKJEHCIR returns control in the same addressing mode in which it is invoked.

The output area for IKJEHCIR can be in two formats, format 1 or format 2:

- The format 1 output area provides for a 65535-byte output area. This is due to halfword length fields in the output area itself. If the amount of data retrieved is less than 65535 bytes, the format 1 output area is sufficient.
- The format 2 output area should be used if your program is requesting to retrieve all data set names (entry code, CIROPT, = X'02') and the amount of data retrieved could exceed 65535 bytes. A format 2 output area is only supported with entry code = X'02'. Your program can indicate that a format 2 output area is being passed as input by setting on the CIRWA2 bit in the parameter list. This will indicate that the length fields (ccc and CCC) in the output area are signed 32-bit numbers. Your program can test to determine whether the code supporting a format 2 work area is available by testing the DFACIR2 flag in the Data Facilities Area (DFA) control block.

The Catalog Information Routine Parameter List (CIRPARM)

The catalog information routine parameter list (CIRPARM) is shown in Table 103 on page 343.

Table 103. The catalog information routine parameter list

Offset dec(Hex)	Number of bytes	Field name	Contents or meaning
0(0)	1	CIROPT	Entry code indicating the option requested. For a description of the entry codes, see Table 104 on page 344.

Table 103. The catalog information routine parameter list (continued)

Offset dec(Hex)	Number of bytes	Field name	Contents or meaning
1(1)	1	CIRFLAGS	Processing Flags:
		CIRWA2	1... .... indicates user work area format (on) - Format 2 user work area (off) - Format 1 user work area
2(2)	1		Reserved.
3(3)	1	CIRLOCRC	LOCATE return code.
4(4)	4	CIRSRCH	Address of the search argument. For entry codes X'01' and X'02', the search argument is either a prefix (user ID) or a name of the form *prefix.user-supplied-name*. In this case, the search argument is *not* a fully-qualified TSO/E data set name. For entry code X'04', the search argument is a fully-qualified data set name. For entry codes X'05' and X'06', the search argument is a prefix, optionally followed by a period. For information on data set naming conventions for TSO/E, see *z/OS TSO/E User's Guide*.
8(8)	4	CIRVOL	Address of the USERCAT name. The name is 44 bytes long with blanks padded to the right.
12(C)	4	CIRWA	Address of the user work area. See Table 105 on page 345 or Table 106 on page 345 for a description of the user work area.
16(10)	4		Reserved.
20(14)	4	CIRPSWD	Address of an 8-byte data set or catalog password (or zero).

Output from the Catalog Information Routine

The catalog information routine returns the requested information to the caller in a user work area that is based on CIRWA. The data that is returned for each entry code value is described in Table 104 on page 344.

Table 104. The data returned for each entry code

Entry code	Meaning	Data returned
X'01'	Retrieve the data set names having one more level of qualifier above what the caller specified.	Nine bytes of data per data set name. Each entry in the list contains a 1-byte prefix, which can be ignored, followed by an 8-byte qualifier.
X'02'	Retrieve all data set names.	45-byte data set names are moved into the user work area.
X'04'	Retrieve the volume information associated with a given data set name.	Volume information is moved into the user work area. See Table 107 on page 346 for volume information format.
X'05'	Retrieve the next level data set name and volume information. (Excludes data set names with no volume information, for example, cluster, and GDG base.)	A list of variable-length entries, one entry per data set name. Each entry in the list contains a 1-byte prefix, which can be ignored, followed by the 8-byte qualifiers making up the data set name (including periods if they occur), and then volume information. See Table 107 on page 346 for the format of the volume data.

Table 104. The data returned for each entry code (continued)

Entry code	Meaning	Data returned
X'06'	Retrieve all data set names and volume information. (Excludes data set names with no volume information, for example, cluster, and GDG base.)	45-byte data set name followed by volume information is moved to the user work area for all levels.

Note: For codes X'02' and X'06', a 1-byte field precedes a 44-byte name field. The type field has one of the following values:

- V for volume
- C for cluster
- G for alternate index
- R for path
- F for FREE
- Y for upgrade
- B for GDG base
- X for alias name
- P for page space
- M for master catalog
- U for user catalog
- A for non-VSAM data set
- D for data component
- I for index component

A format 1 user work area that is based on CIRWA is shown in Table 105 on page 345.

Table 105. Format 1 user work area for CIRPARM

Number of bytes	Field name	Contents or meaning
2	AREALN	Length of work area (an unsigned, 16-bit number).
2	DATALIN	Length of data returned +4 (an unsigned, 16-bit number).
Variable	DATA	An array of entries where data is stored. Each entry consists of a 1-byte type field followed by a 44-byte name field. The array has an end indicator of X'FF'.

A format 2 user work area that is based on CIRWA is shown in Table 106 on page 345.

Table 106. Format 2 user work area for CIRPARM

Number of bytes	Field name	Contents or meaning
4	AREALN2	Length of work area (a signed, 32-bit number).
4	DATALIN2	Length of data returned +8 (a signed, 32-bit number).
Variable	DATA	An array of entries where data is stored. Each entry consists of a 1-byte type field followed by a 44-byte name field. The array has an end indicator of X'FF'.

When you specify a data set name, a volume list is built in your work area. A volume list consists of an entry for each volume on which part of the data set resides; it is preceded by a 1-byte field that contains a count of the number of volumes in the list. The count field is followed by a variable number of 12-byte entries, with one entry for each volume. Each 12-byte entry consists of a four-byte device code, a six-byte volume serial number, and a two-byte sequence number. As many as 255 of these 12-byte entries can be

built in your work area. The volume list has an end indicator of X'FF'. Table 107 on page 346 shows the format of the volume list.

Table 107. Volume information format

Number of bytes	Field name	Contents or meaning
1		Number of volumes on which part of the data set resides.
4	DEVTYP	Device type.
6	VOLSER	Volume serial number.
2	FILESEQ	File sequence number. (This field is provided for compatibility with the OS/VS catalog, and is used for non-VSAM data sets that reside on tape volumes.)

Return Codes from IKJEHCIR

When IKJEHCIR returns to its caller, register 15 contains one of the following return codes:

Table 108. Return codes from IKJEHCIR

Return code dec(Hex)	Meaning
0(0)	The request was successfully completed.
4(4)	The LOCATE macro instruction has failed. The LOCATE return code is stored in CIRLOCRC.
12(C)	Volumes were returned by LOCATE, indicating that a fully-qualified data set name was passed in the parameter list, but options other than volumes were requested. The list of the volumes returned by LOCATE is in the work area.

Return Codes from LOCATE

The LOCATE return codes have the following meaning:

Table 109. Return codes from LOCATE to IKJEHCIR

Return code dec (Hex)	Meaning
0(0)	The request was successfully completed.
4(4)	The required catalog does not exist, it cannot be opened, or there is a closed chain of user catalog pointers.
8(8)	One of the following occurred: • The entry was not found. Register 0 contains the catalog return code. • The user is not authorized to perform this operation. Register 0 contains hexadecimal 38. • A generation data group (GDG) alias was found. Register 0 contains the number of valid index levels. The alias name was replaced by the true name.
12(C)	Obsolete. No longer possible.
16(10)	Obsolete. No longer possible.
20(14)	A syntax error exists in the name.

Table 109. Return codes from LOCATE to IKJEHCIR (continued)

Return code dec (Hex)	Meaning
24(18)	One of the following occurred: • Permanent I/O error occurred. Register 0 contains the catalog return code. • Non-zero ESTAE return code. • Error in parameter list.
28(1C)	Obsolete. No longer possible.
44(2C)	The length of the work area (AREALN) is not large enough to contain the output data returned by LOCATE.
Note: Register 0 data described above will NOT be returned to the caller of IKJEHCIR.	

For additional LOCATE return codes, see the description of message IDC3009I in *z/OS MVS System Messages, Vol 6 (GOS-IEA)*.

Chapter 19. Constructing a fully-qualified data set name with IKJEHDEF

This chapter describes how to use the default service routine (IKJEHDEF) in a Command Processor to construct a fully-qualified data set name.

Functions of the Default Service Routine

Your Command Processor can use the default service routine when a terminal user refers to a data set without giving a fully-qualified name. The default service routine constructs a fully-qualified data set name from a partially-qualified name that it receives from a Command Processor. A fully-qualified data set name has three fields: a user ID, a data set name, and a descriptive qualifier. IKJEHDEF prefixes the user ID to the data set name, checks the data set name against the system catalog, and if necessary, either inserts the proper qualifier or prompts the terminal user to enter a qualifier.

Passing Control to the Default Service Routine

Your Command Processor can invoke the default service routine by using either the CALLTSSR, LINK, or CALL macro instruction, specifying IKJEHDEF as the entry point name. However, you must first create the default parameter list (DFPL) and place its address into register 1. Register 13 must contain the address of an 18–word save area.

IKJEHDEF can be invoked in either 24- or 31-bit addressing mode. However, all input passed to IKJEHDEF must reside below 16 MB in virtual storage. The caller's parameters must be in the primary address space. IKJEHDEF returns control in the same addressing mode in which it is invoked.

The Default Parameter List (DFPL)

At entry to IKJEHDEF, register 1 must point to a default parameter list that you have built. The addresses of the user profile table, environment control table and event control block can be obtained from the Table 4 on page 15 (CPPL) that the TMP passes to your Command Processor.

The default parameter list (DFPL) is shown in Table 110 on page 349. You can use the IKJDFPL mapping macro, which is provided in SYS1.MACLIB, to map the fields of the DFPL.

Table 110. The default parameter list

Offset dec(Hex)	Number of bytes	Field name	Contents or meaning
0(0)	4	DFPLUPT	The address of the user profile table (UPT).
4(4)	4	DFPLECT	The address of the environment control table (ECT).
8(8)	4	DFPLECB	The address of the command processor's event control block (ECB).
12(C)	4	DFPLDFPB	The address of the default parameter block (DFPB).

The Default Parameter Block (DFPB)

The fourth word of the default parameter list must contain a pointer to the default parameter block (DFPB) built by the calling routine.

The default parameter block (DFPB) is shown in Table 111 on page 350. You can use the IKJDFPB mapping macro, which is provided in SYS1.MACLIB, to map the DFPB.

Table 111. The default parameter block

Offset dec(Hex)	Number of bytes	Field name	Contents or meaning
0(0)	4	DFPBDSN	The high-order byte of this field (DFPBCODE) is the entry code indicating the option requested. Table 112 on page 351 describes the options and their meanings. The remaining three bytes contain the address of the data set name buffer. Your Command Processor must build a data set name buffer that contains the length of the unqualified data set name in the first two bytes followed by the data set name that was entered by the terminal user. If the data set name is less than 44 bytes in length, it must be left justified and padded on the right with blanks.
4(4)	4	DFPBPSCB	The high-order byte of this field (DFPBCNTL) contains control codes that your Command Processor sets to indicate the functions requested. **Code** **Meaning** **DFPBUID (X'20')** The user ID is to be prefixed to the data set name. **DFPBRET (X'04')** Return a copy of the added qualifier. A copy of this qualifier is stored in the location pointed to by the DFPBQUAL field. **DFPBADD (X'02')** Add the qualifier supplied by the terminal user, which is pointed to by the DFPBQUAL field. **DFPBMSG (X'01')** Issue a message to the terminal user. The remaining three bytes contain the address of the protected step control block (PSCB). You can obtain this address from the Command Processor parameter list (CPPL) that the TMP passes to your Command Processor.
8(8)	4	DFPBQUAL	The high-order byte (DFPBLOCR) contains the LOCATE return code. The remaining three bytes contain the address of the default qualifier.
12(C)	4	DFPBCAT	The address of the user catalog.
16(10)	4	DFPBPSWD	The address of the password.

Your Command Processor must specify an entry code in the DFPBCODE field of the DFPB to specify the functions requested. Table 112 on page 351 describes the entry codes.

Table 112. The default service routine entry codes

Entry code	Meaning	Functions performed by IKJEHDEF
X'00'	Use the qualifier provided by the caller.	Uses the qualifier in the DFPB that is provided by the caller.
X'04'	Find a qualifier. If there is more than one, prompt the terminal user to choose one.	Performs the following functions: 1. Builds a list of possible qualifiers. 2. Prompts the terminal user to choose one. 3. Checks the terminal user's response against the list.
X'08'	Find a descriptive qualifier, but do not interrupt the terminal user.	Performs the following functions: • Builds a list of possible qualifiers. • Returns control to the caller with a return code indicating that more than one qualifier was found; therefore, prompting is necessary.
X'0C'	Either use the qualifier specified in the DFPB, find one from the system catalog, or use a new one submitted by the terminal user.	Does one of the following: • If a qualifier is provided in the DFPB, IKJEHDEF uses it. • If no qualifier is provided: 1. Builds a list of possible qualifiers. 2. Sends list to terminal. 3. Prompts terminal user to choose one from the list or submit a new one.

Note: Entry codes X'80', X'84', X'88', and X'8C' are the same as X'00', X'04', X'08', and X'0C' respectively, except that a catalog name and a password are obtained from the DFPB when X'80', X'84', X'88', or X'8C' are specified. For entry codes X'00', X'04', X'08', and X'0C', the system catalog is searched.

Output from the Default Service Routine

The default service routine returns the fully-qualified data set name to the caller in the data set name buffer, which is pointed to by DFPBDSN. The first two bytes of the data set name buffer are set by IKJEHDEF to the length of the fully-qualified data set name. The following bytes contain the fully-qualified data set name in the form:

```
USERID.DSNAME.QUALIFIER
```

Return Codes from IKJEHDEF

When IKJEHDEF returns to its caller, register 15 contains one of the following return codes:

Table 113. Return codes from IKJEHDEF

Return code dec(Hex)	Meaning
0(0)	Successful completion of the request.
4(4)	Unable to obtain a qualifier from the terminal user.

Table 113. Return codes from IKJEHDEF (continued)

Return code dec(Hex)	Meaning
8(8)	With qualifiers added, the length of the data set name exceeds 44 bytes.
12(C)	One of the following occurred: • A permanent I/O error occurred in the system catalog. • The catalog data set is not available. • There was a syntax error in the data set name.
16(10)	The data set exists at some level of index other than the lowest level specified.
20(14)	One of the data set names was not found.
24(18)	An attention interruption occurred.
28(1C)	An incorrect parameter was specified for one of the following reasons: • The entry code was not valid. • The data set length was not halfword aligned. • The data set length was greater than 44 bytes, or the data set length was 0 (except with an entry code of X'00'.)
32(20)	Prompting is necessary to qualify the data set name.
36(24)	No qualifiers were found.

Chapter 20. Using the DAIRFAIL routine IKJEFF18

This chapter describes how to use the DAIRFAIL routine to analyze return codes from dynamic allocation (SVC 99) or the dynamic allocation interface routine (DAIR).

Functions of DAIRFAIL

The DAIRFAIL routine analyzes return codes from SVC 99 or DAIR, and performs one of the following functions, as requested:

- Issues an error message when appropriate.
- Returns the error message to the caller.
- Issues an error message and returns the message to the caller.

This process of returning the message(s) to the caller is referred to as extracting the message.

DAIRFAIL issues a message using write-to-programmer (WTP) or PUTLINE. You can indicate to DAIRFAIL what service is to be used to issue the message, or you can allow the default, PUTLINE, to be used. Issuing a write-to-programmer message is especially useful for analyzing errors in a batch invocation of SVC 99.

Passing Control to DAIRFAIL

Your program can invoke the DAIRFAIL routine by using the LINK macro instruction, specifying IKJEFF18 as the entry point name. However, you must first create the parameter list and place its address into register 1.

DAIRFAIL can be invoked in either 24- or 31-bit addressing mode. The caller's parameters must be in the primary address space. When invoked in 31-bit addressing mode, DAIRFAIL accepts input that resides above 16 MB in virtual storage.

The Parameter List

Use the IKJEFFDF macro to map the parameter list for IKJEFF18. This mapping macro, which is provided in SYS1.MACLIB, has the following syntax:

```
IKJEFFDF   [DFDSECT={YES} ]
           [        {NO } ]

           [,DFDSEC2={YES} ]
           [         {NO } ]
```

DFDSECT=YES | NO
Use the DFDSECT=YES option to map the DFDSECTD DSECT, instead of obtaining storage. DFDSECT=NO is the default.

DFDSEC2=YES | NO
Use the DFDSEC2=YES option to map the DFDSECT2 DSECT, instead of obtaining storage. DFDSEC2=NO is the default.

The IKJEFFDF macro generates the following six-word parameter list:

Table 114. The parameter list (DFDSECTD DSECT)

Offset dec(Hex)	Field name	Contents or meaning
0(0)	DFS99RBP or DFDAPLP	Address of the failing SVC 99 request block or address of the failing DAIR parameter list.
4(4)	DFRCP	Address of a fullword containing either the SVC 99 or DAIR return code.

Table 114. The parameter list (DFDSECTD DSECT) (continued)

Offset dec(Hex)	Field name	Contents or meaning
8(8)	DFJEFF02	Address of a fullword. The fullword contains the entry point address of IKJEFF02 (message issuer routine). If the entry point address of IKJEFF02 is unknown, the fullword must contain zeros.
12(C)	DFIDP	Address of a two-byte area containing: **Byte 1** **Switches** **Bit 0:** 0 - PUTLINE issued **Bit 0:** 1 - WTP issued **Bit 1:** 1 - Caller wants message extracted only. **Bit 2:** 1 - Caller wants message extracted as well as issued using PUTLINE or write-to-programmer (WTP). **Byte 2** Caller identification number X'01' - DAIR X'32' - SVC 99 X'33' - SVC 99 invoked by the FREE command
16(10)	DFCPPLP	Address of the CPPL. This is needed only when IKJEFF18 is called with an SVC 99 error and the user is not requesting a write-to-programmer message.
20(14)	DFBUFP	Address of DFBUFS buffer if bit 2 (DFBUFSW) or bit 3 (DFBUFS2) of DFIDP is on. This is required when the message is to be extracted and returned to the caller. If the DFBUFSW is on, the message(s) will only be extracted. If DFBUFS2 is on, the message(s) will be issued as well as extracted and returned to the caller. It will be possible to extract the first-level and one second-level message.

DFDSECT2, which is described in Table 115 on page 354, defines a storage area supplied by the caller. DAIRFAIL will return the requested informational message(s) in the associated buffers. It is not necessary to initialize these buffers. On return from DAIRFAIL, the buffers will contain the extracted message(s).

Table 115. The parameter list (DFDSECT2 DSECT)

Offset dec(Hex)	Field name	Contents or meaning
0(0)	DFBUFS or DFBUFL1	A 2-byte field that will contain the total length of the first-level message, plus 4 bytes for length and offset fields.
2(2)	DFBUF01	A 2-byte field containing the offset field. It will be set to zero when a message is extracted.
4(4)	DFBUFT1	A 251-byte buffer that will contain the text of the first-level message extracted. If the message is greater than 251 bytes, the message will be truncated.
256(100)	DFBUFL2	A 2-byte field containing the total length of the first second-level message plus four bytes. If there is no second-level message, this field will contain HEX zeros.
258(102)	DFBUF02	A 2-byte field containing the offset. It will be set to zero when a message is extracted.
260(104)	DFBUFT2	A 251-byte field that will contain the text of the first second-level message extracted. If the message is greater than 251 bytes, the message will be truncated.

If the high-order bit of the caller identification area (pointed to by DFIDP) is on, a write-to-programmer message will be issued instead of a PUTLINE. When the write-to-programmer feature is used, the address of the CPPL (DFCPPLP) need not be specified.

Return Codes from DAIRFAIL

When DAIRFAIL returns to its caller, register 15 contains one of the following return codes:

Table 116. Return codes from DAIRFAIL

Return code dec(Hex)	Meaning
0(0)	A message was issued successfully.
4(4)	An incorrect caller identification number was passed to DAIRFAIL.
8(8)	The message writer detected an error while attempting to issue a message.
12(C)	The extracted message buffer parameter list is in error.

Chapter 21. Analyzing error conditions with GNRLFAIL/VSAMFAIL

This chapter describes how to use the GNRLFAIL/VSAMFAIL routine (IKJEFF19) to analyze error conditions and issue appropriate error messages.

Functions of GNRLFAIL/VSAMFAIL

The GNRLFAIL/VSAMFAIL routine analyzes VSAM macro instruction failures, subsystem request (SSREQ) failures, Parse Service Routine or PUTLINE failures, and abend codes, and issues an appropriate error message. It inserts the meaning of return codes from the VSAM/job entry subsystem interface. Other VSAM codes are explained in *z/OS DFSMS Macro Instructions for Data Sets*.

Passing Control to GNRLFAIL/VSAMFAIL

Your program can invoke the GNRLFAIL/VSAMFAIL routine by using the LINK macro, specifying IKJEFF19 as the entry point name. However, you must first create the parameter list, then place the address of the parameter list into a fullword, and then place the address of the fullword into register 1.

GNRLFAIL/VSAMFAIL can be invoked in either 24- or 31-bit addressing mode. The caller's parameters must be in the primary address space. When invoked in 31-bit addressing mode, GNRLFAIL/VSAMFAIL can be passed input that resides above 16 MB in virtual storage.

The Parameter List

The GNRLFAIL/VSAMFAIL routine returns a single parameter that contains the requested diagnostic information. You can use the IKJEFFGF macro, which is provided in SYS1.MACLIB, to map this diagnostic information. Specify the GFDSECT=YES option to map the GFDSECTD DSECT instead of obtaining storage; GFDSECT=NO is the default.

The IKJEFFGF macro generates the following diagnostic information:

Table 117. Diagnostic information returned by GNRLFAIL/VSAMFAIL (GFDSECTD DSECT)

Offset dec(Hex)	Field name	Contents or meaning
0(0)	GFCBPTR	Pointer to VSAM ACB if GFOPEN or GFCLOSE callerid. Pointer to VSAM RPL for other VSAM macro failures. Pointer to SSOB if GFSSREQ caller id.
4(4)	GFRCODE	Error return code from register 15 or ABEND code if GFCALLID is GFABEND.
8(8)	GF02PTR	Zero, or address of TSO/E message issuer routine (IKJEFF02) if already loaded.

Table 117. Diagnostic information returned by GNRLFAIL/VSAMFAIL (GFDSECTD DSECT) (continued)

Offset dec(Hex)	Field name	Contents or meaning
12(C)	GFCALLID	ID for caller's failing VSAM macro, or other failure. This field can have the following values: **Value** **Meaning** **GFCHECK (X'0001')** VSAM CHECK macro error **GFCLOSE (X'0002')** VSAM CLOSE macro error **GFENDREQ (X'0003')** VSAM ENDREQ macro error **GFERASE (X'0004')** VSAM ERASE macro error **GFGET (X'0005')** VSAM GET macro error **GFOPEN (X'0006')** VSAM OPEN macro error **GFPOINT (X'0007')** VSAM POINT macro error **GFPUT (X'0008')** VSAM PUT macro error **GFPARSE (X'0015')** Parse Service Routine error, other than a return code of 4 or 20. **GFPUTL (X'0016')** PUTLINE service routine error **GFABEND (X'001F')** Issue ABEND message **GFSSREQ (X'0020')** Subsystem interface request (SSREQ) error
14(E)	GFBITS	Special processing switches. This field can have the following values: **Value** **Meaning** **GFKEYN08 (X'80')** Caller not in key 0 or 8. **GFSUBSYS (X'40')** Caller used VS2 VSAM/job entry subsystem interface. **GFWTPSW (X'20')** Issue error message as write-to-programmer instead of PUTLINE.
16(10)	GFCPPLP	Pointer to TMP's CPPL control block (needed if PUTLINE is issued, or to have command name inserted in the failure message).
20(14)	GFECBP	Pointer to ECB for PUTLINE (optional).
24(18)	GFDSNLEN	Length of data set name.
26(1A)	GFPGMNL	Length of program name.
28(1C)	GFDSNP	Pointer to data set name to insert in VSAMFAIL error messages (optional; default is ddname).
32(20)	GFPGMNP	Pointer to program name for insertion in all error messages (optional; default is ddname).

Return Codes from GNRLFAIL/VSAMFAIL

When GNRLFAIL/VSAMFAIL returns to its caller, register 15 contains one of the following return codes:

Table 118. Return codes from GNRLFAIL/VSAMFAIL

Return code dec(Hex)	Meaning
0(0)	The message was issued successfully.
80(50)	The input parameter list for IKJEFF19 is not valid. A message is also issued.
Other	This error return code is from either PUTLINE, PUTGET or the message issuer routine (IKJEFF02).

Chapter 22. Using the table look-up service IKJTBLS

This chapter describes how an application program can use the table look-up service to search the lists of authorized commands and programs and commands not supported in the background.

Functions of IKJTBLS

Use the table look-up service (IKJTBLS) to determine if the name of a command or program is present in one of the following lists:

- Names of authorized command processors that the terminal monitor program executes
- Names of authorized programs that the CALL command executes
- Names of authorized programs that can be invoked by the TSO/E Service Facility (IKJEFTSR)
- Names of commands not supported in the background

These lists, which are maintained by your installation, allow users to issue authorized commands and programs, and restrict users from executing certain commands in background jobs.

You can use IKJTBLS in an unauthorized program to determine whether a particular command or program is authorized. Based on whether the command or program is authorized, your program can determine if the command or program must be invoked through the TSO/E Service Facility (IKJEFTSR). Usually only authorized programs can invoke an authorized command or program. However, the TSO/E Service Facility allows *any* program to invoke an authorized command or program. The TSO/E Service Facility is described in Chapter 23, “Using the TSO/E Service Facility IKJEFTSR,” on page 365.

Passing Control to IKJTBLS

Your program can invoke the table look-up service by using either the CALLTSSR or LINK macro instructions, specifying IKJTBLS as the entry point name.

However, you must first create the IKJTBLS parameter list and place its address into general register 1. Figure 139 on page 361 shows the standard parameter list structure for IKJTBLS.

IKJTBLS can be invoked in either 24- or 31-bit addressing mode. IKJTBLS accepts input above or below 16 MB in virtual storage. The caller's parameters must be in the primary address space.

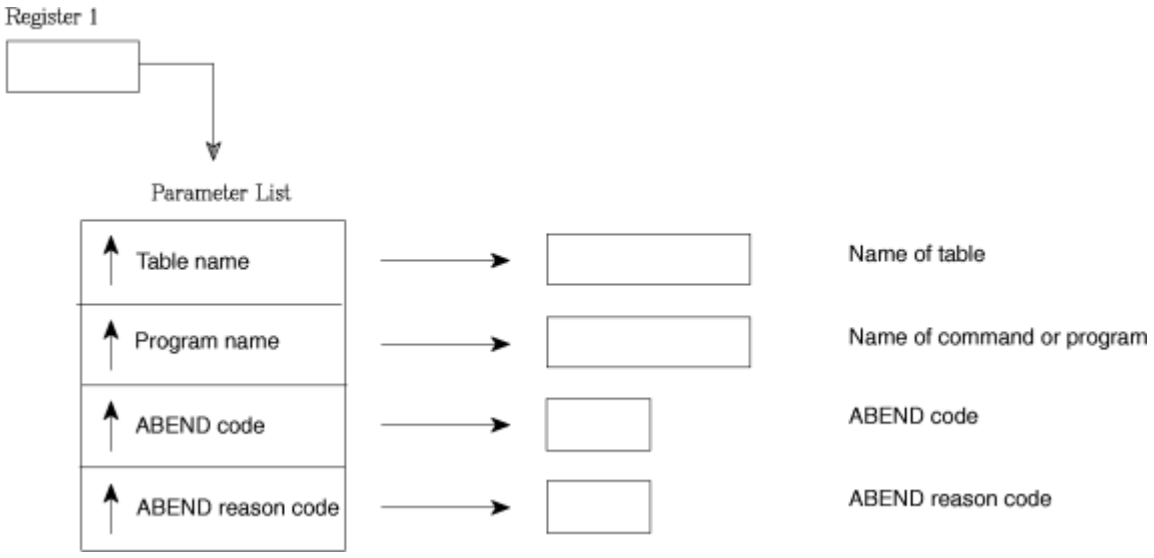


Figure 139. Parameter List Structure for IKJTBLS

The IKJTBLS Parameter List

Use the IKJTLS macro, provided in SYS1.MACLIB, to map the parameter list for IKJTBLS.

TLSTAB
An 8-byte character string that indicates the table to be searched. Set the contents of this 8-byte field to one of the following EBCDIC values:

AUTHCMD
Search the table of authorized commands.

AUTHPGM
Search the table of programs that are authorized when invoked via the CALL command.

AUTHTSF
Search the table of programs that are authorized when invoked through the TSO/E Service Facility.

NOTBKGND
Search the table of commands not supported in the background.

TLSCMD
An 8-byte character string that contains the name of the program or command to search for. Set the contents of the 8-byte field to the EBCDIC representation of the program or command name. If the name of the program or command is less than eight characters, either it must be left- justified if the table was built from a SYS1.PARMLIB member or it must appear exactly as it does in the table built using the CSECT.

TLSABND
A fullword containing the hexadecimal abend code issued by IKJTBLS. If IKJTBLS returns to its caller with a return code of 20 (decimal), this field contains the abend code. For all other return codes, IKJTBLS sets this field to zero.

TLSREAS
A fullword containing the hexadecimal abend reason code issued by IKJTBLS. If IKJTBLS returns to its caller with a return code of 20 (decimal), this field contains the abend reason code. For all other return codes, IKJTBLS sets this field to zero.

Return Codes from IKJTBLS

When IKJTBLS returns control to its caller, general register 15 contains one of the following return codes:

Table 119. Return codes from IKJTBLS

Return code dec(Hex)	Meaning
0(0)	Successful completion. The command or program was found in the specified table.
4(4)	Successful completion. The command or program was *not* found in the specified table.
8(8)	Unsuccessful completion. The specified table does not exist.
20(14)	Unsuccessful completion. An error occurred while processing the request. IKJTBLS passes the abend code and abend reason code to its caller in the TLSABND and TLSREAS fields of the parameter list.

Example Using IKJTBLS

Figure 140 on page 363 is an example showing how to invoke IKJTBLS. The segment of assembler code shown sets up the parameter list for IKJTBLS and invokes IKJTBLS using the CALLTSSR macro instruction.

```
*********************************************************************
*                                                                   *
*MODULE-NAME - TLSSAMP                                              *
*                                                                   *
*DESCRIPTION - TSO/E TABLE LOOK-UP SERVICE EXAMPLE                  *
*                                                                   *
*FUNCTION/OPERATION = THIS MODULE INVOKES THE TSO/E                 *
*        TABLE LOOK-UP SERVICE TO SEE IF THE USER-WRITTEN           *
*        COMMAND USERCMD IS FOUND IN THE AUTHORIZED COMMAND TABLE.  *
*                                                                   *
*                                                                   *
*PROCESSOR = HIGH LEVEL ASSEMBLER                                   *
*                                                                   *
*                                                                   *
*ATTRIBUTES =                                                       *
*  STATE = PROBLEM                                                  *
*  AMODE = 31                                                       *
*  RMODE = ANY                                                      *
*  KEY = 8                                                          *
*  TYPE = REFRESHABLE                                               *
*                                                                   *
*********************************************************************
*
TLSSAMP  CSECT
TLSSAMP  AMODE 31
TLSSAMP  RMODE ANY
       STM  R14,R12,12(R13)         SAVE CALLER'S REGISTERS
       BALR R12,0                   ESTABLISH ADDRESSABILITY WITHIN
       USING *,R12                  THIS CSECT
       ST R13,SAVEAREA+4            PERFORM SAVE AREA CHAINING
       LA R11,SAVEAREA
       ST R11,8(,R13)
       LA R13,SAVEAREA
@MAIN  EQU *
```

```
*********************************************************************
*                                                                   *
*  SET UP THE PARAMETER LIST FOR IKJTBLS USING THE IKJTLS MAPPING   *
*  MACRO.                                                           *
*                                                                   *
*********************************************************************
       LA R2,AUTHCMD                SEARCH THE AUTHORIZED COMMAND
       ST R2,TLSPTAB                TABLE (IKJEFTE2)
       LA R2,USERCMD                SEARCH FOR A PROGRAM NAME
       ST R2,TLSPCMD                CALLED "USERCMD"
       LA R2,TLSABND
       ST R2,TLSPABND               PASS FIELD FOR ABEND CODE
       LA R2,TLSREAS
       ST R2,TLSPREAS               PASS FIELD FOR ABEND REASON CODE
       LA R1,TLSPARM                REGISTER 1 POINTS TO PARAMETER LIST
       CALLTSSR EP=IKJTBLS          INVOKE TABLE LOOK-UP SERVICE
       LTR R15,R15                  CHECK RETURN CODE FROM IKJTBLS
       BZ @OK                       IF NAME FOUND IN TABLE, BRANCH
@NOTFND EQU *
```

Figure 140. A Sample Program Using IKJTBLS

Figure of 'A Sample Program Using IKJTBLS' (**Continued**)

```
*********************************************************************
*                                                                   *
*  PROCESS NON-ZERO RETURN CODES FROM THE TABLE LOOK-UP SERVICE.    *
*  POSSIBLE RETURN CODES ARE:                                       *
*                                                                   *
*    4 - COMMAND NOT FOUND                                          *
*    8 - TABLE NOT FOUND                                            *
*   20 - ERROR IN REQUEST                                           *
*        IN THIS CASE, THE ABEND CODE AND ABEND REASON CODE FIELDS  *
*        ARE SET UP TO THE APPROPRIATE INFORMATION.                 *
*                                                                   *
*********************************************************************
        B  @DONE
@OK     EQU *
```

```
*********************************************************************
*                                                                   *
*  THE COMMAND WAS FOUND IN THE TABLE.  PERFORM APPROPRIATE ACTIONS *
*                                                                   *
*********************************************************************
@DONE  EQU *
       L  R13,4(,R13)              OBTAIN RETURN ADDRESS
       LM R14,R12,12(R13)          RESTORE REGISTERS
       SLR R15,R15
       BR  R14
```

```
*********************************************************************
*                                                                   *
* GENERAL REGISTER EQUATES                                          *
*                                                                   *
*********************************************************************
R0       EQU   0             GENERAL REGISTER 0
R1       EQU   1             GENERAL REGISTER 1
R2       EQU   2             GENERAL REGISTER 2
R3       EQU   3             GENERAL REGISTER 3
R4       EQU   4             GENERAL REGISTER 4
R5       EQU   5             GENERAL REGISTER 5
R6       EQU   6             GENERAL REGISTER 6
R7       EQU   7             GENERAL REGISTER 7
R8       EQU   8             GENERAL REGISTER 8
R9       EQU   9             GENERAL REGISTER 9
R10      EQU   10            GENERAL REGISTER 10
R11      EQU   11            GENERAL REGISTER 11
R12      EQU   12            GENERAL REGISTER 12
R13      EQU   13            GENERAL REGISTER 13
R14      EQU   14            GENERAL REGISTER 14
R15      EQU   15            GENERAL REGISTER 15
*                                                                   *
*                                                                   *
SAVEAREA DS 18F
USERCMD  DC CL8'USERCMD '
       IKJTLS
       CVT DSECT=YES
       IKJTSVT
         END
```

Chapter 23. Using the TSO/E Service Facility IKJEFTSR

This chapter provides an overview about the TSO/E Service Facility and describes how to use it in an application program to invoke commands, programs, CLISTs and REXX execs.

Overview of the TSO/E Service Facility

The TSO/E Service Facility is an interface that allows application programmers to invoke commands, programs, CLISTs, and REXX execs from within their application programs. The application programs can be written in assembler, or in a high-level language such as PL/I, COBOL, FORTRAN, or PASCAL. Application programs using the TSO/E Service Facility can be run in foreground or background TSO/E sessions.

The TSO/E Service Facility also provides a mechanism to invoke *authorized* commands, programs, or CLISTs (consisting of only authorized commands or programs) from unauthorized application programs. Usually, authorized commands, programs, or CLISTs can be invoked only from authorized environments. The TSO/E Service Facility allows you to invoke authorized commands, programs, and CLISTs *and* unauthorized commands, programs, CLISTs, and REXX execs from unauthorized application programs.

Note: REXX execs cannot be invoked authorized in either the foreground or the background.

Further, with the TSO/E Service Facility you can create a command/program invocation platform to run the invoked commands, programs, CLISTs, and REXX execs on it. The use of a command/program invocation platform bypasses some internal processing for repeated MVS task initialization and termination caused by the invocation of commands, programs, CLISTs, or REXX execs. Use of the command/program invocation platform can result in a potential performance benefit.

Using the TSO/E Service Facility, you can access ISPF services. For information about accessing ISPF variables, see *z/OS ISPF Dialog Developer's Guide and Reference*.

The TSO/E Service Facility Routines

The TSO/E Service Facility consists of three routines:

1. IKJEFTSI - TSO/E Service Facility initialization routine

   The routine IKJEFTSI creates a command/program invocation platform to run commands, programs, CLISTs, or REXX execs, invoked by the TSO/E Service Facility routine, on it. The initialization routine is optional. It is useful if you want to bypass internal processing and gain potential performance benefit.

   IKJEFTSI lets you specify a command/program invocation platform environment you want to use.

   IKJEFTSI returns a token to the calling application program that identifies the specified platform environment to the TSO/E Service Facility routine IKJEFTSR and the termination routine IKJEFTST.
2. IKJEFTSR - TSO/E Service Facility routine

   The routine IKJEFTSR executes a specified authorized or unauthorized command, program, CLIST, or REXX exec.

   An unauthorized command, program, CLIST, or REXX exec, called by IKJEFTSR, will run on the command/program invocation platform, if one has been initialized; thus gaining from the benefits described before. IKJEFTSR will use the token passed to it by the IKJEFTSI initialization routine, to identify the command/program invocation platform.

   An authorized command, program, or CLIST, called by IKJEFTSR does not use a command/program invocation platform, even if one has been initialized. It will run in its own isolated environment.

You can use IKJEFTSR multiple times to run unauthorized commands, programs, CLISTs, or REXX execs on the initialized command/program invocation platform before you terminate the platform with IKJEFTST.

3. IKJEFTST - TSO/E Service Facility termination routine

   The routine IKJEFTST cleans up and terminates the command/program invocation platform set up by IKJEFTSI. It uses the token, passed to it by the initialization routine IKJEFTSI, to identify the platform to be terminated.

The TSO/E Service Facility routines can be nested as required. The token, passed from the initialization routine IKJEFTSI to IKJEFTSR and IKJEFTST, remains valid on the same task level. Every initialized command/program invocation platform must be properly terminated with IKJEFTST.

"Using the Command/Program Invocation Platform" on page 367 provides more details on how the TSO/E Service Facility routines work together.

Program Authorization and Isolation

Commands, programs, CLISTs, or REXX execs can either be authorized or unauthorized functions to an application program. Application programs are most often unauthorized functions to the system they are running on, rarely authorized functions. For system security reasons, an authorized function can normally invoke only authorized functions.

IKJEFTSR specifically allows you to invoke authorized functions from an unauthorized application program. It maintains system security by running an invoked authorized function in its own isolated environment.

However, to maintain system security, an authorized application program can use the TSO/E Service Facility to invoke only authorized programs or commands, or CLISTs consisting of only authorized programs and commands.

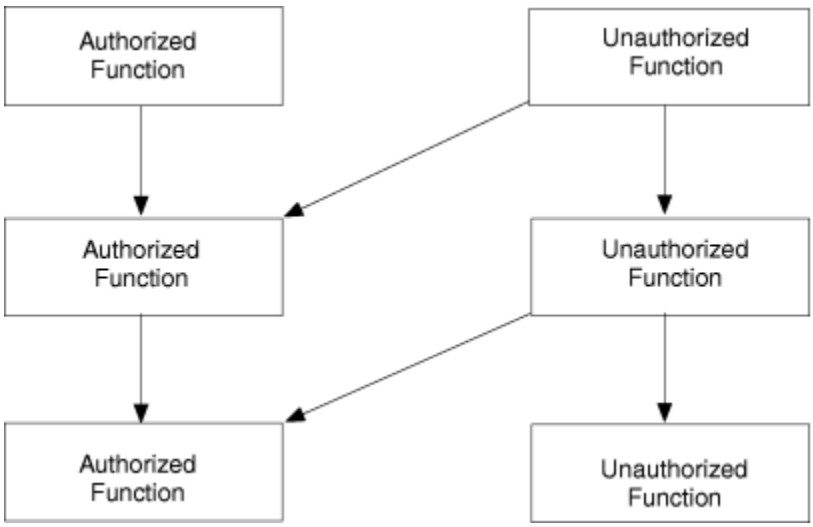


Figure 141. Invoking Authorized Functions with the TSO/E Service Facility

While the invocation of authorized functions automatically makes the TSO/E Service Facility to run the function in an isolated environment, you can specify the type of environment for the invocation of unauthorized functions (parameter 1 in "IKJEFTSR Parameter List" on page 373).

- If you want the TSO/E Service Facility to run the unauthorized function in an *unisolated environment*, the unauthorized function itself can invoke other (authorized or unauthorized) functions through IKJEFTSR. This also provides for access to ISPF services and TSO/E REXX programming services. However, after an authorized function is invoked, it is run in its own isolated environment (see below).
- If you want the TSO/E Service Facility to run the unauthorized function in an *isolated environment*, the invoked function itself can only invoke authorized functions. This makes the TSO/E Service Facility to run the requested function as an isolated subtask of the TSO terminal monitor program. The existing

environment is suspended until the requested function completes. It is the existing environment's responsibility to release any resources that may have been required by the requested function (such as serialization resources).

TSO/E determines the authorization of commands and programs and the execution environment they are running in by analyzing the following statements in member IKJTSOxx of SYS1.PARMLIB:

AUTHCMD
identifies authorized commands to TSO/E

AUTHPGM
identifies programs that are authorized when invoked via the CALL command

AUTHTSF
identifies programs that are authorized when invoked through the TSO/E Service Facility. In most cases these programs are not in AUTHPGM. They are primarily those that expect more complex parameter lists than that of the CALL command and use parameter 7 of the IKJEFTSR parmlist to supply parameters to the invoked program. As a general rule programs in this list, it should not accept parameters that are pointers to code that will be executed (such as exit routines) because this might introduce an integrity exposure.

Note: Do not place programs from any IBM products in this table unless the documentation of that product requires. For example, do not put IDCAMS in AUTHTSF.

PLATCMD
identifies authorized and unauthorized commands that can run on a command/program invocation platform.

PLATPGM
identifies authorized and unauthorized programs that can run on a command/program invocation platform.

Further details about the statements in SYS1.PARMLIB member IKJTSOxx can be found in *z/OS TSO/E Customization*.

You may want to use the table look-up service, described in Chapter 22, “Using the table look-up service IKJTBLS,” on page 361, in your application programs to determine if a program or command name is identified by one or more of these statements.

Using the Command/Program Invocation Platform

The routines IKJEFTSI and IKJEFTST set up and terminate, respectively, a specified command/program invocation platform. Routine IKJEFTSR allows eligible commands and programs to run on that platform.

Eligible commands and programs are those having an entry in SYS1.PARMLIB, member IKJTSOxx. Commands and programs are specified in this member using the PLATCMD and PLATPGM statements, respectively. Installations can add their own commands and programs to the appropriate platform statement, but they must first ensure that these commands and programs do not require the services of MVS task termination. If an application program terminates before the environment created by IKJEFTSI terminates, a system abend A03 can result. For more information about PLATCMD, PLATPGM, SYS1.PARMLIB, and about adding installation-defined commands and programs, see *z/OS TSO/E Customization*.

The following three sections describe how the TSO/E Service Facility routines interact with each other and with the application program using them. Refer also to Figure 142 on page 368.

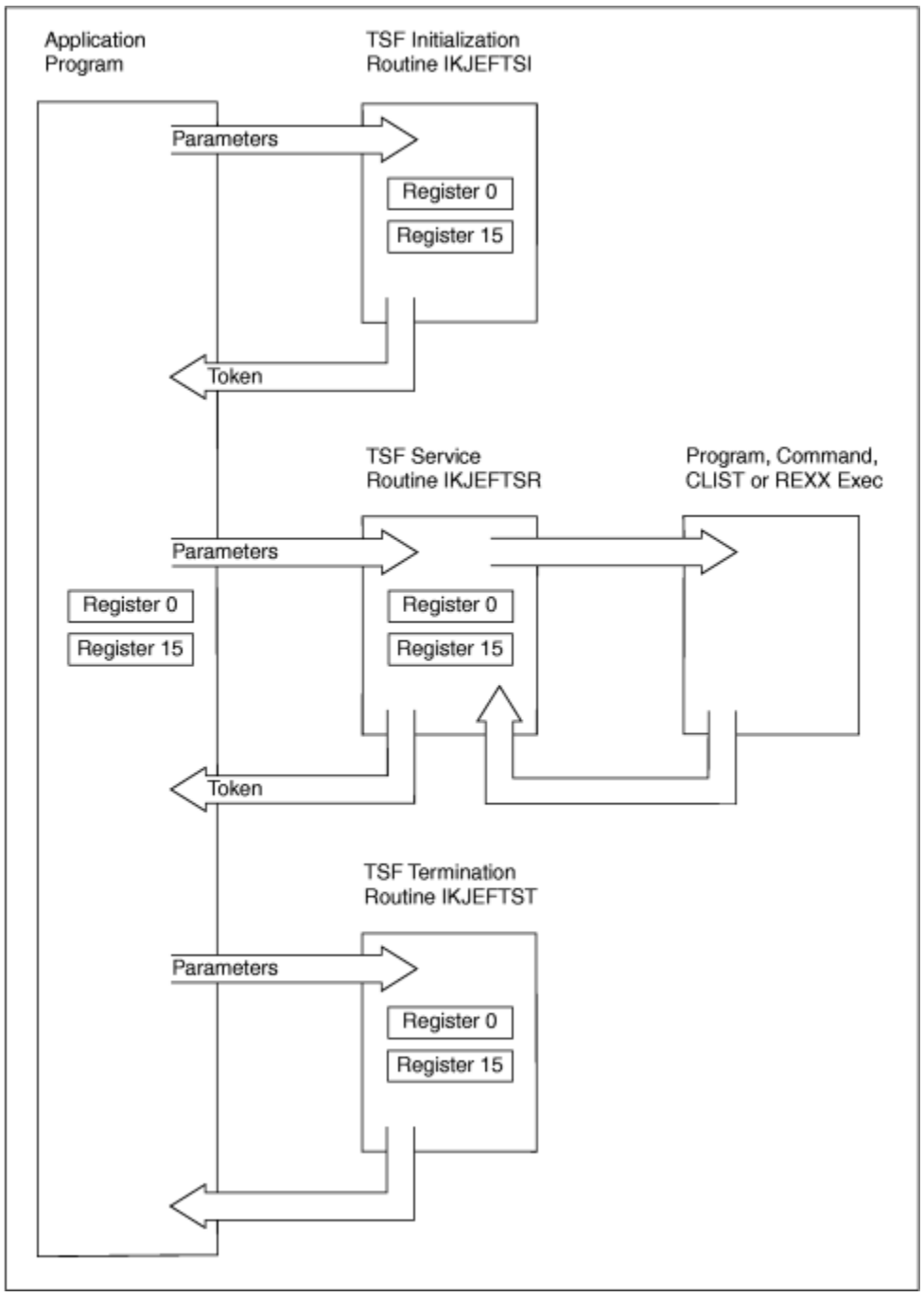


Figure 142. Interaction of the TSO/E Service Facility Routines

Creating the Platform with IKJEFTSI

Use the TSO/E Service Facility initialization routine (IKJEFTSI) or its alias (TSOLKI) to create a command/program invocation platform. This routine returns a token to the caller that identifies the command/program invocation platform that was created. This token must be passed to the TSO/E Service Facility routine (IKJEFTSR) to enable the commands or programs to execute on the command/program invocation platform. This token is valid only for calls to IKJEFTSR that occur on the same task level as that for which the initialization routine was invoked.

Executing Commands or Programs on the Platform with IKJEFTSR

Use the TSO/E Service Facility routine (IKJEFTSR) or its alias (TSOLNK) to execute eligible commands or programs on the command/program invocation platform. This routine accepts the token passed from IKJEFTSI to identify the command/program invocation environment. IKJEFTSR searches the list of eligible commands or programs that can run on the command/program invocation platform. If the command or program is eligible and located in a system library, it will be invoked on the command/program invocation platform. If it is located in a step library, it will not be invoked from the platform task, because commands or programs in a step library might not be system commands or programs intended for this kind of invocation.

Terminating the Platform with IKJEFTST

Use the TSO/E Service Facility termination routine (IKJEFTST) or its alias (TSOLKT) to clean up and terminate the command/program invocation platform. This routine accepts the token passed from IKJEFTSI and terminates the environment at the task level created by IKJEFTSI.

TSO/E Service Facility Initialization Routine IKJEFTSI

The TSO/E Service Facility initialization routine (IKJEFTSI) initializes a command/program invocation platform. It returns a token to the caller that identifies the command/program invocation platform that was created.

Passing Control to IKJEFTSI

Invoke the TSO/E Service Facility initialization routine using one of the following methods:

- The CALLTSSR macro instruction, specifying IKJTSFI as the entry point name
- The LINK macro instruction, specifying IKJEFTSI (or TSOLKI, the alias of IKJEFTSI), as the entry point name
- The address of IKJEFTSI that is in the TSVTTSFI field of the TSVT.

You must first create the IKJEFTSI parameter list and place its address into general register 1.

Standard linkage conventions are:

- Register 1 must contain the address of a parameter list.
- Register 13 must contain the address of an 18-word save area.
- Register 14 must contain the return address.
- Register 15 must contain the entry point address.

IKJEFTSI must receive control in 31-bit addressing mode. IKJEFTSI accepts input above or below 16 MB in virtual storage. The caller's parameters must be in the primary address space.

IKJEFTSI Parameter List

Use the IKJEFTSJ macro to map the parameter list for IKJEFTSI. This mapping macro is provided in SYS1.MACLIB. Use the TJDSECT=YES option to map the TJDSECTD DSECT, instead of obtaining storage.

```
IKJEFTSJ TJDSECT=YES
```

TJDSECT=NO is the default.

Figure 143 on page 370 describes the parameter list passed to the TSO/E Service Facility initialization routine (IKJEFTSI) pointed to by register 1.

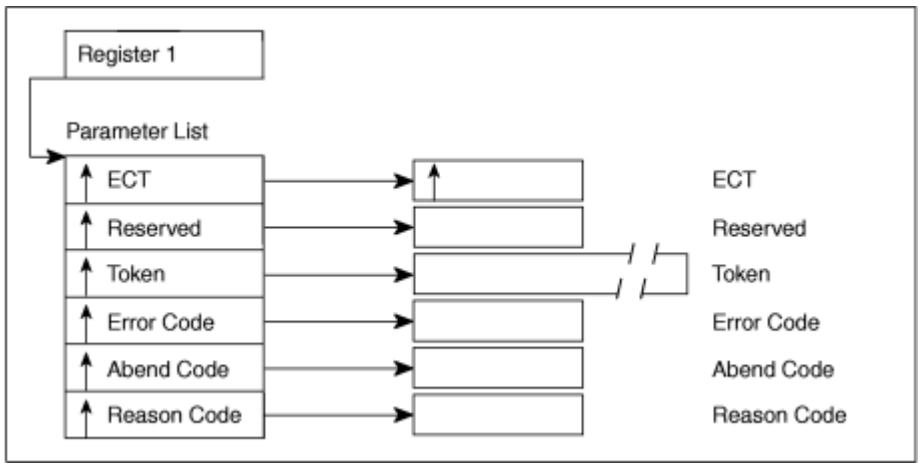


Figure 143. Parameter List for IKJEFTSI

The parameters are:

Parameter 1
IKJEFTSI (also IKJEFTSR and IKJEFTST) needs to invoke TSO/E I/O services (STACK, PUTLINE, GETLINE, PUTGET). It needs to tell these underlying services which environment control table (ECT) to use. You have a choice at this point which ECT these services are to use.

The first parameter specifies the environment control table (ECT) to be used. It can be set to a fullword of either:

- The address of the user's *current* environment control table (ECT)
- A value of X'00000000', specifying that the *original* ECT, created when your TSO/E session was initialized, is to be used.

  The address of the original ECT is placed in this parameter on return to the caller.
- A value of X'FFFFFFFF', specifying that a *new* ECT is to be created. This new ECT uses the original ECT (created when your TSO/E session was initialized) as a model.

  The address of the new ECT is placed in this parameter on return to the caller.

Note that it is important for you to pass this same ECT address in parameter 8 to IKJEFTSR so that it also uses the same ECT.

Parameter 2
The second parameter is a reserved fullword. Although this parameter is not used, it must contain X'00000000' on input.

Parameter 3
On input to IKJEFTSI this parameter must be set to four fullwords of X'00000000'.

On output from IKJEFTSI, this parameter consists of a token of four fullwords that identifies the TSO/E command/program invocation platform that is created by the initialization routine. The use of the token is an option with the calls to the TSO/E Service Facility routine (IKJEFTSR) and the TSO/E Service Facility termination routine (IKJEFTST).

Parameter 4
The fourth parameter is a fullword containing an error code if IKJEFTSI completes unsuccessfully. The error code is used with return codes of decimal 12 or 24 to give a more detailed diagnosis of the cause of the failure.

All following parameters are optional. Note that the high-order bit of the address of the last parameter used must be on to indicate the end of the parameter list.

Parameter 5
The fifth parameter is optional. It is a fullword containing the abend code returned from IKJEFTSI when terminated abnormally.

Parameter 6
The sixth parameter is optional. It is a fullword containing the reason code returned from IKJEFTSI when terminated abnormally.

Output from IKJEFTSI

IKJEFTSI passes a return code to the calling program in general register 15. For high-level languages that cannot interrogate register 15, IKJEFTSI also places the return code in general register 0.

After IKJEFTSI successfully completes (return code 0), parameter 3 contains a token that identifies the command/program invocation platform. To execute commands or programs on the command/program invocation platform, pass this token to IKJEFTSR (TSOLNK). To terminate the platform, pass the token to IKJEFTST (TSOLKT).

Return Codes from IKJEFTSI

The TSO/E Service Facility initialization routine return codes are shown in Table 120 on page 371.

Table 120. Return codes from IKJEFTSI

Return code dec(Hex)	Meaning
0(0)	TSO/E Service Facility initialization was successful: • When the ECT address (parameter 1) contains X'00000000' on input, the field is updated to contain the address of the original ECT created when your TSO/E session was initialized. • When the ECT address (parameter 1) contains X'FFFFFFFF' on input, the field is updated to contain the address of the new ECT. • The TOKEN field contains four fullwords to be passed to IKJEFTSR and IKJEFTST. • The ERROR field contains zero.
12(C)	TSO/E Service Facility initialization was unsuccessful because of inconsistent or incorrect parameters. The ERROR field shows the reason for the error: **Error code = 1 (dec)** Non-zero reserved parameter passed to IKJEFTSI. **Error code = 2 (dec)** Non-zero token parameter passed to IKJEFTSI. **Note:** The high-order bits of all parameters in the parameter list pointed to by register 1 must be off except for the last parameter. If IKJEFTSI detects this error, it sets return code = 12, but it cannot set the ERROR field.
20(14)	TSO/E Service Facility initialization was unsuccessful because of an environmental error. The ERROR field shows the reason for the error: **Error code = 20 (dec)** IKJEFTSI invoked in a non-TSO/E environment. **Error code = 21 (dec)** IKJEFTSI invoked in an authorized TSO/E environment. **Error code = 22 (dec)** Storage for a TSF environment could not be obtained. **Error code = 23 (dec)** A value of X'FFFFFFFF' was passed to IKJEFTSI in parameter 1 (ECT address), but IKJEFTSI was unable to create the new ECT.

Table 120. Return codes from IKJEFTSI (continued)	
Return code dec(Hex)	**Meaning**
92(5C)	TSO/E Service Facility initialization was unsuccessful. A recovery environment could not be established.
96(60)	TSO/E Service Facility initialization was unsuccessful. A parameter is not accessible; see the abend code and reason code parameters for the abend and reason codes.
100(64)	TSO/E Service Facility initialization was unsuccessful. Abnormal termination; see the abend code and reason code parameters for the abend and reason codes.

TSO/E Service Facility Routine IKJEFTSR

The TSO/E Service Facility routine (IKJEFTSR) allows a user to invoke functions such as commands, programs, CLISTs, or REXX execs from an application program.

Passing Control to IKJEFTSR

Invoke the TSO/E Service Facility routine (IKJEFTSR) using one of the following methods:

- The LINK macro instruction. Use this, for example, from an assembler program.

  Specify IKJEFTSR or its alias TSOLNK as the entry point name. TSOLNK is useful if the programming language you are using does not allow you to use names longer than six characters.
- The address of IKJEFTSR that is in the TSVTASF field of the TSVT

  Use this, for example, when you want to get addressability to the common copy of IKJEFTSR for all applications.
- Link-editing IKJEFTSR with your application program.

  Be aware that if you use this method and a change, for example, a release upgrade, occurs, then it is your responsibility to link-edit again to pick up the new level. For link-editing the user program, the SYSLIB concatenation must contain SYS1.LPALIB, in which TSOLNK and IKJEFTSR reside.

Standard linkage conventions are:

- Register 1 must contain the address of a parameter list.
- Register 13 must contain the address of an 18-word save area.
- Register 14 must contain the return address.
- Register 15 must contain the entry point address.

The parameter list pointed to by register 1 consists of a list of addresses. Each address points to a parameter. If you want to use a command, program, CLIST, or REXX exec, you must specify the first six parameters. If you want to pass parameters to a program, you can specify an optional seventh parameter when you invoke the TSO/E Service Facility. The seventh parameter is intended for use with assembler programs. The eighth and ninth parameters are optional, and are intended for use when invoking a command, REXX exec, or CLIST in an unauthorized environment. To indicate the end of the parameter list, you must set the high-order bit of the last address to 1.

Your application program can invoke IKJEFTSR in 24-bit or 31-bit addressing mode. IKJEFTSR returns control to its caller in the same addressing mode with which it was invoked.

The caller's parameters must be in the primary address space. Input can reside above or below 16 MB in virtual storage. IKJEFTSR treats input addresses according to the addressing mode in which IKJEFTSR was invoked.

IKJEFTSR Parameter List

Figure 144 on page 373 describes the parameter list passed to the TSO/E Service Facility routine (IKJEFTSR) pointed to by register 1.

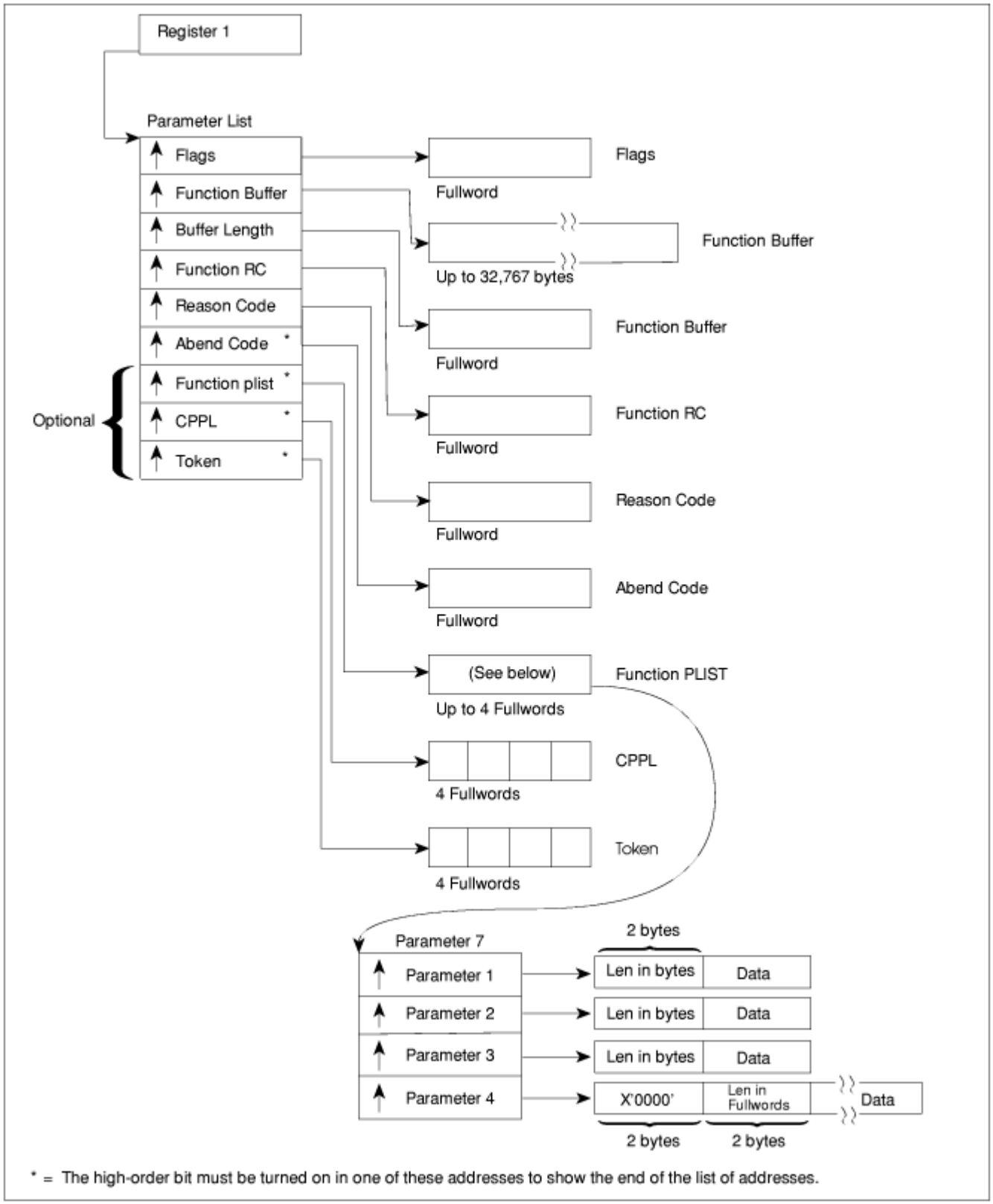


Figure 144. Parameter List for IKJEFTSR

The parameters are:

Parameter 1

This parameter contains a fullword of flags.

1. Byte 1 is all zeros.
2. Byte 2 is the internal processing options flag byte. This flag indicates whether the TSO/E service facility should establish an isolated or unisolated environment before invoking the requested function. Set byte 2 to one of the following values:
   - X'00' to show that the TSO/E service facility should invoke the requested function in an *isolated* environment.

     Use X'00' when you invoke the TSO/E service facility from an *authorized* program or command.
   - X'01' to show that the TSO/E service facility should invoke the requested function in an *unisolated* environment.

     Use X'01' when you invoke the TSO/E service facility from an *unauthorized* program or command.

   When invoked from an *unauthorized* program or command, the TSO/E service facility can invoke the requested function in either an isolated or an unisolated environment. Therefore, either value for byte 2 is accepted. However, you should consider the following:

   - Invoking the requested function in an unisolated environment by setting byte 2 to X'01' might result in a potential performance gain.
   - You must set byte 2 to X'01' when invoking commands or programs that are to run on the command/program invocation platform specified by IKJEFTSI.
   - You must set byte 2 to X'00' when invoking commands or programs that need to run in a new, clean RACF program control environment. This will enable the command or program to use RACF Program Access to Data Sets (PADS) or EXECUTE-controlled programs. See *z/OS Security Server RACF Security Administrator's Guide* for more information on these functions.
   - If your application program invokes a function that invokes ISPF services or TSO/E REXX programming services, you must indicate to the TSO/E service facility that an unisolated environment be established (X'01'). In an unauthorized environment all ISPF services and REXX services are available.

     In an isolated environment no ISPF services are available and no REXX services that depend on an environment other than the first TSO/E environment created at LOGON time (for example, you cannot get to the data stack of an ISPF environment).

   Note: If you are running in the foreground, all input/output is received from the terminal, respectively sent to the terminal. If you are running in the background, all input/output is received from SYSTSIN, respectively sent to SYSTSPRT.
3. Byte 3 is the error processing flag byte. Set byte 3 to one of the following values:
   - X'00' to show that no dump should be taken if the invoked function abends.
   - X'01' to show that a dump should be taken if the invoked function abends.
4. Byte 4 is the function flag byte. This flag indicates the type of function being invoked. Set byte 4 to one of the following values:
   - X'01' to show that a TSO/E command, REXX exec, or CLIST is being invoked. The processing of the requested function may depend on the value of byte 2 of parameter 1:
     - If the requested function is a CLIST and byte 2 of parameter 1 is set to X'00', the CLIST is placed on the TSO/E input stack and is *not* run before the return from the TSO/E service facility.
     - If the requested function is a CLIST and byte 2 of parameter 1 is set to X'01', the CLIST is placed on the TSO/E input stack and *is run* before the return from the TSO/E service facility.
   - X'02' to show that a program is being invoked.

- X'05' to show that a TSO/E command, REXX exec, or CLIST is being invoked. This value should only be used when byte 2 of parameter 1 is set to X'00'. If the requested function is a CLIST, the CLIST is placed on the TSO/E input stack and is run before the return from the TSO/E service facility.

  Note: To maintain compatibility , the TSO/E service facility interprets a value of X'05' to mean that a CLIST is being invoked if the requested function cannot be found as a command. However, use a value of X'05' only when the internal processing options flag (byte 2 of parameter 1) is set to X'00'.

For non-assembler language programmers, parameter 1 can be thought of as an integer containing the sum of all the following:

- The value of byte 2, multiplied by 65 536
- The value of byte 3, multiplied by 256
- The value of byte 4.

Parameter 2
The second parameter contains a character string containing the name of the command, program, CLIST or REXX exec being invoked. If a command is invoked, the character string must also contain all the parameters for the command. If you are invoking an authorized program, the invoked program must be in a member of a partitioned data set allocated to STEPLIB or LINKLIB. If the authorized program resides as a member in the STEPLIB concatenation, all of the data sets in the concatenation must be authorized in order for the system to give control to the program in an authorized state.

Note: The execution of the LOGON and LOGOFF commands remains pending until the program environment terminates. That is, if invoked from within a program, these commands would take effect after the program finishes, where you would ordinarily see a READY prompt. These commands provide a return code of 0 in parameter 4 if the syntax is accurate.

Parameter 3
The third parameter is a fullword containing the length of the name of the invoked command, program, CLIST or REXX exec (parameter 2). This is automatically provided in some FORTRAN compilers.

Parameter 4
The fourth parameter is an output parameter. It is a fullword containing the return code of the invoked function specified in parameter 2. If a CLIST is invoked, parameter 4 will contain the value of the CLIST variable LASTCC, or the value specified in an exit code statement.

The TSO/E service facility initially sets this parameter to -1 on entry, and resets its value to be the return code as appropriate on return.

Parameter 5
Parameter 5 is an output parameter. The meaning of it depends on the service facility return code found in register 15. If the service facility return code is 12, parameter 5 is a fullword containing the abend reason code of the invoked function. If the service facility return code is either decimal 20 or 24, parameter 5 contains the service facility reason code. If the return code is either decimal 20 or 24, save the return code and notify your IBM service representative. See Table 122 on page 378 for the meaning of the service facility reason codes.

The TSO/E Service Facility initially sets this parameter to -1 on entry, and resets its value to be the return code as appropriate on return.

Parameter 6
Parameter 6 is an output parameter. If the requested program or command ends unsuccessfully, it is a fullword containing the abend code.

The TSO/E service facility initially sets this parameter to -1 on entry, and resets its value to be the return code as appropriate on return.

All following parameters are optional. Note that the high-order bit of the address of the last parameter used must be on to indicate the end of the parameter list.

Parameter 7

The seventh parameter is optional. It is used to pass parameters to the invoked program. It is intended for use with assembler language programs. Use parameter 7 when a program (*not* a TSO/E command, CLIST, or REXX exec) is being invoked.

Set parameter 7 to a fullword containing zeros when your application program:

- Is invoking a command (as opposed to a program)
- Has set byte 2 of parameter 1 to indicate that the TSO/E service facility should establish an unauthorized environment to invoke the function, and
- Uses the eighth parameter.

If you choose to code parameter 7 to pass parameters to your program, comply with the following rules. Parameter 7 is a variable-length list consisting of the addresses of from one to four parameters. The high- order bit of the last address must be on to indicate the end of the list. Each entry in the list points to a parameter whose format is described below.

- Parameters 1-3
  - A halfword containing the parameter length in bytes, immediately followed by
  - A variable-length data string
- Parameter 4
  - A fullword containing 2 bytes of zeros, immediately followed by
  - Two bytes containing the number of fullwords of data, immediately followed by
  - A variable-length data string

The exact format of this parameter list will vary depending on the program being invoked. For most assembler programs, only the first parameter is used. Figure 144 on page 373 above shows the format of the function parameter list within the parameter list.

If parameter 7 is not coded when invoking a program, the TSO/E service facility passes one parameter to the program. Before invoking the requested program, the TSO/E service facility sets this parameter to the address of a halfword containing zeros. The high-order bit of the address of the halfword is on to indicate the end of the list.

Parameter 8

The eighth parameter is optional. It is four fullwords containing the Command Processor parameter list (CPPL). It is intended for use when requesting the TSO/E service facility to establish an unauthorized environment to invoke the function. Use parameter 8 only when byte 2 of parameter 1 is set to a value of 1.

Your choices when deciding whether to code parameter 8 are as follows:

- Choice 1: Set parameter 8 to four fullwords of zeros (that is, parameter 8 is a dummy parameter) when your application program uses the ninth parameter and you request that IKJEFTSR construct a CPPL for you.
- Choice 2: Set parameter 8 to four fullwords containing a valid CPPL that IKJEFTSR will use in the unauthorized environment. For a description of the CPPL, see “Interfacing with the TSO/E service routines” on page 13.

If you are invoking a command and use parameter 8, you must set parameter 7 to a fullword containing zeros.

Parameter 9

The ninth parameter is optional. It is four fullwords containing the token that was passed to the application program by IKJEFTSI. It is intended for use when requesting the TSO/E service facility to invoke an unauthorized command on the command invocation platform. Use parameter 9 only when byte 2 of parameter 1 is set to a value of 1.

Your choices when deciding whether to code parameter 9 are as follows:

- Choice 1: To omit parameter 9 altogether by turning on the high-order bit in either parameter 7 or 8 (as appropriate).
- Choice 2: To send a null token, set parameter 9 to four fullwords of zeroes.
- Choice 3: Use parameter 9 to specify the token from IKJEFTSI.

If you are invoking a command and use either parameter 8 or 9, you must set parameter 7 to the address of a fullword containing zeroes.

If you request that an unauthorized environment be used to invoke the function and you do not code parameter 9, the requested command or program will not execute on the command/program invocation platform.

Output from IKJEFTSR

Return Codes from IKJEFTSR

Table 121 on page 377 contains the return codes from the TSO/E Service Facility.

Table 121. Return codes from IKJEFTSR

Return code dec(Hex)	Meaning
0(0)	IKJEFTSR and the requested program, command, CLIST or REXX exec completed successfully.
4(4)	The invoked program, command, CLIST or REXX exec had a non-zero return code, which is in parameter 4.
8(8)	The invoked function was terminated because of an attention interruption. If the application programmer wants to notify the end user, the application program should issue a message.
12(C)	The invoked function terminated abnormally. The sixth parameter contains the abend code. The fifth parameter contains the reason code associated with the abend.
16(10)	One of the first 6 parameters in the parameter list contains addresses of storage not accessible to the calling program.
20(14)	The IKJEFTSR parameter list contains an error. The fifth parameter contains the reason code associated with the error.
24(18)	The TSO/E routines associated with IKJEFTSR encountered an unexpected failure. The fifth parameter contains the reason code associated with the error.
28(1C)	The caller of IKJEFTSR is executing in 24-bit addressing mode, but the parameter list contains 31-bit addresses.

Reason Codes from IKJEFTSR

Table 122 on page 378 shows the reason codes that are found in parameter 5 if IKJEFTSR completes with a return code of 20.

Table 122. Reason codes from IKJEFTSR (when return code is decimal 20)

Reason code dec(Hex)	Meaning
4(4)	The length of the parameter list is not valid. One of the following is true: • The invoker of the TSO/E Service Facility did not turn on the first bit of the last parameter to indicate the end of the list. • The high-order bit is on in any of the first five parameters. • More than nine parameters are coded.
8(8)	The first byte of the flag field pointed to by the first parameter is non-zero.
12(C)	The function flag byte (byte 4) of the flag field pointed to by the first parameter is not valid. It should contain a 1 for a command, CLIST or REXX exec, or a 2 for a program.
16(10)	The function flag byte (byte 4) of the flag field pointed to by the first parameter specified a command (contained a 1). However, a seventh parameter (program parameter list) was also coded. The seventh parameter can only be coded when the function being invoked is a program.
20(14)	The abend processing flag byte is not valid. This byte (byte 3) of the flag field pointed to by the first parameter should contain either a zero to request a dump, or a 1 to indicate no dump is to be taken.
24(18)	IKJEFTSR was invoked from a non-TSO/E environment. This service can only be used in a foreground or background TSO/E environment.
28(1C)	The function buffer length is not valid. The function buffer pointed to by the second parameter must be greater than zero and less than 32K-5.
32(20)	The program parameter list (pointed to by the seventh parameter of the TSO/E Service Facility parameter list) contains addresses of storage not accessible to the calling program.
36(24)	The program parameter list pointed to by the seventh parameter is not valid.
40(28)	The requested function (program, command, CLIST or REXX exec) was not found.
44(2C)	A syntax error in the function (program, command, CLIST or REXX exec) name was detected.
48(30)	The command name began with "%". However, CLIST processing was not requested in parameter 1.
52(34)	Unsupported background function (program or command).
56(38)	The function (either a program or command) is authorized, but a copy of the function could not be found in an authorized library.
60(3C)	One of the following occurred: • An authorized program or command requested that an unauthorized function be invoked. • An authorized program or command invoked the TSO/E Service Facility, but indicated that the requested function be invoked from an unauthorized environment. An authorized program must set the internal processing options flag (byte two of parameter one) to zero.
64(40)	An incorrect token was passed to IKJEFTSR.

Table 122. Reason codes from IKJEFTSR (when return code is decimal 20) (continued)	
Reason code dec(Hex)	**Meaning**
68(44)	The invoker of the TSF asked for parallel TMP processing under the dynamic TSO environment. Programs running in this environment cannot use the TSO/E Service Facility (IKJEFTSR) to invoke functions from an authorized environment.

Considerations on Attention Interruptions with IKJEFTSR

If you choose isolated environment TSO/E also isolates the active attention exits. If you choose unisolated environment this is not the case and you may find that the wrong attention exit gets control.

The application program issuing the TSO/E Service Facility routines may as well use the STAX macro to process attention interruptions. The STAX macro is to specify the address of an attention exit routine in your application program that gains control when an attention interruption occurs.

On the other hand, the commands, programs, CLISTs, or REXX execs invoked with the TSO/E Service Facility routine IKJEFTSR may have their own attention interruption processing.

If your application program issues its own STAX macro it may be required to temporarily relinquish its control if you want the invoked function's attention interruption processing to take control. Consider the following sequence, which shows how an application relinquishes control of its attention exit while the TSO/E Service Facility is executing:

1. The application indicates that its attention exit is to receive control:

   STAX *exit_address*, USADDR= ...

   During application processing, its attention exit will receive control if an attention interruption occurs.
2. The application relinquishes control of its attention exit before invoking IKJEFTSR:

   STAX
3. The application invokes IKJEFTSR.
4. After IKJEFTSR has finished the application reestablishes its own attention processing:

   STAX *exit_address*, USADDR= ...

If the TSO/E Service Facility is invoked and an ISPF service is invoked from within, it is required to temporarily relinquish control; else ISPF will not be able to perform its attention processing.

For further details see Chapter 12, “Using the STAX service routine to handle attention interrupts,” on page 285 and *z/OS TSO/E Programming Guide* about processing attention interruptions.

TSO/E Service Facility Termination Routine IKJEFTST

The TSO/E Service Facility termination routine (IKJEFTST) terminates the environment that IKJEFTSI creates for this task level. It is required when using the command/program invocation platform support. If an application program terminates before the environment created by IKJEFTSI terminates, a system abend A03 can result.

Passing Control to IKJEFTST

Invoke the TSO/E Service Facility termination routine using one of the following methods:

- The CALLTSR macro instruction, specifying IKJTSFT as the entry point name
- The LINK macro instruction, specifying IKJEFTST (or TSOLKT, the alias of IKJEFTST), as the entry point name
- The address of IKJEFTST that is in the TSVTTSFT field of the TSVT

You must first create the IKJEFTST parameter list and place its address into general register 1.

Standard linkage conventions are:

- Register 1 must contain the address of a parameter list.
- Register 13 must contain the address of an 18-word save area.
- Register 14 must contain the return address.
- Register 15 must contain the entry point address.

IKJEFTST executes and must receive control in 31-bit addressing mode. It accepts input above or below 16 MB in virtual storage and executes in primary address space control (ASC) mode.

IKJEFTST Parameter List

Use the IKJEFTSV macro to map the parameter list for IKJEFTST. This mapping macro is provided in SYS1.MACLIB. Use the TVDSECT=YES option to map the TVDSECTD DSECT, instead of obtaining storage.

```
IKJEFTSV TVDSECT=YES
```

TVDSECT=NO is the default.

Figure 145 on page 380 describes the parameter list passed to the TSO/E Service Facility termination routine (IKJEFTST) pointed to by register 1.

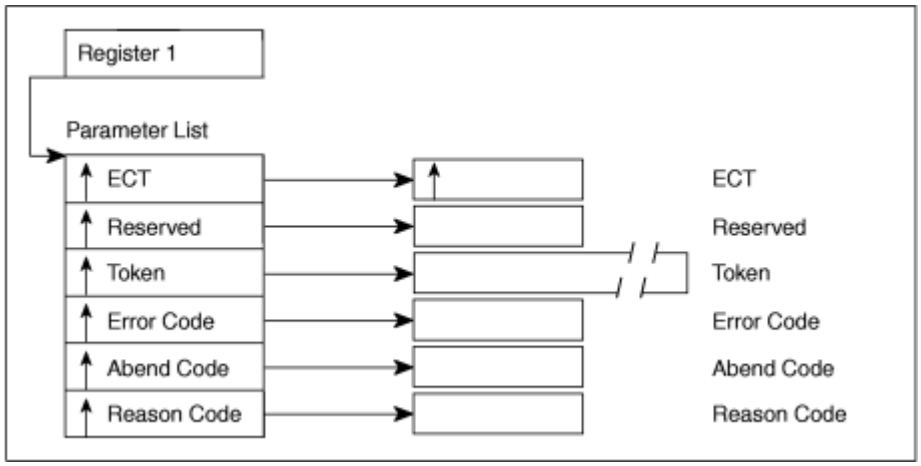


Figure 145. Parameter List for IKJEFTST

The parameters are:

Parameter 1
IKJEFTST (and IKJEFTSI and IKJEFTSR) need to invoke TSO/E I/O services (STACK, PUTLINE, GETLINE, PUTGET). It need to tell these underlying services which environment control table (ECT) to use. You have a choice at this point which ECT these services are to use.

The first parameter is a fullword containing a pointer to the environment control table (ECT) for the current environment. The ECT address can be set to one of the following values:

- The user's current environment control table (ECT)
- A value of X'00000000'

If the ECT address is set to X'00000000', the original ECT created when your TSO/E session was initialized is used. The address of the ECT is placed in this parameter on return to the caller.

Parameter 2
The second parameter is a reserved fullword. Although this parameter is not used, it must contain X'00000000'.

Parameter 3
The third parameter is a token consisting of four fullwords that identifies the TSO/E command/program invocation platform. This token must identify a command/program invocation platform that exists on the current task.

Parameter 4
The fourth parameter is a fullword containing an error code if IKJEFTST completes unsuccessfully.

All following parameters are optional. Note that the high-order bit of the address of the last parameter used must be on to indicate the end of the parameter list.

Parameter 5
The fifth parameter optional. It is a fullword containing the abend code returned from IKJEFTST to the application program when IKJEFTST terminates abnormally.

Parameter 6
The sixth parameter is optional. It is a fullword containing the reason code returned from IKJEFTST to the application program when IKJEFTST terminates abnormally.

Output from IKJEFTST

IKJEFTST passes a return code to the calling program in general register 15. For high-level languages that cannot interrogate register 15, IKJEFTST also places the return code in general register 0.

Return Codes from IKJEFTST

The TSO termination routine return codes are shown in Table 123 on page 381.

Table 123. Return codes from IKJEFTST

Return code dec(Hex)	Meaning
0(0)	TSO/E Service Facility termination was successful: • If an ECT was created during TSO/E Service Facility initialization, the ECT is destroyed. • The TOKEN field contains a zero, indicating the token is no longer valid. • The ERROR field contains zero.
12(C)	TSO/E Service Facility termination was unsuccessful because of inconsistent or incorrect parameters. The ERROR field shows the reason for the error: **Error code = 1 (dec)** A non-zero reserved parameter (2) passed to IKJEFTST. **Error code = 2 (dec)** A null token parameter passed to IKJEFTST. **Note:** The high-order bits of all the parameters in the parameter list pointed to by register 1 must be off, except for the last parameter. If IKJEFTST detects this error, it sets return code = 12, but it cannot set the ERROR field.

Table 123. Return codes from IKJEFTST (continued)

Return code dec(Hex)	Meaning
20(14)	TSO/E Service Facility termination was unsuccessful because of an environmental error. The ERROR field shows the reason for the error: **Error code = 20 (dec)** IKJEFTST invoked in a non-TSO/E environment. **Error code = 21 (dec)** IKJEFTST invoked in an authorized environment. **Error code = 22 (dec)** An incorrect token was passed to IKJEFTST. **Error code = 23 (dec)** IKJEFTST was unable to release all resources related to the passed TSF token.
92(5C)	TSO/E Service Facility termination was unsuccessful. A recovery environment could not be established.
96(60)	TSO/E Service Facility termination was unsuccessful. A parameter is not accessible; see the abend code and reason code parameters for the abend and reason codes.
100(64)	TSO/E Service Facility termination was unsuccessful. Abnormal termination; see the abend code and reason code parameters for the abend and reason codes.

Application Program Interface to IKJEFTSR

IKJEFTSR can be invoked according to the rules of the application programming language in use. For example, certain programming languages can accept only up to six characters in a name. The alias TSOLNK may be used for IKJEFTSR for this purpose.

Call Invocations Using TSOLNK

The TSO/E Service Facility supports the call invocation formats for PL/I, COBOL, FORTRAN, PASCAL, and Assembler. Therefore, you should use the syntax that is appropriate for the language you are using.

The language you choose may include language-specific constructs for invoking external programs. For example, PL/I contains the FETCH statement to get dynamic addressability to external programs. You may choose to use these statements. Other languages may have compiler options that allow for dynamic invocation of subroutines. Use these methods, if possible, to avoid link-editing TSOLNK with your application program. See also “Passing Control to IKJEFTSR” on page 372.

PL/I

For calls in PL/I the format for invoking the TSO/E Service Facility from functions by using TSOLNK is:

```
CALL TSOLNK (PARM1,PARM2,PARM3,PARM4,PARM5,PARM6);
```

In PL/I programs, you should include the following declare statements:

```
DECLARE 1 PARM1,
          2 PARM11 FIXED BINARY (15,0), /* RESERVED                */
          2 PARM13 BIT(8),              /* ABEND FLAG              */
                                        /* 0 -ABEND WITHOUT DUMP   */
                                        /* 1 -ABEND WITH DUMP      */
          2 PARM14 BIT(8);              /* FUNCTION CODE           */
DECLARE PARM2 CHARACTER(8);             /* NAME OF FUNCTION        */
DECLARE PARM3 FIXED BINARY(31,0);       /* LENGTH OF CMD OR PROG   */
DECLARE PARM4 FIXED BINARY(31,0);       /* FUNCTION RETURN CODE    */
DECLARE PARM5 FIXED BINARY(31,0);       /* TSF REASON CODE         */
DECLARE PARM6 FIXED BINARY(31,0);       /* FUNCTION ABEND CODE     */
DECLARE (FILEOUT) FILE;                 /* PL/I OUTPUT FILE        */
DECLARE TSOLNK ENTRY(                   /*                         */
  1,                                    /* STRUCTURE OF 4 BYTES    */
    2 FIXED BINARY(15,0),               /* BYTE 1 RESERVED         */
    2 BIT(8),                           /* BYTE 3 ABEND FLAG       */
    2 BIT(8),                           /* BYTE 4 FUNCTION FLAG    */
  CHARACTER (*),                        /* NAME OF PROGRAM OR CMD  */
  FIXED BINARY(31,0),                   /* LENGTH OF CMD OR PROG   */
  FIXED BINARY(31,0),                   /* FUNCTION RETURN CODE    */
  FIXED BINARY(31,0),                   /* TSF REASON CODE         */
  FIXED BINARY(31,0)                    /* FUNCTION ABEND CODE     */
  )
  EXTERNAL OPTIONS(ASSEMBLER RETCODE INTER);
```

Figure 146. Format of the Parameter List Written in PL/I

COBOL

For calls in COBOL the format for invoking the TSO/E Service Facility from functions by using TSOLNK is:

```
CALL 'TSOLNK' USING PARM1 PARM2 PARM3 PARM4 PARM5 PARM6.
```

In COBOL programs, you should include the following:

```
* DEFINE STORAGE FOR PARMS
*    PARM1 IS DECIMAL VALUE OF FLAGS
*    PARM2 IS COMMAND TEXT
*    PARM3 IS COMMAND LENGTH (SET TO 80)
*    PARM4 IS FUNCTION RETURN CODE VALUE FROM TSOLNK
*    PARM5 IS TSF REASON CODE VALUE FOR ABEND FROM TSOLNK
*    PARM6 IS FUNCTION ABEND CODE VALUE FROM TSOLNK

 01  PARM1 PICTURE S9(9) COMP.
 01  PARM2 PICTURE X(80).
 01  PARM3 PICTURE S9(9) VALUE +80 COMP.
 01  PARM4 PICTURE S9(9) VALUE +0 COMP.
 01  PARM5 PICTURE S9(9) VALUE +0 COMP.
 01  PARM6 PICTURE S9(9) VALUE +0 COMP.
```

Figure 147. Format of the Parameter List Written in COBOL

FORTRAN

For calls in FORTRAN the format for invoking the TSO/E Service Facility from functions by using TSOLNK is:

```
I = TSOLNK(PARM1,PARM2,PARM3,PARM4,PARM5,PARM6)
```

In FORTRAN programs, you should include the following:

```
EXTERNAL TSOLNK
INTEGER TSOLNK
INTEGER PARM1,PARM3,PARM4,PARM5
INTEGER PARM2(20),FILL
```

Figure 148. Format of the Parameter List Written in FORTRAN

PASCAL

For calls in PASCAL the format for invoking the TSO/E Service Facility from functions by using TSOLNK is:

```
TSOLNK(PARM1,PARM2,PARM3,PARM4,PARM5,PARM6)
```

In PASCAL programs you should include the following:

```
PROCEDURE TSOLNK( VAR PARM1:PA4;
                  VAR PARM2:PA80;
                  VAR PARM3:INTEGER;
                  VAR PARM4:INTEGER;
                  VAR PARM5:INTEGER;
                  VAR PARM6:INTEGER);
FORTRAN; (* THIS KEYWORD IS REQUIRED TO ESTABLISH LINKAGE TO TSF *)
VAR
 PARM1:PA4;          (* WORD OF CONTROL BITS *)
 PARM2:PA80;         (* PROGRAM BUFFER       *)
 PARM3:INTEGER;      (* LENGTH OF PROGRAM    *)
 PARM4:INTEGER;      (* FUNCTION RETURN CODE               *)
 PARM5:INTEGER;      (* TSO SERVICE FACILITY REASON CODE   *)
 PARM6:INTEGER;      (* FUNCTION ABEND CODE                *)
 FILEOUT:TEXT;       (* DECLARE OUTPUT FILE NAME           *)
```

Figure 149. Format of the Parameter List Written in PASCAL

Examples of Invoking the TSO/E Service Facility

The following sample programs and CLIST demonstrate the use of the TSO/E Service Facility to invoke commands, programs, CLISTs, and REXX execs in:

- Assembler
- FORTRAN
- COBOL
- PL/I
- PASCAL

In these examples, the term ‘function’ means the program, command, CLIST, or REXX exec IKJEFTSR invokes.

Assembler Program Using IKJEFTSI

```
*****************************************************************
*                                                               *
* SET UP THE PARAMETER LIST FOR IKJEFTSI.  A VALUE OF ZERO IS   *
* PASSED FOR ALL PARAMETERS.                                    *
*                                                               *
*****************************************************************
     XC    IKJEFTSJ(72),IKJEFTSJ     INITIALIZE PARAMETER VALUES

     LA    R2,EFTSI_ECTPARM          PLACE ADDRESS OF ECTPARM
     ST    R2,EFTSI_ECTPARM@          IN PARAMETER LIST
     LA    R2,EFTSI_RESERVED         PLACE ADDRESS OF RESERVED
     ST    R2,EFTSI_RESERVED@         DATA IN PARAMETER LIST
     LA    R2,EFTSI_TOKEN            PLACE ADDRESS OF TOKEN
     ST    R2,EFTSI_TOKEN@            DATA IN PARAMETER LIST
     LA    R2,EFTSI_ERROR            PLACE ADDRESS OF ERROR
     ST    R2,EFTSI_ERROR@            DATA IN PARAMETER LIST
     LA    R2,EFTSI_ABEND            PLACE ADDRESS OF ABEND
     ST    R2,EFTSI_ABEND@            DATA IN PARAMETER LIST
     LA    R2,EFTSI_REASON           PLACE ADDRESS OF REASON
     ST    R2,EFTSI_REASON@           DATA IN PARAMETER LIST
     OI    EFTSI_REASON@,X'80'       SET HIGH ORDER BIT


     LA    R1,IKJEFTSJ               REG 1 POINTS TO PARM LIST
     CALLTSSR EP=IKJTSFI             INVOKE IKJEFTSI, SPECIFYING
*                                     ENTRY POINT IKJTSFI.

     ST    R15,IKJEFTSI_RC           SAVE RETURN CODE
*
```

```
*****************************************************************
*                                                               *
*  CHECK THE RETURN CODE FROM IKJEFTSI.                         *
*                                                               *
*****************************************************************
     SR    R3,R3                       DETERMINE IF THE RETURN
     CR    R15,R3                      CODE IS ZERO
     BL    NO_ERROR                   BRANCH ON ZERO RC
     B     ERROR                       BRANCH ON NON-ZERO RC
```

Figure 150. Assembler Language Program Demonstrating the Use of IKJEFTSI

Assembler Program Using IKJEFTSR to Invoke a Command

```
TSF      CSECT
         STM   R14,R12,12(R13)
         BALR  R12,0
         USING *,R12
         ST    R13,SAVEAREA+4
         LA    R11,SAVEAREA
         ST    R11,8(,R13)
         LA    R13,SAVEAREA
*
```

```
MAIN     DS    0H
         .
         .
         .
         L     R15,CVTPTR              ESTABLISH
         L     R15,CVTTVT(,R15)        ADDRESSABILITY TO THE
         L     R15,TSVTASF-TSVT(,R15)  TSO SERVICE FACILITY
*
*        INVOKE THE TSO SERVICE FACILITY -- EXECUTE LISTBC COMMAND
*
         CALL  (15),(FLAGS,CMDBUF,BUFLEN,RETCODE,RSNCODE,ABNDCODE),VL
         LTR   R15,R15            CHECK TSR RETURN CODE
         BNZ   ERRORRTN           BAD RETURN CODE FROM TSR
         CLC   RETCODE,ZERO       CHECK COMMAND PROCESSOR ERROR
         BH    ERRORCMD           BAD RETURN CODE FROM COMMAND
         B     ENDUP              NO ERROR --- EXIT
ERRORRTN DS   0H
*
```

```
*
*        ANALYZE TSO SERVICE FACILITY ERROR
         .
         .
         .
*
*
         B    ENDUP
ERRORCMD DS   0H
*
```

```
*        ANALYZE COMMAND PROCESSOR ERROR
         .
         .
         .
*
ENDUP    DS   0H
         L    R13,4(,R13)
         LM   R14,R12,12(R13)
         SLR  R15,R15
         BR   R14
*
```

Figure 151. Assembler Language Program Demonstrating the Use of IKJEFTSR to Invoke a Command

Figure 'Assembler Language Program Demonstrating the Use of IKJEFTSR to Invoke a Command' (**Continued**)

```
*        DATA AREAS
*
ZERO     DC    F'0'        ZERO CONSTANT
FLAGS    DS    0F          MAPS FIRST PARM TO IKJEFTSR
RESFLAGS DC    H'0'        FLAG WORD
ABFLAGS  DC    X'01'       DUMP IF ABEND OCCURS
FNCFLAGS DC    X'01'       TELL TSR TO EXECUTE THE COMMAND
         *
```

```
CMDBUF   DC    C'LISTBC'   NAME OF COMMAND TO BE EXECUTED
         *
BUFLEN   DC    F'6'        LENGTH OF COMMAND BUFFER
RETCODE  DS    F           RETURN CODE FROM COMMAND
RSNCODE  DS    F           REASON CODE
ABNDCODE DS    F           ABEND CODE
SAVEAREA DS    18F         SAVE AREA
         .
         .
         .
```

```
CVTPTR   EQU   16
CVTTVT   EQU   X'9C'
R15      EQU   15
R14      EQU   14
R13      EQU   13
R12      EQU   12
R11      EQU   11
R9       EQU   9
R8       EQU   8
         IKJTSVT
         END
```

Assembler Program Using IKJEFTST

```
*****************************************************************
*                                                               *
* SET UP THE PARAMETER LIST FOR IKJEFTST.  A VALUE OF ZERO IS   *
* PASSED FOR ALL PARAMETERS, EXCEPT FOR THE TOKEN THAT IS       *
* GOTTEN FROM IKJEFTSI.                                         *
*                                                               *
*****************************************************************
     XC    IKJEFTSV(72),IKJEFTSV       INITIALIZE PARAMETER VALUES

     LA    R2,EFTST_ECTPARM            PLACE ADDRESS OF ECTADDR
     ST    R2,EFTST_ECTPARM@            DATA IN PARAMETER LIST
     LA    R2,EFTST_RESERVED           PLACE ADDRESS OF RESERVED
     ST    R2,EFTST_RESERVED@           DATA IN PARAMETER LIST
     LA    R2,EFTST_TOKEN              PLACE ADDRESS OF TOKEN
     ST    R2,EFTST_TOKEN@              DATA IN PARAMETER LIST
     MVC   EFTST_TOKEN(16),EFTSI_TOKEN PASS TOKEN FROM IKJEFTSI
     LA    R2,EFTST_ERROR              PLACE ADDRESS OF ERROR
     ST    R2,EFTST_ERROR@              DATA IN PARAMETER LIST
     LA    R2,EFTST_ABEND              PLACE ADDRESS OF ABEND
     ST    R2,EFTST_ABEND@              DATA IN PARAMETER LIST
     LA    R2,EFTST_REASON             PLACE ADDRESS OF REASON
     ST    R2,EFTST_REASON@             DATA IN PARAMETER LIST
     OI    EFTST_REASON@,X'80'         SET HIGH ORDER BIT

     LA    R1,IKJEFTSV                 REG 1 POINTS TO PARM LIST
     CALLTSSR EP=IKJTSFT               INVOKE IKJEFTST, SPECIFYING
*                                       ENTRY POINT IKJTSFT.

     ST    R15,IKJEFTST_RC             SAVE RETURN CODE
```

```
*****************************************************************
*                                                               *
*  CHECK THE RETURN CODE FROM IKJEFTST.                         *
*                                                               *
*****************************************************************
     SR    R3,R3                        DETERMINE IF THE RETURN
     CR    R15,R3                       CODE IS ZERO
     BL    NO_ERROR                    BRANCH ON ZERO RC
     B     ERROR                        BRANCH ON NON-ZERO RC
```

Figure 152. Assembler Language Program Demonstrating the Use of IKJEFTST

Assembler Program Using IKJEFTSI, IKJEFTSR, IKJEFTST to Invoke a Command

```
COMPOS4  CSECT  ,
COMPOS4  AMODE  31
COMPOS4  RMODE  ANY
@MAINENT DS     0H
         STM    R14,R12,12(R13)       ENTRY LINKAGE
         LR     R12,R15
@PSTART  EQU    COMPOS4
         USING  @PSTART,R12
         ST     R13,SAVEAREA+4
         LA     R11,SAVEAREA
         ST     R11,8(,R13)
         LA     R13,SAVEAREA
*
```

```
*
MAIN     DS     0H
*
*****************************************************************
*                                                               *
* SET UP THE PARAMETER LIST FOR IKJEFTSI.  A VALUE OF ZERO IS   *
* PASSED FOR ALL PARAMETERS.                                    *
*                                                               *
*****************************************************************
         XC     IKJEFTSJ(72),IKJEFTSJ   INITIALIZE PARAMETER VALUES
         LA     R2,EFTSI_ECTPARM       PLACE ADDRESS OF ECTPARM
         ST     R2,EFTSI_ECTPARM@       IN THE PARAMETER LIST
         LA     R2,EFTSI_RESERVED      PLACE ADDRESS OF RESERVED
         ST     R2,EFTSI_RESERVED@      DATA IN PARAMETER LIST
         LA     R2,EFTSI_TOKEN         PLACE ADDRESS OF TOKEN
         ST     R2,EFTSI_TOKEN@         DATA IN PARAMETER LIST
         LA     R2,EFTSI_ERROR         PLACE ADDRESS OF ERROR
         ST     R2,EFTSI_ERROR@         DATA IN PARAMETER LIST
         LA     R2,EFTSI_ABEND         PLACE ADDRESS OF ABEND
         ST     R2,EFTSI_ABEND@         DATA IN PARAMETER LIST
         LA     R2,EFTSI_REASON        PLACE ADDRESS OF REASON
         ST     R2,EFTSI_REASON@        DATA IN PARAMETER LIST
         OI     EFTSI_REASON@,X'80'    SET HIGH ORDER BIT
         LA     R1,IKJEFTSJ            REG 1 POINTS TO PARM LIST
         CALLTSSR EP=IKJTSFI           INVOKE IKJEFTSI, SPECIFYING
*                                        ENTRY POINT IKJTSFI.
*        ----   Provide for error conditions here   ----
*
```

Figure 153. Assembler Language Program Demonstrating the Use of IKJEFTSI, IKJEFTSR, and IKJEFTST to Invoke a Command

Figure of 'Assembler Language Program Demonstrating the Use of IKJEFTSI, IKJEFTSR, and IKJEFTST to Invoke a Command' (**Continued**)

```
*****************************************************************
*                                                               *
* SET UP THE PARAMETER LIST FOR IKJEFTSR.                       *
*                                                               *
*****************************************************************
         LA     R2,TSR1                PLACE ADDR OF TSR1
         ST     R2,TSR1@                IN THE PARAMETER LIST
         LA     R2,TSR2                PLACE ADDR OF TSR2
         ST     R2,TSR2@                IN THE PARAMETER LIST
         LA     R2,TSR3                PLACE ADDR OF TSR3
         ST     R2,TSR3@                IN THE PARAMETER LIST
         LA     R2,TSR4                PLACE ADDR OF TSR4
         ST     R2,TSR4@                IN THE PARAMETER LIST
         LA     R2,TSR5                PLACE ADDR OF TSR5
         ST     R2,TSR5@                IN THE PARAMETER LIST
         LA     R2,TSR6                PLACE ADDR OF TSR6
         ST     R2,TSR6@                IN THE PARAMETER LIST
         LA     R2,TSR7                PLACE ADDR OF TSR7
         ST     R2,TSR7@                IN THE PARAMETER LIST
         LA     R2,TSR8                PLACE ADDR OF TSR8
         ST     R2,TSR8@                IN THE PARAMETER LIST
         LA     R2,EFTSI_TOKEN         PLACE ADDR OF THE TOKEN
         ST     R2,TSR9@                IN THE PARAMETER LIST
         OI     TSR9@,X'80'            SET HIGH ORDER BIT
         LA     R1,TSR1@               REG 1 POINTS TO PARM LIST
         L      R15,CVTPTR
         L      R15,CVTTVT(,R15)
         L      R15,TSVTASF-TSVT(,R15)
         BALR   R14,R15                CALL IKJEFTSR
*        ----   Provide for error conditions here   ----
```

```
*
         L      R15,CVTPTR
         L      R15,CVTTVT(,R15)
         L      R15,TSVTASF-TSVT(,R15)
         BALR   R14,R15                CALL IKJEFTSR
         CALL   (15),(TSR1,TSR2,TSR3,TSR4,TSR5,TSR6,TSR7,TSR8,
               EFTSI_TOKEN),VL
*        ----   Provide for error conditions here   ----
*
```

Figure of 'Assembler Language Program Demonstrating the Use of IKJEFTSI, IKJEFTSR, and IKJEFTST to Invoke a Command' (**Continued**)

```
*****************************************************************
*                                                               *
* SET UP THE PARAMETER LIST FOR IKJEFTST.  A VALUE OF ZERO IS    *
* PASSED FOR ALL PARAMETERS, EXCEPT FOR THE TOKEN THAT IS        *
* GOTTEN FROM IKJEFTSI.                                         *
*                                                               *
*****************************************************************
         XC     IKJEFTSV(72),IKJEFTSV  INITIALIZE PARAMETER VALUES
         LA     R2,EFTST_ECTPARM       PLACE ADDRESS OF ECTPARM
         ST     R2,EFTST_ECTPARM@       IN THE PARAMETER LIST
         LA    R2,EFTST_RESERVED       PLACE ADDRESS OF RESERVED
         ST    R2,EFTST_RESERVED@       DATA IN PARAMETER LIST
         LA    R2,EFTST_TOKEN          PLACE ADDRESS OF TOKEN
         ST    R2,EFTST_TOKEN@          DATA IN PARAMETER LIST
         MVC   EFTST_TOKEN(16),EFTSI_TOKEN      PASS TOKEN
*                                               FROM IKJEFTSI
         LA    R2,EFTST_ERROR          PLACE ADDRESS OF ERROR
         ST    R2,EFTST_ERROR@          DATA IN PARAMETER LIST
         LA    R2,EFTST_ABEND          PLACE ADDRESS OF ABEND
         ST    R2,EFTST_ABEND@          DATA IN PARAMETER LIST
         LA    R2,EFTST_REASON         PLACE ADDRESS OF REASON
         ST    R2,EFTST_REASON@         DATA IN PARAMETER LIST
         OI     EFTST_REASON@,X'80'    SET HIGH ORDER BIT
         LA     R1,IKJEFTSV            REG 1 POINTS TO PARM LIST
         CALLTSSR EP=IKJTSFT           INVOKE IKJEFTST, SPECIFYING
*                                        ENTRY POINT IKJTSFT.
*        ----   Provide for error conditions here   ----
*
```

```
         DS     0H
         L      R13,4(,R13)           EXIT  LINKAGE
         LM     R14,R12,12(R13)
         SLR    R15,R15
         BR     R14
*
```

```
*
SAVEAREA DS   18F
*
```

```
*
ZERO     DC     F'0'
*
```

```
*                               IKJEFTSI Input:
         IKJADFMT
         IKJEFTSJ
*                                                    000
*                               IKJEFTSR Input:
TSR1@    DS     F               o  Address of Parm 1
TSR2@    DS     F               o  Address of Parm 2
TSR3@    DS     F               o  Address of Parm 3
TSR4@    DS     F               o  Address of Parm 4
TSR5@    DS     F               o  Address of Parm 5
TSR6@    DS     F               o  Address of Parm 6
TSR7@    DS     F               o  Address of Parm 7
TSR8@    DS     F               o  Address of Parm 8
TSR9@    DS     F               o  Address of Parm 9
TSR1     DS     0F              o  Parm 1:
RESFLAGS DC     X'00'              Byte 1 - Reserved
UNAUTHFL DC     X'01'              Byte 2 - Unauthorized environment
ABFLAGS  DC     X'01'              Byte 3 - Dump indicator
FNCFLAGS DC     X'01'              Byte 4 - Cmd, REXX, CLIST indicator
*
```

Figure of 'Assembler Language Program Demonstrating the Use of IKJEFTSI, IKJEFTSR, and IKJEFTST to Invoke a Command' (**Continued**)

```
TSR2     DC     C'ALTLIB DISPLAY      '
*
TSR3     DC     F'20'           o  Parm 3 - Buffer Len
TSR4     DS     F               o  Parm 4 - Return code
TSR5     DS     F               o  Parm 5 - Reason code
TSR6     DS     F               o  Parm 6 - Abend code
TSR7     DC     F'0'            o  Parm 7 - Parm for pgms
TSR8     DC    4F'0'            o  Parm 8 - CPPL
*                                  let IKJEFTSR get appropriate values
*                                o  Parm 9 - Token
*                                  obtained from IKJEFTSI Parm 3
*
```

```
*                                IKJEFTST Input:
         IKJEFTSV
*
CVTPTR   EQU  16
CVTTVT   EQU  X'9C'
R15      EQU  15
R14      EQU  14
R13      EQU  13
R12      EQU  12
R11      EQU  11
R10      EQU  10
R9       EQU  9
R8       EQU  8
R7       EQU  7
R6       EQU  6
R5       EQU  5
R4       EQU  4
R3       EQU  3
R2       EQU  2
R1       EQU  1
         IKJTSVT
         END
```

FORTRAN Program Using TSOLNK to Invoke a Command (FORTRAN G1)

```
C     THIS FORTRAN PROGRAM WILL INTERFACE WITH THE TSO SERVICE FACILITY.
C     THE PROGRAM ISSUES THE LISTD COMMAND IN TSO/E AND THEN PRINTS OUT
C     THE COMMAND BUFFER  AND THE RETURN, REASON AND ABEND CODES
C     RESULTING FROM THE EXECUTION OF THE TSO SERVICE FACILITY.
C     BECAUSE THIS PROGRAM DOES ITS OWN I/O AFTER IT CALLS THE
C     TSO SERVICE FACILITY, THE USER MIGHT WANT TO ALLOCATE THE FILE
C     NAME FT06F001 TO THE TERMINAL WITH THE TSO/E COMMAND:
C          ALLOC F(FT06F001) DSN(*)
C     THIS PROGRAM WAS COMPILED ON THE FORTRAN G1 COMPILER
C
```

```
      EXTERNAL TSOLNK
      INTEGER TSOLNK
      INTEGER PARM11,PARM12,PARM13,PARM14
      INTEGER PARM1,PARM3,PARM4,PARM5
      INTEGER PARM2(20),FILL
```

```
C     PLACE COMMAND NAME IN PARM2
      DATA PARM2  /'LIST','D 'S','YS1.','LINK','LIB','    ','    ',
     +'    ','    ','    ','    ','    ','    ','    ','    ','    ',
     +'    ','    ','    ','    '/
      DATA FILL /'    '/
      PARM11 = 0
      PARM12 = 0
      PARM13 = 0
```

```
C     SPECIFY THAT A COMMAND IS TO BE EXECUTED
      PARM14 = 1
C     FILL IN THE CONTROL BITS OF THE FIRST PARAMETER
C       TO REQUEST DUMP OFF
      PARM1 = (PARM11*16**6)+(PARM12*16**4)+(PARM13*16**2)+PARM14
C     PUT THE COMMAND LENGTH INTO THE THIRD PARAMETER
      PARM3 = 80
```

```
C     ZERO OUT THE RETURNED VALUES BEFORE THE CALL
      PARM4 = 0
      PARM5 = 0
      PARM6 = 0
C     EXECUTE THE TSO SERVICE FACILITY
      I = TSOLNK(PARM1,PARM2,PARM3,PARM4,PARM5,PARM6)
      WRITE (6,104)
```

```
C     PRINT OUT THE COMMAND EXECUTED
104   FORMAT (' ','COMMAND EXECUTED:  ')
C     PRINT OUT THE RETURNED VALUES
      WRITE (6,103) I
103   FORMAT (' ','THE TSOLNK RETURN CODE IS ',I6)
      WRITE (6,105) FILL,(PARM2(I),I=1,20)
105   FORMAT (21A4)
      WRITE (6,100) PARM4
100   FORMAT (' ','THE FUNCTION RETURN CODE IS ',I6)
      WRITE (6,101) PARM5
101   FORMAT (' ','     THE TSF REASON CODE IS ',I6)
      WRITE (6,102) PARM6
102   FORMAT (' ','THE ABEND  CODE IS ',I6)
      STOP
      END
```

Figure 154. FORTRAN Program Demonstrating the Use of TSOLNK to Invoke a Command (FORTRAN G1)

FORTRAN Program Using TSOLNK to Invoke a Command (VS FORTRAN)

```
C     THIS FORTRAN PROGRAM WILL INTERFACE WITH THE TSO SERVICE FACILITY.
C     ISSUE COMMAND   LISTD 'SYS1.LINKLIB'   AND THEN PRINT THE
C     COMMAND BUFFER, RETURN CODE, REASON CODE AND ABEND CODE
C     FORTRAN FILE FT06FT001 IS USED FOR OUTPUT
C     THIS PROGRAM WAS COMPILED ON THE VS FORTRAN COMPILER
      EXTERNAL TSOLNK                                                 00001000
      INTEGER  TSOLNK                                                 00001000
      INTEGER  PARM11,PARM12,PARM13,PARM14                            00001000
      INTEGER  PARM1,PARM3,PARM4,PARM5                                00001000
      CHARACTER*80 PARM2
      CHARACTER*4  FILL

C     PLACE COMMAND IN PARM2
      DATA PARM2  /'LISTD 'SYS1.LINKLIB'/
      DATA FILL /'    '/
C     COMPUTE PARM1
C         SPECIFY RESERVED BITS (ALL ZERO)
      PARM11 = 0
C         SPECIFY AUTHORIZATION = YES TO BE USED
      PARM12 = 0
C         SPECIFY THAT A DUMP IS NOT TO BE TAKEN
      PARM13 = 0
C         SPECIFY THAT A COMMAND IS BEING INVOKED
      PARM14 = 1
C         SPECIFY THAT A PROGRAM IS BEING INVOKED
      PARM14 = 2
C         SPECIFY THAT A REXX EXEC IS BEING INVOKED
      PARM14 = 5
C         FILL IN THE CONTROL BITS OF PARM1
      PARM1 = (PARM11*16**6)+(PARM12*16**4)+(PARM13*16**2)+PARM14

C     PUT THE COMMAND LENGTH INTO THE THIRD PARAMETER
      PARM3 = 80
C     ZERO OUT THE RETURNED VALUES BEFORE THE CALL
      PARM4 = 0
      PARM5 = 0
      PARM6 = 0

C     EXECUTE THE TSO SERVICE FACILITY
      I = TSOLNK(PARM1,PARM2,PARM3,PARM4,PARM5,PARM6)
C     PRINT OUT THE COMMAND EXECUTED
      WRITE (6,100)
100   FORMAT (' ','COMMAND EXECUTED: ')
      WRITE (6,101) FILL,PARM2
101   FORMAT (A4,A80)
      WRITE (6,102) PARM3
102   FORMAT (' ','LENGTH OF COMMAND BUFFER IS ',I6)

C     PRINT OUT THE RETURNED VALUES
      WRITE (6,103) I
103   FORMAT (' ','  THE TSOLNK RETURN CODE IS ',I6)
      WRITE (6,104) PARM4
104   FORMAT (' ','THE FUNCTION RETURN CODE IS ',I6)
      WRITE (6,105) PARM5
105   FORMAT (' ','     THE TSF REASON CODE IS ',I6)
      WRITE (6,106) PARM6
106   FORMAT (' ','          THE ABEND CODE IS ',I6)
      STOP
      END
```

Figure 155. FORTRAN Program Demonstrating the Use of TSOLNK to Invoke a Command (VS FORTRAN)

COBOL Program Using TSOLNK to Invoke a Command

```
 ID DIVISION.
 PROGRAM-ID.   TSOSVR.
* THIS COBOL PROGRAM WILL INTERFACE WITH THE TSO SERVICE FACILITY.
* THIS PROGRAM WILL ISSUE THE LISTBC COMMAND.
* BECAUSE THIS PROGRAM DOES
* ITS OWN I/O AFTER THE TSO/E COMMAND IS EXECUTED TO DISPLAY RETURN
* CODES, USER SHOULD ALLOCATE FILE "SYSPRT" TO THE TERMINAL.

* THIS PROGRAM WILL RUN ON THE OS/VS COBOL COMPILER RELEASE 3 OR
* HIGHER

 ENVIRONMENT DIVISION.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.

*    DEFINE OUTPUT DEVICE

     SELECT TRMPRT ASSIGN TO UT-S-SYSPRT.
 DATA DIVISION.
 FILE SECTION.

     DEFINE OUTPUT FILE

 FD  TRMPRT
     LABEL RECORDS ARE OMITTED
     RECORD CONTAINS 133 CHARACTERS.

*    DEFINE OUTPUT RECORD

 01  OUTREC.
     02 OUT-LINE     PICTURE X(133).

 WORKING-STORAGE SECTION.

*    DEFINE OUTPUT RECORD FORM

 01  OUT-RECORD.
     02 CONTROL-CHAR PICTURE X VALUE SPACE.
     02 COMMENT      PICTURE X(25).
     02 OUT-VALUE    PICTURE +++++++++9.
     02 FILLER       PICTURE X(111) VALUE SPACES.

*    DEFINE COMMENT VALUES FOR OUTPUT RECORD FORM

 01  RETURN-COMMENT PICTURE X(25) VALUE 'FUNCTION RETURN CODE IS  '.
 01  REASON-COMMENT PICTURE X(25) VALUE '     TSF REASON CODE IS '.
 01  ABEND-COMMENT  PICTURE X(25) VALUE 'FUNCTION ABEND  CODE IS  '.

*    DEFINE FLAGS TO BE FULL WORDS WITH APPROPRIATE BITS ON

 01  FLAG1-ON  PICTURE S9(9) VALUE +16777216 COMP.
 01  FLAG2-ON  PICTURE S9(9) VALUE +65536 COMP.
 01  FLAG3-ON  PICTURE S9(9) VALUE +256 COMP.
 01  FLAG4-ON  PICTURE S9(9) VALUE +2 COMP.
 01  FLAG1-OFF PICTURE S9(9) VALUE +0 COMP.
 01  FLAG2-OFF PICTURE S9(9) VALUE +0 COMP.
 01  FLAG3-OFF PICTURE S9(9) VALUE +0 COMP.
 01  FLAG4-OFF PICTURE S9(9) VALUE +1 COMP.
```

Figure 156. COBOL Program Demonstrating the Use of TSOLNK to Invoke a Command

Figure of 'COBOL Program Demonstrating the Use of TSOLNK to Invoke a Command' (**Continued**)

```
* DEFINE STORAGE FOR PARMS
*    PARM1 IS DECIMAL VALUE OF FLAGS
*    PARM2 IS COMMAND TEXT
*    PARM3 IS COMMAND LENGTH (SET TO 80)
*    PARM4 IS FUNCTION RETURN CODE VALUE FROM TSOLNK
*    PARM5 IS TSF REASON CODE VALUE FOR ABEND FROM TSOLNK
*    PARM6 IS FUNCTION ABEND CODE VALUE FROM TSOLNK

 01  PARM1 PICTURE S9(9) COMP.
 01  PARM2 PICTURE X(80).
 01  PARM3 PICTURE S9(9) VALUE +80 COMP.
 01  PARM4 PICTURE S9(9) VALUE +0 COMP.
 01  PARM5 PICTURE S9(9) VALUE +0 COMP.
 01  PARM6 PICTURE S9(9) VALUE +0 COMP.

 PROCEDURE DIVISION.

*    MOVE DESIRED COMMAND TO PARM2

 READY-COMMAND.
     MOVE SPACES TO PARM2.
     MOVE 'LISTBC' TO PARM2.

*    SET FLAGS BY ADDING APPROPRIATELY VALUED FLAG VARIABLES

 READY-FLAGS.
     MOVE 0 TO PARM1.

*    RESERVED FLAG

     ADD FLAG1-OFF TO PARM1.

*    RESERVED FLAG

     ADD FLAG2-OFF TO PARM1.

*    FLAG3-ON TO REQUEST ABEND WITH DUMP

     ADD FLAG3-ON  TO PARM1.

*    FLAG4-OFF TO REQUEST A TSO/E COMMAND (NOT A PROGRAM) BE INVOKED

     ADD FLAG4-OFF TO PARM1.

*    CALL TSOLNK

 CALL-TSOLNK.
     CALL 'TSOLNK' USING PARM1 PARM2 PARM3 PARM4 PARM5 PARM6.

*    PRINT RESULTS

 PRINT-COMMENTS.
     OPEN OUTPUT TRMPRT.

*    PRINT THE FUNCTION RETURN CODE

     MOVE RETURN-COMMENT TO COMMENT.
     MOVE PARM4 TO OUT-VALUE.
     WRITE OUTREC FROM OUT-RECORD.

*    PRINT THE TSF REASON CODE

     MOVE REASON-COMMENT TO COMMENT.
     MOVE PARM5 TO OUT-VALUE.
     WRITE OUTREC FROM OUT-RECORD.

*    PRINT THE FUNCTION ABEND CODE

     MOVE ABEND-COMMENT TO COMMENT.
     MOVE PARM6 TO OUT-VALUE.
     WRITE OUTREC FROM OUT-RECORD.

     CLOSE TRMPRT.
     STOP RUN.
```

Assembler Program Using IKJEFTSR to Invoke a Program

```
*****************************************************************
*  THIS ASSEMBLER PROGRAM CALLS THE TSO SERVICE FACILITY TO     *
*  EXECUTE THE LINKAGE EDITOR PROGRAM (SYS1.LINKLIB(IEWL)).     *
*  THIS PROGRAM PASSES ONE PARAMETER TO THE LINKAGE EDITOR.     *
*  TO SUCCESSFULLY EXECUTE THE PROGRAM, THE USER SHOULD         *
*  ALLOCATE THE FOLLOWING FILES: SYSUT1, SYSLMOD, SYSLIN AND    *
*  SYSPRINT.                                                    *
*****************************************************************
TSFPROG  CSECT
         STM   14,12,12(13)        ENTRY LINKAGE
         BALR  12,0
         USING *,12                USE R12 AS BASE REG
         ST    13,SAVE+4           SAVE CALLERS SAVE AREA
         LA    13,SAVE             HAVE POINTER TO THIS SAVE AREA
         L     15,=V(IKJEFTSR)     GET ADDRESS OF IKJEFTSR
   CALL  (15),(FLAGS,PGM,PGMLEN,RETCODE,REASONC,ABENDCD,PARMLIST),VL
         LTR   15,15               WAS RETURN CODE FROM IKJEFTSR = 0?
         BZ    NOERROR             NO ERROR ---- PROCEED ON
*
*
* CHECK RETCODE, REASONC, AND ABENDCD AT THIS POINT
*
*
NOERROR  EQU  *                   CONTINUE ON WITH PROGRAM
*
*
*
         L     13,SAVE+4           GET CALLERS SAVE AREA
         LM    14,12,12(13)        EXIT LINKAGE
         BR    14                  RETURN TO SUPERVISOR
* DECLARES
SAVE     DS    18F
FLAGS    DS    0F
         DC    XL2'0000'           INITIALIZE TO ZERO
         DC    XL1'01'             SPECIFY DUMP TO BE TAKEN
         DC    XL1'02'             PROGRAM TO BE EXECUTED
PGM      DC    C'IEWL'             NAME OF PROGRAM / LINKAGE EDITOR
PGMLEN   DC    F'4'                LENGTH OF PROGRAM NAME
RETCODE  DS    F                   FUNCTION RETURN CODE
REASONC  DS    F                   TSF REASON CODE
ABENDCD  DS    F                   ABEND CODE
PGMPARM1 DS    0F                  FIRST AND ONLY PARAMETER TO IEWL
         DC    H'8'                LENGTH OF PARM TO IEWL
         DC    C'MAP,XREF'         THE ACTUAL PARM TO IEWL
PARMLIST CALL  ,(PGMPARM1),VL,MF=L SET UP PARM LIST TO IEWL
         END
```

Figure 157. Assembler Program Demonstrating the Use of IKJEFTSR to Invoke a Program

PL/I Program Using TSOLNK to Invoke a Program

```
/*****************************************************************/
/*  THIS PL/I PROGRAM ISSUES THE IEBCOPY PROGRAM IN TSO/E AND THEN */
/*  PRINTS OUT THE COMMAND BUFFER  AND THE RETURN, REASON AND     */
/*  ABEND CODES RESULTING FROM THE EXECUTION OF THE TSO SERVICE   */
/*  FACILITY.  TO USE THE EXAMPLE THE USER MUST ALLOCATE THE      */
/*  FOLLOWING FILES:                                             */
/*       ALLOC F(FILEOUT) DSN(*)                                 */
/*       ALLOC F(SYSIN) DSN(YOUR.SYSIN)                          */
/*       ALLOC F(SYSPRINT) DSN(*)                                */
/*       ALLOC F(INDD) DSN(YOUR.INPDS)                           */
/*       ALLOC F(OUTDD) DSN(YOUR.OUTPDS)                         */
/*  THE EXAMPLE REQUIRES THE FOLLOWING CARD IN YOUR.SYSIN FILE:   */
/*       EXAMPLE    COPY  OUTDD=OUTDD,INDD=INDD                  */
/*                                                               */
/*  THIS PROGRAM WILL RUN ON THE OS/VS PL/I COMPILER RELEASE 2 OR */
/*  HIGHER.                                                      */
/*****************************************************************/
TSOCALL:
  PROCEDURE OPTIONS(MAIN);
DECLARE 1 PARM1,
          2 PARM11 FIXED BINARY (15,0), /* RESERVED                */
          2 PARM13 BIT(8),              /* ABEND FLAG              */
                                        /* 0 -ABEND WITHOUT DUMP   */
                                        /* 1 -ABEND WITH DUMP      */
          2 PARM14 BIT(8);              /* FUNCTION CODE           */
DECLARE PARM2 CHARACTER(8);             /* NAME OF FUNCTION        */
DECLARE PARM3 FIXED BINARY(31,0);       /* LENGTH OF CMD OR PROG   */
DECLARE PARM4 FIXED BINARY(31,0);       /* FUNCTION RETURN CODE    */
DECLARE PARM5 FIXED BINARY(31,0);       /* TSF REASON CODE         */
DECLARE PARM6 FIXED BINARY(31,0);       /* FUNCTION ABEND CODE     */
DECLARE (FILEOUT) FILE;                 /* PL/I OUTPUT FILE        */
DECLARE TSOLNK ENTRY(                   /*                         */
  1,                                    /* STRUCTURE OF 4 BYTES    */
    2 FIXED BINARY(15,0),               /* BYTE 1 RESERVED         */
    2 BIT(8),                           /* BYTE 3 ABEND FLAG       */
    2 BIT(8),                           /* BYTE 4 FUNCTION FLAG    */
  CHARACTER (*),                        /* NAME OF PROGRAM OR CMD  */
  FIXED BINARY(31,0),                   /* LENGTH OF CMD OR PROG   */
  FIXED BINARY(31,0),                   /* FUNCTION RETURN CODE    */
  FIXED BINARY(31,0),                   /* TSF REASON CODE         */
  FIXED BINARY(31,0)                    /* FUNCTION ABEND CODE     */
  )
  EXTERNAL OPTIONS(ASSEMBLER RETCODE INTER);
DECLARE PLIRETV BUILTIN;
```

Figure 158. PL/I Program Demonstrating the Use of TSOLNK to Invoke a Program

Figure 'PL/I Program Demonstrating the Use of TSOLNK to Invoke a Program' (**Continued**)

```
/*****************************************************************/
/*  START OF EXECUTABLE CODE                                     */
/*****************************************************************/
PARM13='00000001'B;                  /* DUMP YES OR NO           */
PARM14='00000010'B;                  /* FUNCTION IS A PROGRAM    */
PARM2 = 'IEBCOPY';                   /* FUNCTION NAME            */
PARM3 = 7;                           /* LENGTH OF PROGRAM        */
CALL TSOLNK(PARM1,PARM2,PARM3,PARM4,PARM5,PARM6);
                                    /* CALL TSO SERVICE FACILITY */
     PUT FILE (FILEOUT) EDIT (' THE TSOLNK RETURN CODE IS ',PLIRETV)
                            (A,F(3)); /* PRINT OUT RETURN CODE OF
                                       TSO SERVICE FACILITY */
     PUT FILE (FILEOUT) EDIT (' THE FUNCTION RETURN CODE IS ',PARM4)
                       (SKIP, A, F(3)); /* PRINT OUT RETURN CODE OF
                                          IEBCOPY PROGRAM         */
     PUT FILE (FILEOUT) EDIT ('      THE TSF REASON CODE IS ',PARM5)
                       (SKIP, A, F(3)); /* PRINT OUT TSF REASON CODE*/
     PUT FILE (FILEOUT) EDIT ('  THE FUNCTION ABEND CODE IS ',PARM6)
                       (SKIP, A, F(3)); /* PRINT OUT ABEND CODE
                                          FOR IEBCOPY */
 END TSOCALL;
```

PASCAL Program Using TSOLNK to Invoke a Program

```
(*****************************************************************)
(*                                                               *)
(*  THIS PASCAL PROGRAM ISSUES THE IEBCOPY PROGRAM IN TSO/E AND  *)
(*  THEN PRINTS OUT THE COMMAND BUFFER  AND THE RETURN, REASON   *)
(*  AND ABEND CODES RESULTING FROM THE EXECUTION OF THE TSO      *)
(*  SERVICE FACILITY.  TO USE THE EXAMPLE THE USER MUST ALLOCATE *)
(*  THE FOLLOWING FILES:                                         *)
(*       ALLOC F(OUTPUT) DSN(*)                                  *)
(*       ALLOC F(SYSIN) DSN(YOUR.SYSIN)                          *)
(*       ALLOC F(SYSPRINT) DSN(*)                                *)
(*       ALLOC F(INDD) DSN(YOUR.INPDS)                           *)
(*       ALLOC F(OUTDD) DSN(YOUR.OUTPDS)                         *)
(*  THE EXAMPLE REQUIRES THE FOLLOWING CARD IN YOUR.SYSIN FILE:  *)
(*       EXAMPLE    COPY  OUTDD=OUTDD,INDD=INDD                  *)
(*                                                               *)
(*                                                               *)
(*****************************************************************)
PROGRAM TSOINTER;
 TYPE
    PA4=PACKED ARRAY (.1..4.) OF CHAR;
    PA80=PACKED ARRAY (.1..80.) OF CHAR;
(*****************************************************************)
(*                                                               *)
(*  SET UP CALL TO TSOLNK - THE TSO SERVICE FACILITY.            *)
(*  WITH PARAMETER LIST.                                         *)
(*                                                               *)
(*****************************************************************)
 PROCEDURE TSOLNK( VAR PARM1:PA4;
                   VAR PARM2:PA80;
                   VAR PARM3:INTEGER;
                   VAR PARM4:INTEGER;
                   VAR PARM5:INTEGER;
                   VAR PARM6:INTEGER);
 FORTRAN; (* THIS KEYWORD IS REQUIRED TO ESTABLISH LINKAGE TO TSF *)
 VAR
  PARM1:PA4;          (* WORD OF CONTROL BITS              *)
  PARM2:PA80;         (* PROGRAM BUFFER                    *)
  PARM3:INTEGER;      (* LENGTH OF PROGRAM                 *)
  PARM4:INTEGER;      (* FUNCTION RETURN CODE              *)
  PARM5:INTEGER;      (* TSO SERVICE FACILITY REASON CODE  *)
  PARM6:INTEGER;      (* FUNCTION ABEND CODE               *)
  FILEOUT:TEXT;       (* DECLARE OUTPUT FILE NAME          *)
```

Figure 159. PASCAL Program Demonstrating the Use of TSOLNK to Invoke a Program

Figure 'PASCAL Program Demonstrating the Use of TSOLNK to Invoke a Program' (**Continued**)

```
BEGIN
 PARM1(.1.):=CHR(0);     (* ZERO OUT                          *)
 PARM1(.2.):=CHR(0);     (* ZERO OUT BYTE                     *)
 PARM1(.3.):=CHR(1);     (* SPECIFY DUMP                      *)
 PARM1(.4.):=CHR(2);     (* SPECIFY PROGRAM TO BE EXECUTED    *)
 PARM2:='IEBCOPY';       (* FILL IN PROGRAM BUFFER            *)
 PARM3:=7;               (* SPECIFY PROGRAM LENGTH            *)
 PARM4:=0;       (* ZERO OUT FUNCTION RETURN CODE             *)
 PARM5:=0;       (* ZERO OUT TSF REASON CODE                  *)
 PARM6:=0;       (* ZERO OUT FUNCTION ABEND CODE              *)
 TSOLNK(PARM1,
        PARM2,
        PARM3,
        PARM4,
        PARM5,
        PARM6);  (* INTERFACE WITH TSO SERVICE FACILITY       *)
 WRITELN(FILEOUT, 'THE FUNCTION RETURN CODE IS ',PARM4);
                 (* PRINT OUT FUNCTION RETURN CODE            *)
 WRITELN(FILEOUT, '     THE TSF REASON CODE IS ',PARM5);
                 (* PRINT OUT TSF REASON CODE                 *)
 WRITELN(FILEOUT, 'THE FUNCTION  ABEND CODE IS ',PARM6);
                 (* PRINT OUT FUNCTION ABEND CODE             *)
END.
```

COBOL Program Using TSOLNK to Invoke a Program

```
 ID DIVISION.
 PROGRAM-ID.   TSOSVR.
*     THIS COBOL PROGRAM ISSUES THE IEBCOPY PROGRAM IN TSO/E AND
*     THEN PRINTS OUT THE COMMAND BUFFER  AND THE RETURN, REASON
*     AND ABEND CODES RESULTING FROM THE EXECUTION OF THE TSO
*     SERVICE FACILITY.  TO USE THE EXAMPLE THE USER MUST ALLOCATE
*     THE FOLLOWING FILES:
*          ALLOC F(SYSPRT) DSN(*)
*          ALLOC F(SYSIN) DSN(YOUR.SYSIN)
*          ALLOC F(SYSPRINT) DSN(*)
*          ALLOC F(INDD) DSN(YOUR.INPDS)
*          ALLOC F(OUTDD) DSN(YOUR.OUTPDS)
*     THE EXAMPLE REQUIRES THE FOLLOWING CARD IN YOUR.SYSIN FILE:
*          EXAMPLE    COPY  OUTDD=OUTDD,INDD=INDD
*
* THIS PROGRAM WILL RUN ON THE OS/VS COBOL COMPILER RELEASE 3 OR
* HIGHER

 ENVIRONMENT DIVISION.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.

*    DEFINE OUTPUT DEVICE

     SELECT TRMPRT ASSIGN TO UT-S-SYSPRT.
 DATA DIVISION.
 FILE SECTION.

     DEFINE OUTPUT FILE

 FD  TRMPRT
     LABEL RECORDS ARE OMITTED
     RECORD CONTAINS 133 CHARACTERS.

*    DEFINE OUTPUT RECORD

 01  OUTREC.
     02 OUT-LINE     PICTURE X(133).

 WORKING-STORAGE SECTION.

*    DEFINE OUTPUT RECORD FORM

 01  OUT-RECORD.
     02 CONTROL-CHAR PICTURE X VALUE SPACE.
     02 COMMENT      PICTURE X(25).
     02 OUT-VALUE    PICTURE +++++++++9.
     02 FILLER       PICTURE X(111) VALUE SPACES.

*    DEFINE COMMENT VALUES FOR OUTPUT RECORD FORM

 01  RETURN-COMMENT PICTURE X(25) VALUE 'FUNCTION RETURN CODE IS  '.
 01  REASON-COMMENT PICTURE X(25) VALUE '     TSF REASON CODE IS '.
 01  ABEND-COMMENT  PICTURE X(25) VALUE 'FUNCTION ABEND  CODE IS  '.

*    DEFINE FLAGS TO BE FULL WORDS WITH APPROPRIATE BITS ON

 01  FLAG1-ON  PICTURE S9(9) VALUE +16777216 COMP.
 01  FLAG2-ON  PICTURE S9(9) VALUE +65536 COMP.
 01  FLAG3-ON  PICTURE S9(9) VALUE +256 COMP.
 01  FLAG4-ON  PICTURE S9(9) VALUE +2 COMP.
 01  FLAG1-OFF PICTURE S9(9) VALUE +0 COMP.
 01  FLAG2-OFF PICTURE S9(9) VALUE +0 COMP.
 01  FLAG3-OFF PICTURE S9(9) VALUE +0 COMP.
 01  FLAG4-OFF PICTURE S9(9) VALUE +1 COMP.
```

Figure 160. COBOL Program Demonstrating the Use of TSOLNK to Invoke a Program

Figure 'COBOL Program Demonstrating the Use of TSOLNK to Invoke a Program' (**Continued**)

```
* DEFINE STORAGE FOR PARMS
*    PARM1 IS DECIMAL VALUE OF FLAGS
*    PARM2 IS FUNCTION TEXT
*    PARM3 IS FUNCTION LENGTH (SET TO 80)
*    PARM4 IS FUNCTION RETURN CODE VALUE FROM TSOLNK
*    PARM5 IS TSF REASON CODE VALUE FROM TSOLNK
*    PARM6 IS FUNCTION ABEND CODE VALUE FROM TSOLNK

 01  PARM1 PICTURE S9(9) COMP.
 01  PARM2 PICTURE X(80).
 01  PARM3 PICTURE S9(9) VALUE +80 COMP.
 01  PARM4 PICTURE S9(9) VALUE +0 COMP.
 01  PARM5 PICTURE S9(9) VALUE +0 COMP.
 01  PARM6 PICTURE S9(9) VALUE +0 COMP.

 PROCEDURE DIVISION.

*    MOVE DESIRED FUNCTION TO PARM2

 READY-COMMAND.
     MOVE SPACES TO PARM2.
     MOVE 'IEBCOPY' TO PARM2.

*    SET FLAGS BY ADDING APPROPRIATELY VALUED FLAG VARIABLES

 READY-FLAGS.
     MOVE 0 TO PARM1.

*    RESERVED FLAG

     ADD FLAG1-OFF TO PARM1.

*    RESERVED FLAG

     ADD FLAG2-OFF TO PARM1.

*    FLAG3-ON TO REQUEST ABEND WITH DUMP

     ADD FLAG3-ON  TO PARM1.

*    FLAG4-OFF TO REQUEST A TSO/E PROGRAM (NOT A COMMAND) BE INVOKED

     ADD FLAG4-ON TO PARM1.

*    CALL TSOLNK

 CALL-TSOLNK.
     CALL 'TSOLNK' USING PARM1 PARM2 PARM3 PARM4 PARM5 PARM6.

*    PRINT RESULTS

 PRINT-COMMENTS.
     OPEN OUTPUT TRMPRT.

*    PRINT THE FUNCTION RETURN CODE

     MOVE RETURN-COMMENT TO COMMENT.
     MOVE PARM4 TO OUT-VALUE.
     WRITE OUTREC FROM OUT-RECORD.

*    PRINT THE TSF REASON CODE

     MOVE REASON-COMMENT TO COMMENT.
     MOVE PARM5 TO OUT-VALUE.
     WRITE OUTREC FROM OUT-RECORD.

*    PRINT THE FUNCTION ABEND CODE

     MOVE ABEND-COMMENT TO COMMENT.
     MOVE PARM6 TO OUT-VALUE.
     WRITE OUTREC FROM OUT-RECORD.

     CLOSE TRMPRT.
     STOP RUN.
```

PL/I Program Using TSOLNK to Invoke a CLIST

```
/*********************************************************************/
/*  THIS PL/I PROGRAM INTERFACES WITH THE TSO SERVICE FACILITY.      */
/*  THE PROGRAM WILL EXECUTE MYCLIST AND THEN PRINT OUT THE          */
/*  RETURN, REASON AND ABEND CODES AS A RESULT OF USING THE          */
/*  SERVICE.  SINCE THIS PROGRAM DOES ITS OWN I/O AFTER CALLING THE */
/*  TSO SERVICE FACILITY, THE USER MUST ALLOCATE THE FILE NAME       */
/*  FILEOUT, PREFERABLY TO THE TERMINAL WITH THE TSO/E COMMAND:      */
/*        ALLOC F(FILEOUT) DSN(*)                                    */
/*                                                                   */
/*  THIS PROGRAM WILL RUN ON THE OS/VS PL/I COMPILER RELEASE 2 OR    */
/*  HIGHER.                                                          */
/*********************************************************************/
TSOCALL:
  PROCEDURE OPTIONS(MAIN);
/*********************************************************************/
/*  DECLARE PARAMETERS                                               */
/*********************************************************************/
DECLARE 1 PARM1,
          2 PARM11 FIXED BINARY (15,0), /* RESERVED                 */
          2 PARM13 BIT(8),              /* ABEND FLAG               */
                                        /* 0 -ABEND WITHOUT DUMP    */
                                        /* 1 -ABEND WITH DUMP       */
          2 PARM14 BIT(8);              /* FUNCTION CODE            */
DECLARE PARM2 CHARACTER(8);             /* NAME OF FUNCTION         */
DECLARE PARM3 FIXED BINARY(31,0);       /* LENGTH OF CMD OR PROG    */
DECLARE PARM4 FIXED BINARY(31,0);       /* FUNCTION RETURN CODE     */
DECLARE PARM5 FIXED BINARY(31,0);       /* TSF REASON CODE          */
DECLARE PARM6 FIXED BINARY(31,0);       /* FUNCTION ABEND CODE      */
/*********************************************************************/
/*  DECLARE OUTPUT FILE                                              */
/*********************************************************************/
DECLARE (FILEOUT) FILE;
/*********************************************************************/
/*  DECLARE TSOLNK ROUTINE PARAMETER LIST                            */
/*********************************************************************/
DECLARE TSOLNK ENTRY(                   /*                          */
  1,                                    /* STRUCTURE OF 4 BYTES     */
    2 FIXED BINARY(15,0),               /* BYTE 1 RESERVED          */
    2 BIT(8),                           /* BYTE 3 ABEND FLAG        */
    2 BIT(8),                           /* BYTE 4 FUNCTION FLAG     */
  CHARACTER (*),                        /* NAME OF PROGRAM OR CMD   */
  FIXED BINARY(31,0),                   /* LENGTH OF CMD OR PROG    */
  FIXED BINARY(31,0),                   /* FUNCTION RETURN CODE     */
  FIXED BINARY(31,0),                   /* REASON CODE              */
  FIXED BINARY(31,0)                    /* FUNCTION ABEND CODE      */
  )
EXTERNAL OPTIONS(ASSEMBLER RETCODE INTER);
DECLARE PLIRETV BUILTIN;
```

Figure 161. PL/I Program Demonstrating the Use of TSOLNK to Invoke a CLIST

Figure 'PL/I Program Demonstrating the Use of TSOLNK to Invoke a CLIST' (**Continued**)

```
/*********************************************************************/
/*  START OF EXECUTABLE CODE                                         */
/*********************************************************************/
PARM13='00000001'B;               /* DUMP YES OR NO, SET TO NO     */
PARM14='00000101'B;               /* INDICATE FUNCTION IS A CLIST  */
PARM2 ='MYCLIST';                 /* FUNCTION NAME                 */
PARM3 = 7;                        /* COMMAND LENGTH                */
/*********************************************************************/
/* CALL TSO SERVICE FACILITY                                         */
/*********************************************************************/
CALL TSOLNK(PARM1,PARM2,PARM3,PARM4,PARM5,PARM6);
/*********************************************************************/
```

```
/* PRINT RESULTS OF CALL                                             */
/* PRINT OUT RETURN CODE OF TSO SERVICE FACILITY                     */
/*********************************************************************/
PUT FILE (FILEOUT) EDIT (' THE TSOLNK RETURN CODE IS ',PLIRETV)
                        (A,F(3));
/*********************************************************************/
/* PRINT OUT RETURN CODE OF MYCLIST                                  */
/*********************************************************************/
PUT FILE (FILEOUT) EDIT (' THE FUNCTION RETURN CODE IS ',PARM4)
                        (SKIP, A, F(3));
/*********************************************************************/
/* PRINT OUT REASON CODE OF MYCLIST                                  */
/*********************************************************************/
PUT FILE (FILEOUT) EDIT ('      THE TSF REASON CODE IS ',PARM5)
                        (SKIP, A, F(3));
/*********************************************************************/
/* PRINT OUT ABEND CODE OF MYCLIST                                   */
/*********************************************************************/
PUT FILE (FILEOUT) EDIT (' THE FUNCTION ABEND CODE IS ',PARM6)
                        (SKIP, A, F(3));
 END TSOCALL;
```

PL/I Program Calling a CLIST

```
/*********************************************************************/
/*  THIS CLIST IS CALLED BY PL/I PROGRAM, TSOCALL. AFTER IT IS CALLED, */
/*  IT ALLOCATES THE NECESSARY DATA SETS FOR FORTRAN PROGRAM, MYPGM,   */
/*  CALLS MYPGM, AND TRANSMITS THE RESULTS OF MYPGM TO ANOTHER USER,   */
/*  HISID AT HISNODE.  NOTE: THE DATA SETS FOR FORTRAN PROGRAM, MYPGM, */
/*  MUST ALREADY EXIST AND BE CATALOGED.                             */
/*********************************************************************/
 PROC 0

 ALLOC F(FT06FT01) DSN(*) REUSE
 ALLOC F(SYSIN) DSN(YOUR.SYSIN) REUSE
 ALLOC F(SYSPRINT) DSN(*) REUSE
 ALLOC F(INDD) DSN(YOUR.INPDS) REUSE
 ALLOC F(OUTDD) DSN(YOUR.OUTPDS) REUSE

 CALL LOAD(MYPGM)

 TRANSMIT HISNODE.HISID DA(YOUR.OUTPDS)  /* SEND RESULTS */

 EXIT CODE(0)
```

Figure 162. MYCLIST called by PL/I program, TSOCALL

PASCAL Program Using TSOLNK to Invoke a CLIST

```
(*********************************************************************)
(*                                                                   *)
(*  THIS PASCAL PROGRAM WILL INTERFACE WITH THE TSO SERVICE          *)
(*  FACILITY. THIS PROGRAM WILL EXECUTE A CLIST WITH MEMBER NAME,    *)
(*  MYCLIST. THE CLIST LIBRARY CONTAINING MYCLIST HAS ALREADY        *)
(*  BEEN ALLOCATED TO FILE SYSPROC.  THIS PROGRAM DOES ITS           *)
(*  OWN I/O AFTER THE TSO/E COMMAND IS EXECUTED TO DISPLAY           *)
(*  RETURN CODES, THE USER SHOULD ALLOCATE FILE "FILEOUT"            *)
(*  TO THE TERMINAL.                                                 *)
(*                                                                   *)
(*                                                                   *)
(*********************************************************************)
PROGRAM TSOINTER;
 TYPE
    PA4=PACKED ARRAY (.1..4.) OF CHAR;
    PA80=PACKED ARRAY (.1..80.) OF CHAR;
```

```
(*********************************************************************)
(*                                                                   *)
(*  SET UP CALL TO TSOLNK - THE TSO SERVICE FACILITY.                *)
(*  WITH PARAMETER LIST.                                             *)
(*                                                                   *)
(*********************************************************************)
 PROCEDURE TSOLNK( VAR PARM1:PA4;
                   VAR PARM2:PA80;
                   VAR PARM3:INTEGER;
                   VAR PARM4:INTEGER;
                   VAR PARM5:INTEGER;
                   VAR PARM6:INTEGER);
 FORTRAN; (* THIS KEYWORD IS REQUIRED TO ESTABLISH LINKAGE TO TSF *)
```

```
 VAR
  PARM1:PA4;               (* WORD OF CONTROL BITS              *)
  PARM2:PA80;              (* COMMAND BUFFER                    *)
  PARM3:INTEGER;           (* LENGTH OF COMMAND                 *)
  PARM4:INTEGER;           (* FUNCTION RETURN CODE              *)
  PARM5:INTEGER;           (* TSO SERVICE FACILITY REASON CODE  *)
  PARM6:INTEGER;           (* FUNCTION ABEND CODE               *)
  FILEOUT:TEXT;            (* DECLARE OUTPUT FILE NAME          *)
```

```
   BEGIN
    PARM1(.1.):=CHR(0);     (* ZERO OUT                          *)
    PARM1(.2.):=CHR(0);     (* ZERO OUT BYTE                     *)
    PARM1(.3.):=CHR(1);     (* SPECIFY DUMP                      *)
    PARM1(.4.):=CHR(5);     (* SPECIFY CLIST TO BE EXECUTED      *)
    PARM2:='MYCLIST';       (* FILL IN COMMAND BUFFER            *)
    PARM3:=7;               (* SPECIFY COMMAND LENGTH            *)
    PARM4:=0;       (* ZERO TSO/E FUNCTION RETURN CODE           *)
    PARM5:=0;       (* ZERO TSO SERVICE FACILITY REASON CODE     *)
    PARM6:=0;       (* ZERO TSO/E FUNCTION ABEND CODE            *)
    TSOLNK(PARM1,
           PARM2,
           PARM3,
           PARM4,
           PARM5,
           PARM6);  (* INTERFACE WITH TSO SERVICE FACILITY       *)
    WRITELN(FILEOUT, 'THE FUNCTION RETURN CODE IS ',PARM4);
                    (* PRINT OUT FUNCTION RETURN CODE            *)
    WRITELN(FILEOUT, '     THE TSR REASON CODE IS ',PARM5);
                    (* PRINT OUT TSR REASON CODE                 *)
    WRITELN(FILEOUT, 'THE FUNCTION  ABEND CODE IS ',PARM6);
                    (* PRINT OUT FUNCTION ABEND CODE             *)
   END.
```

Figure 163. PASCAL Program Demonstrating the Use of TSOLNK to Invoke a CLIST

Assembler Program Using IKJEFTSR to Invoke a REXX Exec

```
*********************************************************************
*                                                                   *
*  THIS ASSEMBLER PROGRAM INVOKES THE TSO SERVICE FACILITY TO PASS  *
*  CONTROL TO A REXX EXEC CALLED MYEXEC.  IN THIS EXAMPLE, MYEXEC   *
*  IS A FULLSCREEN APPLICATION THAT ISSUES ISPF COMMANDS.  THIS     *
*  SAMPLE PROGRAM THEREFORE REQUESTS THAT THE TSO SERVICE FACILITY  *
*  INVOKE THE EXEC FROM AN UNAUTHORIZED ENVIRONMENT.                *
*                                                                   *
*********************************************************************
TSF      CSECT
         STM   R14,R12,12(R13)
         BALR  R12,0
         USING *,R12
         ST    R13,SAVEAREA+4
         LA    R11,SAVEAREA
         ST    R11,8(,R13)
         LA    R13,SAVEAREA
*
```

```
*
MAIN     DS    0H
*        .
*        .
*        .
         L     R15,CVTPTR              ESTABLISH
         L     R15,CVTTVT(,R15)        ADDRESSABILITY TO THE
         L     R15,TSVTASF-TSVT(,R15)  TSO SERVICE FACILITY
*
*        INVOKE THE TSO SERVICE FACILITY -- EXECUTE "MYEXEC" EXEC
*
         CALL  (15),(FLAGS,CMDBUF,BUFLEN,RETCODE,RSNCODE,ABNDCODE),VL
         LTR   R15,R15            CHECK TSR RETURN CODE
         BNZ   ERRORRTN           BAD RETURN CODE FROM TSR
         CLC   RETCODE,ZERO       CHECK FOR EXEC ERROR
         BH    ERRORCMD           BAD RETURN CODE FROM EXEC
         B     ENDUP              NO ERROR --- EXIT
ERRORRTN DS   0H
```

```
*********************************************************************
*        ANALYZE TSO SERVICE FACILITY ERROR                         *
*        .                                                          *
*        .                                                          *
*        .                                                          *
*                                                                   *
*********************************************************************
         B    ENDUP
ERRORCMD DS   0H
```

```
*********************************************************************
*        ANALYZE EXEC ERROR                                         *
*        .                                                          *
*        .                                                          *
*        .                                                          *
*********************************************************************
```

```
ENDUP    DS   0H
         L    R13,4(,R13)
         LM   R14,R12,12(R13)
         SLR  R15,R15
         BR   R14
```

Figure 164. Assembler Language Program Demonstrating the Use of IKJEFTSR to Invoke a REXX Exec

Figure 'Assembler Language Program Demonstrating the Use of IKJEFTSR to Invoke a REXX Exec' (**Continued**)

```
*********************************************************************
*                                                                   *
*        DATA AREAS                                                 *
*                                                                   *
*********************************************************************
ZERO     DC    F'0'        ZERO CONSTANT
FLAGS    DS    0F          MAPS FIRST PARM TO IKJEFTSR
RESFLAGS DC    X'00'       FIRST BYTE IS RESERVED
OPTFLAGS DC    X'01'       TELL TSR TO ESTABLISH AN UNAUTHORIZED
*                          ENVIRONMENT
ABFLAGS  DC    X'01'       DUMP IF ABEND OCCURS
FNCFLAGS DC    X'01'       TELL TSR TO EXECUTE THE EXEC
         *
CMDBUF   DC    C'MYEXEC'   NAME OF EXEC TO BE EXECUTED
         *
BUFLEN   DC    F'6'        LENGTH OF COMMAND BUFFER
RETCODE  DS    F           RETURN CODE FROM EXEC
RSNCODE  DS    F           REASON CODE
ABNDCODE DS    F           ABEND CODE
SAVEAREA DS    18F         SAVE AREA
         .
         .
         .
```

```
CVTPTR   EQU   16
CVTTVT   EQU   X'9C'
R15      EQU   15
R14      EQU   14
R13      EQU   13
R12      EQU   12
R11      EQU   11
R9       EQU   9
R8       EQU   8
         IKJTSVT
         END
```

Chapter 24. Using the variable access routine IKJCT441

This chapter describes how to use the variable access routine (IKJCT441) in an application program to examine and manipulate CLIST and REXX variables.

Functions Provided by IKJCT441

This service allows any application program to examine and manipulate CLIST and REXX variables. IKJCT441 provides the following functions:

- It updates or creates a variable value (entry code TSVEUPDT). If the variable does not exist, IKJCT441 creates it.
- It returns a variable value (entry code TSVERETR). If the variable does not exist, IKJCT441 creates it (implicit creation).
- It returns a variable value (entry code TSVNOIMP). If the variable does not exist, IKJCT441 does *not* create it (no implicit creation), but indicates this by a return code.
- It returns all active variables and their values. (Entry code TSVELOC).

Note: REXX execs and CLISTs cannot access each others variables.

IKJCT441 allows an application program to request, in one invocation, a combination of the functions described above. To perform multiple functions in one invocation, specify a list of individual requests. IKJCT441 performs each function in the order you specify and continues processing each request regardless of the return codes from previous requests.

Some variables are called control variables. Control variables are variables that have a special meaning in a CLIST or REXX exec. Generally, they provide information about the environment during execution. You can change or assign values to only some of these control variables. For CLISTs, see *z/OS TSO/E CLISTs* for lists of the control variables that you can and cannot modify. For REXX execs, see *z/OS TSO/E REXX Reference*.

Note: &SYSOUTLINE is a CLIST control variable that saves TSO/E command output and allows a CLIST or application program to display the output. When a CLIST executes a TSO/E command, it resets the &SYSOUTLINE control variable to zero. However, if a CLIST invokes a program containing TSO/E commands, the program does not reset &SYSOUTLINE to zero for each TSO/E command. To save command output lines in a non-CLIST program, use IKJCT441 to reset &SYSOUTLINE to zero for each TSO/E command. See *z/OS TSO/E CLISTs* for more information about &SYSOUTLINE.

Some CLIST control variables, and CLIST built-in functions, require evaluation before their values can be obtained. Their values cannot be retrieved by IKJCT441. For a list of control variables whose values cannot be retrieved by IKJCT441, see *z/OS TSO/E CLISTs*.

Considerations for Accessing REXX Variables

All commands and programs that invoke IKJCT441 to access REXX variables must be in 31-bit addressing mode.

Programs or commands that are directly invoked from a REXX exec can access only those variables that have valid REXX names. These programs or commands can use IKJCT441 to set and retrieve symbols. IKJCT441 uses the REXX direct interface (rather than the symbolic interface). No substitution or case translation takes place. For more information about the direct interface, see *z/OS TSO/E REXX Reference*.

Authorized programs or commands that are directly invoked from a REXX exec can access variables created by the REXX exec only if the variable names begin with 'SYSAUTH'. However, an authorized program or command can access all variables created by the program or command. Furthermore,

the authorized command or program can set a variable, even one whose name does not begin with 'SYSAUTH', for use by the exec.

Authorized programs or commands invoked from a REXX exec cannot set stems. (A REXX stem is a variable name which contains a single period, which is the last character of the name.) However, these programs or commands can use IKJCT441 to set a compound variable which represents a particular instance of the stem. Also, these programs or commands can use IKJCT441 to retrieve stem variables by specifying the entry code TSVNOIMP in the IKJCT441 parameter list.

For example, an authorized program or command cannot use IKJCT441 to set a value for a stem like 'A.'. However, it can use IKJCT441 to set values for the compound variables 'A.1', 'A.2', or 'A.THIRD' (that is, particular instances of the stem 'A.').

Authorized programs or commands can access variables containing output from the OUTTRAP statement if the variables are created by the exec that invoked the program or command.

Note:

1. If IKJCT441 is used to fetch an uninitialized REXX variable, the value returned is a null value, rather than the name of the variable itself.
2. IKJCT441 can be used to access variables with DBCS names when the underlying REXX exec has enabled the use of DBCS variable names by coding the OPTIONS ETMODE instruction.

Unauthorized programs and commands can use either IKJCT441 or another TSO/E service, IRXEXCOM, to access REXX variables. For information about using IRXEXCOM, see *z/OS TSO/E REXX Reference*.

Passing Control to IKJCT441

Your program can access CLIST or REXX variables by using either the CALL or LINK macro instructions, specifying IKJCT441 as the entry point name. You must also create a parameter list to send input and receive output from IKJCT441.

Callers executing in 31-bit addressing mode can pass data residing above 16 MB in virtual storage as input to IKJCT441. The caller's parameters must be in the primary address space.

Your program can obtain the address of IKJCT441 from the TSVTVACC field in the TSO/E vector table (TSVT). Figure 165 on page 411 shows how to obtain this address.

IBM suggests that commands and programs that invoke IKJCT441 to access CLIST or REXX variables should be in 31-bit addressing mode when calling IKJCT441. Callers of IKJCT441 that access REXX variables are required to be in 31-bit addressing mode. Any program that invokes IKJCT441 to retrieve a CLIST or REXX variable while the user's TSO/E PROFILE is set to VARSTORAGE(HIGH) receives a failing return code from IKJCT441, if the calling program is running in 24-bit addressing mode when IKJCT441 is called. (This is due to the fact that PROFILE VARSTORAGE(HIGH) allows the CLIST and authorized REXX variable pools to be kept in storage above the 16MB line. So the variables returned when VARSTORAGE(HIGH) is set will usually be in 31-bit addressable storage, and therefore, they will not be accessible to 24-bit callers.)

Note: IKJCT441 only supports 24-bit and 31-bit addressing mode callers.

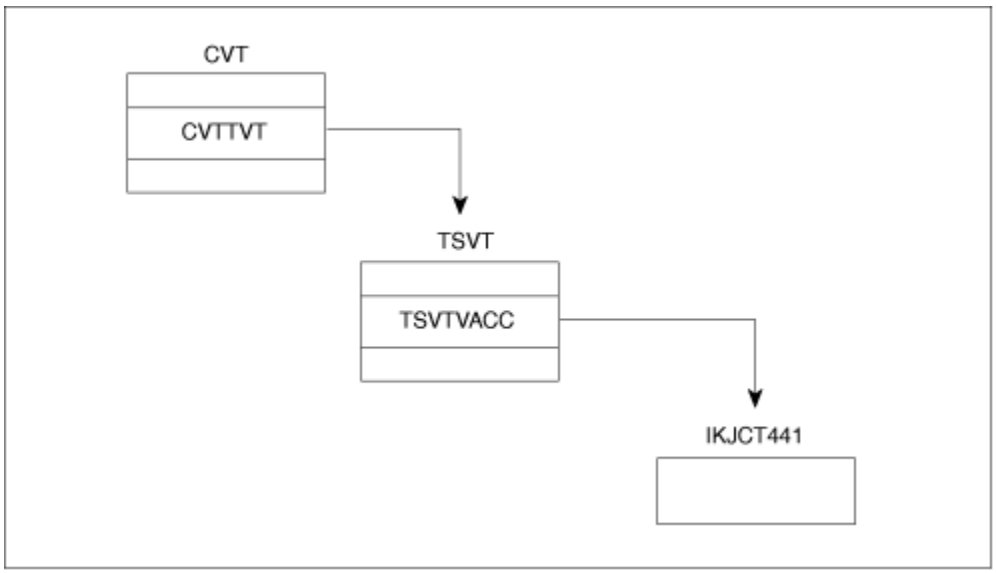


Figure 165. Obtaining the Address of IKJCT441

The IKJCT441 Parameter List

Figure 166 on page 412 describes the format of the caller's parameter list for accessing variables.

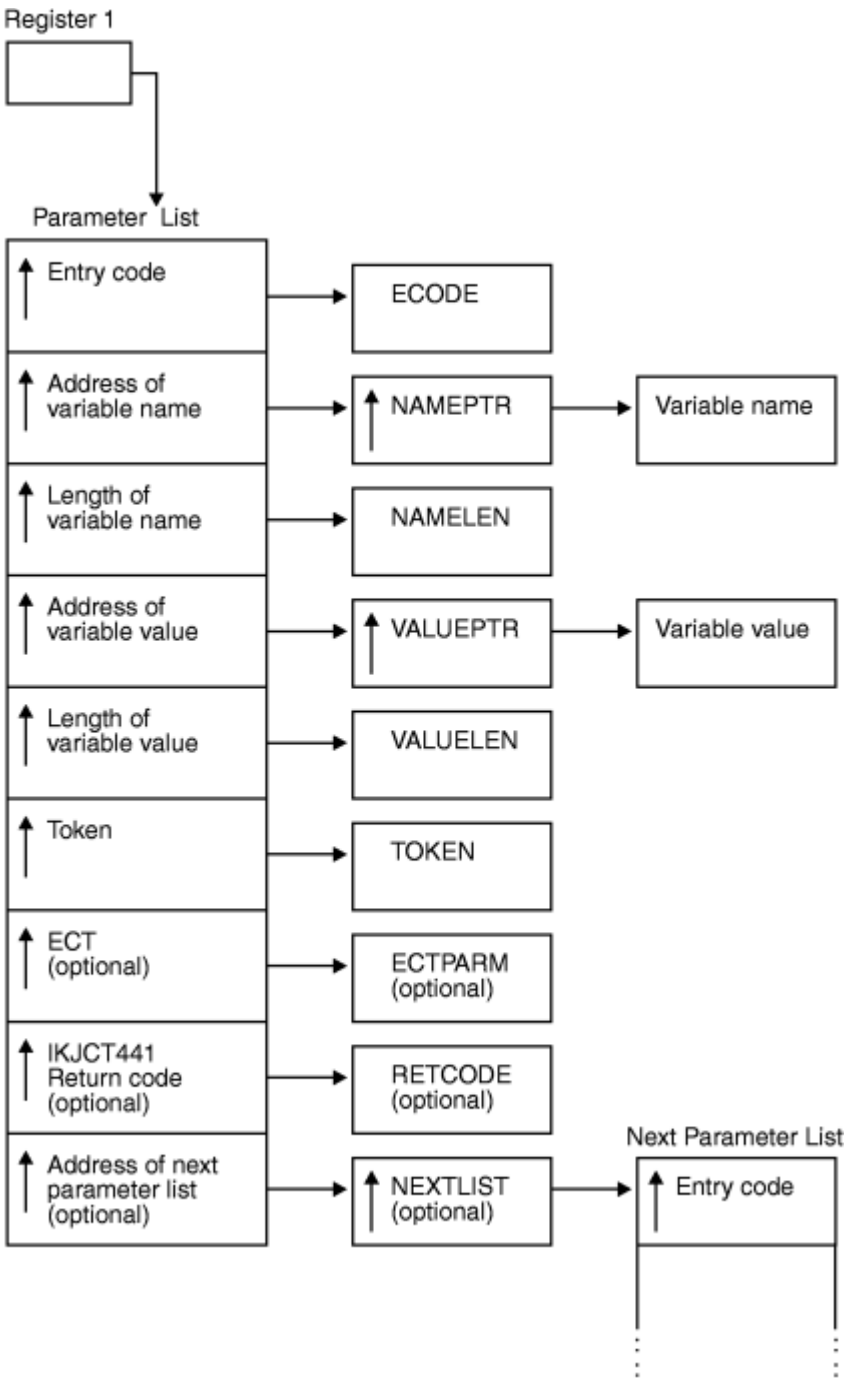


Figure 166. Parameter List Structure for IKJCT441

The parameter list consists of a list of fullword pointers to the actual parameters. The parameter list can be of variable length; therefore, the caller must turn on the high-order bit of the last address in a parameter list to indicate that it is the last address in a list.

A parameter list represents a *single* request to IKJCT441. You can chain *multiple* requests to IKJCT441 by chaining multiple parameter lists (see Figure 167 on page 414 for an example). Note that *each* parameter list must be terminated with the high-order bit turned on.

Table 124 on page 413 shows the symbolic names and descriptions of the caller's parameters. Note that the table describes the parameters pointed to by the parameter list entries, not the parameter list entries

itself. Some of the parameters are by themselves pointers to the actual data (NAMEPTR, VALUEPTR, NEXTLIST).

Table 124. The parameters for IKJCT441

Parameter	Function
ECODE	Entry code. The entry code is a number that indicates to IKJCT441 the function being requested. The entry codes are defined by the mapping macro IKJTSVT. TSO/E supports the following entry codes only: **Entry Code** **Meaning** **TSVEUPDT** Update or create a variable. If the variable does not exist, IKJCT441 should create it. **TSVERETR** Return a variable value. If the variable does not exist, IKJCT441 should create it. **TSVNOIMP** Return a variable. If the variable does not exist, IKJCT441 should not create it. **TSVELOC** Return all active variables and their values.
NAMEPTR	Address of the variable name.
NAMELEN	Length of the variable name.
VALUEPTR	Address of the variable value.
VALUELEN	Length of the variable value.
TOKEN	IKJCT441 uses this value only when finding all active CLIST variables.
ECTPARM	Optional parameter. Contains the environment control table (ECT) to be used. If it is not specified, IKJCT441 uses the system ECT, that is, the LWAPECT. If one of the following parameters are used, and you do not want to specify an ECT (that is, you want IKJCT441 to use the system ECT), the value of ECTPARM must be a fullword containing X'FFFFFFFF'.
RETCODE	Optional parameter. IKJCT441 places the return code from the function that was requested in this parameter. If the NEXTLIST parameter is to be used to chain a list of requests, you must supply the pointer to the RETCODE parameter. This parameter is useful when you use IKJCT441 to perform a combination of functions, that is, when you specify a list of individual requests. It allows you to keep the return codes from the individual requests. IKJCT441 always places a return code in register 15. If the caller requests that a list of functions be performed, register 15 holds the *first* non-zero return code issued by IKJCT441 when processing the individual requests. Further return codes from individual requests do not change the content of register 15. If all individual request are performed successfully, register 15 contains a return code of 0. If the calling program specifies the RETCODE parameter, IKJCT441 also places the return code in the RETCODE parameter. This allows the calling program to take the appropriate actions on return codes from individual requests.
NEXTLIST	Optional parameter. Used if IKJCT441 is to perform *multiple* requests in one invocation. NEXTLIST contains the address of the next parameter list. If *no* next parameter list is to be used, terminate the parameter list at the previous entry. If you code, for whatever reason, the last parameter list entry (pointer to NEXTLIST), ensure that NEXTLIST is a fullword containing X'00000000'.

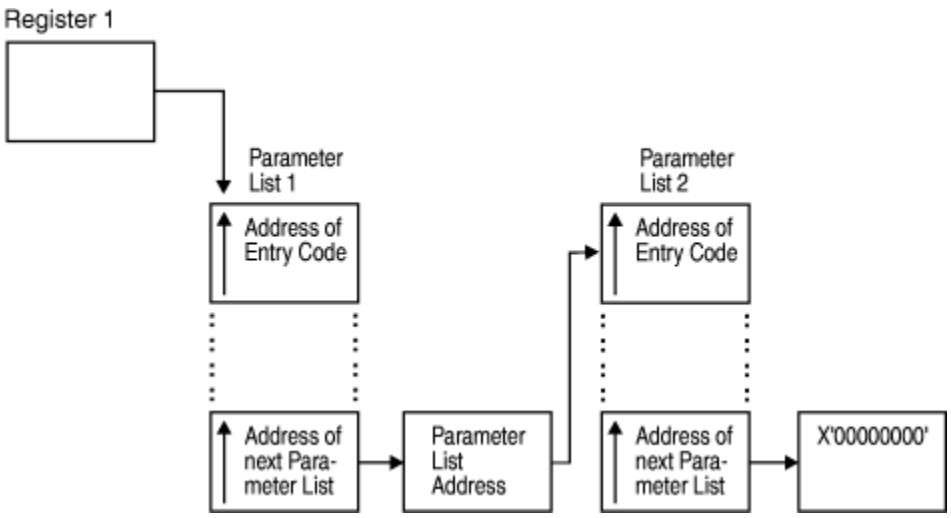


Figure 167. Example – Chain of Two Elements to IKJCT441

Updating or Creating a Variable Value (TSVEUPDT)

Before invoking IKJCT441 to update or create a variable, the caller must:

- Specify at least the first six parameters in the parameter list.
- Specify entry code TSVEUPDT.
- Specify, for the variable to be updated or created:
  - The address of the variable name (NAMEPTR)
  - The length of the variable name (NAMELEN)
  - The address of the variable value (VALUEPTR)
  - The length of the variable value (VALUEPTR)
  - The variable value (labeled VALUE in "Examples Using IKJCT441" on page 420)
- Set the value of TOKEN to zero.
- Turn on the high-order bit of the last word of the parameter list.

Output from IKJCT441 on Entry Code TSVEUPDT

If the caller has specified the RETCODE parameter, RETCODE contains the return code for the request.

Return Codes from IKJCT441 on Entry Code TSVEUPDT

IKJCT441 always places a return code in register 15. If the caller requests that a list of functions be performed, register 15 holds the *first* non-zero return code issued by IKJCT441 when processing the individual requests. Further return codes from individual requests do not change the content of register 15. If all individual request are performed successfully, register 15 contains a return code of 0.

If the calling program specifies the RETCODE parameter, IKJCT441 also places the return code in the RETCODE parameter. This allows the calling program to take the appropriate actions on return codes from individual requests.

Table 125. Return codes from IKJCT441 (entry code TSVEUPDT)

Return code dec(Hex)	Meaning
0(0)	IKJCT441 updated or created the variable.

Table 125. Return codes from IKJCT441 (entry code TSVEUPDT) (continued)

Return code dec(Hex)	Meaning
12(C)	The variable is a label, and IKJCT441 did not update it.
16(10)	The variable is a CLIST built-in function or a control variable that the user cannot modify, such as &SYSDATE, and IKJCT441 did not update it.
24(18)	The variable is a procedure name, and IKJCT441 did not update it.
32(20)	A storage management (GETMAIN/FREEMAIN) failure occurred.
36(24)	For CLIST variables: • The length of the variable name is less than 1 or greater than 252. • The length of the variable value is less than zero or greater than 32,767. For REXX variables: • See return code 80(50) below. • The length of the variable value is less than zero or greater than 32,767.
40(28)	One of the following situations occurred: • The caller's parameter list contains an error. • The caller of IKJCT441 is not activated via a CLIST or REXX exec. • The caller attempted to access a REXX variable pool while another program or REXX exec was accessing the same variable pool.
44(2C)	The entry code is not valid.
80(50)	The variable name is not valid for REXX, or the length of the REXX variable name is less than 1 or greater than 250.
81(51)	A TSO/E REXX routine issued a failing return code.

Returning the Value of a Variable (TSVERETR) - Create

IKJCT441 creates the variable if it does not exist.

Before invoking IKJCT441 to return the value of a variable, the caller must:

- Specify at least the first six parameters in the parameter list.
- Specify entry code TSVERETR.
- Specify, for the variable value to be returned:
  - Address of the variable name (NAMEPTR)
  - Length of the variable name (NAMELEN).
  - The address of the variable value (VALUEPTR)
- Specify, in case the variable is to be created:
  - The length of the variable value (VALUELEN)
  - The variable value (labeled VALUE in “Examples Using IKJCT441” on page 420)
- Set the value of TOKEN to zero.
- Turn on the high-order bit of the last word of the parameter list.

Output from IKJCT441 on Entry Code TSVERETR

IKJCT441 returns values for the following parameters unless specified otherwise by the return code:

- VALUEPTR contains the address of the value of the variable.
- VALUELEN contains the length of the variable value.
- If the caller has specified the RETCODE parameter, RETCODE contains the return code for the request.

Return Codes from IKJCT441 on Entry Code TSVERETR

IKJCT441 always places a return code in register 15. If the caller requests that a list of functions be performed, register 15 holds the *first* non-zero return code issued by IKJCT441 when processing the individual requests. Further return codes from individual requests do not change the content of register 15. If all individual request are performed successfully, register 15 contains a return code of 0.

If the calling program specifies the RETCODE parameter, IKJCT441 also places the return code in the RETCODE parameter. This allows the calling program to take the appropriate actions on return codes from individual requests.

Table 126. Return codes from IKJCT441 (entry code TSVERETR)

Return code dec(Hex)	Meaning
0(0)	IKJCT441 successfully returned the variable.
4(4)	The caller should not rescan the variable. It is an I/O variable containing an & and is not a variable name.
8(8)	The variable is a control variable or a CLIST built-in function, such as &STR, that requires evaluation. IKJCT441 cannot evaluate the variable. IKJCT441 will not update VALUEPTR and VALUELEN.
12(C)	The variable is a label. IKJCT441 updated VALUEPTR and VALUELEN, but the value of the variable is meaningless.
24(18)	The variable is a procedure name. IKJCT441 updated VALUEPTR and VALUELEN, but the value of the variable is meaningless.
36(24)	The length of the variable is less than 1 or greater than 252.
40(28)	One of the following situations occurred: • The caller's parameter list contains an error. • The caller of IKJCT441 is not activated via a CLIST or REXX exec. • The caller attempted to access a REXX variable pool while another program or REXX exec was accessing the same variable pool. IKJCT441 did not update VALUEPTR and VALUELEN.
44(2C)	The entry code is not valid. IKJCT441 did not update VALUEPTR and VALUELEN.
76(4C)	The variable was not found. Because it has the prefix SYSX, it is assumed that this variable is an installation written built-in function. It has not been added to the variable pool.
80(50)	The variable name is not valid for REXX, or the length of the REXX variable name is less than 1 or greater than 250.
81(51)	A TSO/E REXX routine issued a failing return code.

Table 126. Return codes from IKJCT441 (entry code TSVERETR) (continued)	
Return code dec(Hex)	**Meaning**
88(58)	A variable value is not returned. The caller is running AMODE24, but the variable requested might be in 31-bit addressable storage. The caller should change to AMODE31 before calling IKJCT441 so that any variables returned are addressable by the caller. If you are unable to invoke IKJCT441 in AMODE31, you can force variables to be kept in 24-bit addressable storage by having the user set the user PROFILE to VARSTORAGE(LOW) before running the application that creates the variable you are trying to access.

Returning the Value of a Variable (TSVNOIMP) - No Create

IKJCT441 does *not* create the variable if it does not exist.

Before invoking IKJCT441 to return the value of a variable, the caller must:

- Specify at least the first six parameters in the parameter list.
- Specify entry code TSVNOIMP.
- Specify, for the variable value to be returned:
  - Address of the variable name (NAMEPTR)
  - Length of the variable name (NAMELEN).
  - The address of the variable value (VALUEPTR)
- Set the value of TOKEN to zero.
- Turn on the high-order bit of the last word of the parameter list.

Output from IKJCT441 on Entry Code TSVNOIMP

IKJCT441 returns values for the following parameters unless specified otherwise by the return code:

- VALUEPTR contains the address of the value of the variable.
- VALUELEN contains the length of the variable value.
- If the caller has specified the RETCODE parameter, RETCODE contains the return code for the request.

Return Codes from IKJCT441 on Entry Code TSVNOIMP

IKJCT441 always places a return code in register 15. If the caller requests that a list of functions be performed, register 15 holds the *first* non-zero return code issued by IKJCT441 when processing the individual requests. Further return codes from individual requests do not change the content of register 15. If all individual request are performed successfully, register 15 contains a return code of 0.

If the calling program specifies the RETCODE parameter, IKJCT441 also places the return code in the RETCODE parameter. This allows the calling program to take the appropriate actions on return codes from individual requests.

Table 127. Return codes from IKJCT441 (entry code TSVNOIMP)	
Return code dec(Hex)	**Meaning**
0(0)	IKJCT441 successfully returned the variable.

Table 127. Return codes from IKJCT441 (entry code TSVNOIMP) (continued)

Return code dec(Hex)	Meaning
4(4)	The caller should not rescan the variable. It is an I/O variable containing an & and is not a variable name.
8(8)	The variable is a control variable or a CLIST built-in function, such as &STR, that requires evaluation. IKJCT441 cannot evaluate the variable. IKJCT441 did not update VALUEPTR and VALUELEN.
12(C)	The variable is a label. IKJCT441 updated VALUEPTR and VALUELEN, but the value of the variable is meaningless.
24(18)	The variable is a procedure name. IKJCT441 updated VALUEPTR and VALUELEN, but the value of the variable is meaningless.
36(24)	The length of the variable is less than 1 or greater than 252.
40(28)	One of the following situations occurred: • The caller's parameter list contains an error. • The caller of IKJCT441 is not activated via a CLIST or REXX exec. • The caller attempted to access a REXX variable pool while another program or REXX exec was accessing the same variable pool. IKJCT441 did not update VALUEPTR and VALUELEN.
44(2C)	The entry code is not valid. IKJCT441 did not update VALUEPTR and VALUELEN.
52(34)	The variable does not exist, and IKJCT441 did not create it.
80(50)	The variable name is not valid for REXX, or the length of the REXX variable name is less than 1 or greater than 250.
81(51)	A TSO/E REXX routine issued a failing return code.
88(58)	A variable value is not returned. The caller is running AMODE24, but the variable requested might be in 31-bit addressable storage. The caller should change to AMODE31 before calling IKJCT441 so that any variables returned are addressable by the caller. If you are unable to invoke IKJCT441 in AMODE31, you can force variables to be kept in 24-bit addressable storage by having the user set the user PROFILE to VARSTORAGE(LOW) before running the application that creates the variables you are trying to access.

Returning all Active Variables and their Values (TSVELOC)

To list all the CLIST or REXX variables and their values, the caller can either invoke IKJCT441 once for each existing variable, or invoke IKJCT441 once with a chain of requests.

If IKJCT441 is invoked multiple times, the caller must set TOKEN to zero before invoking IKJCT441 for the first time. Similarly, if IKJCT441 is invoked with a chain of requests, the caller must initialize TOKEN to zero for the first element in the chain. IKJCT441 places a value in TOKEN and uses this value on subsequent invocations to find the next variable. The caller must not change the value that IKJCT441 places in TOKEN. When there are no more variables, IKJCT441 places a zero in TOKEN and sets the appropriate return code.

If IKJCT441 is invoked to perform a chain of requests, the caller must pass, for each element of the chain, the same address for TOKEN (parameter 6).

Before invoking IKJCT441 to find all the CLIST or REXX variables, the caller must:

- Specify the entry code TSVELOC.
- Set TOKEN to zero on the first entry, or in the first element for a chain of requests.
- Turn on the high-order bit of the last word of the parameter list.

Output from IKJCT441 on Entry Code TSVELOC

IKJCT441 returns values for the following parameters unless specified otherwise by the return code:

- NAMEPTR contains the address of the variable name.
- NAMELEN contains the variable name length.
- VALUEPTR contains the address of the value of the variable.
- VALUELEN contains the variable value length.
- TOKEN contains zero, or if CLIST variables are to be returned, TOKEN contains an internal value that identifies the next variable.
- If the caller has specified the RETCODE parameter, RETCODE contains the return code for the request.

Return Codes From IKJCT441 on Entry Code TSVELOC

IKJCT441 always places a return code in register 15. If the caller requests that a list of functions be performed, register 15 holds the *first* non-zero return code issued by IKJCT441 when processing the individual requests. Further return codes from individual requests do not change the content of register 15. If all individual request are performed successfully, register 15 contains a return code of 0.

If the calling program specifies the RETCODE parameter, IKJCT441 also places the return code in the RETCODE parameter. This allows the calling program to take the appropriate actions on return codes from individual requests.

Table 128. Return codes from IKJCT441 (entry code TSVELOC)

Return code dec(Hex)	Meaning
0(0)	IKJCT441 successfully updated the parameters for this variable.
4(4)	The caller should not rescan the variable. It is an I/O variable containing an & and is not a variable name.
8(8)	The variable requires evaluation. IKJCT441 did not update VALUEPTR and VALUELEN. The value of the variable is not relevant.
12(C)	The variable is a label. The value of the variable is meaningless.
20(14)	There are no more variables.
24(18)	The variable is a procedure name. IKJCT441 updated VALUEPTR and VALUELEN, but the value of the variable is meaningless.
40(28)	One of the following situations occurred: • The caller's parameter list contains an error. • The caller of IKJCT441 is not activated via a CLIST or REXX exec. • The caller attempted to access a REXX variable pool while another program or REXX exec was accessing the same variable pool. IKJCT441 did not return values for any of the parameters.
44(2C)	The entry code is not valid. IKJCT441 did not return values for any of the parameters.

Table 128. Return codes from IKJCT441 (entry code TSVELOC) (continued)

Return code dec(Hex)	Meaning
72(48)	The variable returned is a reference variable. IKJCT441 updated VALUEPTR and VALUELEN. VALUEPTR points to the name of the corresponding variable in the calling procedure whose value is referred to by this variable.
88(58)	A variable value is not returned. The caller is running AMODE24, but the variable requested might be in 31-bit addressable storage. The caller should change to AMODE31 before calling IKJCT441 so that any variables returned are addressable by the caller. If you are unable to invoke IKJCT441 in AMODE31, you can force variables to be kept in 24-bit addressable storage by having the user set the user PROFILE to VARSTORAGE(LOW) before running the application that creates the variables you are trying to access.

Examples Using IKJCT441

The following examples show sample applications for variable access routines.

Example 1: Update or Create a Variable Value

Figure 168 on page 421 shows an example of how to invoke IKJCT441 to update a variable value or create that variable if it does not exist.

```
SETS     CSECT
CVTPTR   EQU   16
CVTTVT   EQU   X'9C'
R15      EQU   15
R14      EQU   14
R13      EQU   13
R12      EQU   12
R11      EQU   11
R00      EQU   0
         IKJTSVT

 SETS     CSECT
         STM   R14,R12,12(R13)   SAVE CALLER'S REGISTERS
         BALR  R12,0             ESTABLISH ADDRESSABILITY
         USING *,R12             BASE REGISTER OF EXECUTING PROGRAM
         ST    R13,SAVEAREA+4    CALLER'S SAVEAREA ADDRESS
         LA    R15,SAVEAREA      EXECUTING PROGRAM'S SAVEAREA ADDRESS
         ST    R15,8(,R13)       EXECUTING PROGRAM'S SAVEAREA ADDRESS
         LA    R13,SAVEAREA      EXECUTING PROGRAM'S SAVEAREA ADDRESS
*
```

```
*
         L     R15,CVTPTR              ACCESS THE CVT
         L     R15,CVTTVT(,R15)        ACCESS THE TSVT
         L     R15,TSVTVACC-TSVT(,R15) ACCESS THE VARIABLE ACCESS RTN
*

*        INVOKE THE VARIABLE ACCESS SERVICE
*
         LTR   R15,R15         VERIFY TSVT ADDRESS PRESENT
         BNZ   CALL441         IF PRESENT, CALL IKJCT441
```

```
LINK441  LINK  EP=IKJCT441,                                        *
               PARAM=(ECODE,   ENTRY CODE                          *
               NAMEPTR,        POINTER TO VARIABLE NAME            *
               NAMELEN,        LENGTH OF VARIABLE NAME             *
               VALUEPTR,       POINTER TO VARIABLE VALUE           *
               VALUELEN,       LENGTH OF VARIABLE VALUE            *
               TOKEN),         TOKEN TO VARIABLE ACCESS SERVICE    *
               VL=1            CAUSES HI BIT ON IN THE PARM LIST
         B     RET441

CALL441  CALL  (15),                                               *
               (ECODE,         ENTRY CODE                          *
               NAMEPTR,        POINTER TO VARIABLE NAME            *
               NAMELEN,        LENGTH OF VARIABLE NAME             *
               VALUEPTR,       POINTER TO VARIABLE VALUE           *
               VALUELEN,       LENGTH OF VARIABLE VALUE            *
               TOKEN),         TOKEN TO VARIABLE ACCESS SERVICE    *
               VL              CAUSES HI BIT ON IN THE PARM LIST

*
RET441   LTR   R15,R15         CHECK RETURN CODE
         BNZ   ERRORRTN
*
```

Figure 168. Example 1: Update or Create a Variable Value

Figure 'Example 1: Update or Create a Variable Value' **(Continued)**

```
*
ERRORRTN DS   0H
         L    R13,4(,R13)      CALLER'S SAVEAREA
         L    R14,12(,R13)     RESTORE REGISTER 14
         LM   R00,R12,20(R13)  RESTORE REMAINING REGISTERS
         BR   R14              RETURN TO CALLER, REGISTER 15 CONTAINS
*                              THE RETURN CODE FROM IKJCT441
*
```

```
*
NAME     DC    CL12'VARIABLENAME'  NAME OF THE VARIABLE
NAMELEN  DC    F'12'               LENGTH OF THE VARIABLE NAME
VALUE    DC    CL3'YES'            VARIABLE VALUE
VALUELEN DC    F'3'                LENGTH OF THE VARIABLE VALUE
NAMEPTR  DC    A(NAME)             POINTER TO THE VARIABLE NAME
VALUEPTR DC    A(VALUE)            POINTER TO THE VARIABLE VALUE
TOKEN    DC    F'0'                TOKEN (UNUSED HERE)
ECODE    DC    A(TSVEUPDT)         ENTRY CODE FOR SETTING VALUES
SAVEAREA DS    18F
         END
```

Example 2: Return a Variable Value - Create If Required

Figure 169 on page 423 shows an example of how to invoke IKJCT441 to return the value of a variable. If the variable does not exist, IKJCT441 will create it.

```
LOOK     CSECT
CVTPTR   EQU   16
CVTTVT   EQU   X'9C'
R15      EQU   15
R14      EQU   14
R13      EQU   13
R12      EQU   12
R11      EQU   11
R9       EQU   9
R8       EQU   8
R7       EQU   7
R0       EQU   0
         IKJTSVT
```

```
LOOK     CSECT
         STM   R14,R12,12(R13)  SAVE CALLER'S REGISTERS
         BALR  R12,0            ESTABLISH ADDRESSABILITY
         USING *,R12            BASE REGISTER OF EXECUTING PROGRAM
         ST    R13,SAVEAREA+4   CALLER'S SAVEAREA ADDRESS
         LA    R15,SAVEAREA     EXECUTING PROGRAM'S SAVEAREA ADDRESS
         ST    R15,8(,R13)      EXECUTING PROGRAM'S SAVEAREA ADDRESS
         LA    R13,SAVEAREA     EXECUTING PROGRAM'S SAVEAREA ADDRESS
*
```

```
*
         L     R15,CVTPTR               ESTABLISH
         L     R15,CVTTVT(,R15)         ADDRESSABILITY TO THE
         L     R15,TSVTVACC-TSVT(,R15)  VARIABLE ACCESS ROUTINE
*
*        INVOKE THE VARIABLE ACCESS SERVICE
*
         LTR   R15,R15         VERIFY TSVT ADDRESS PRESENT
         BNZ   CALL441         IF PRESENT, CALL IKJCT441
```

```
LINK441  LINK  EP=IKJCT441,                                    *
               PARAM=(ECODE,   ENTRY CODE                      *
               NAMEPTR,        POINTER TO VARIABLE NAME        *
               NAMELEN,        LENGTH OF VARIABLE NAME         *
               VALUEPTR,       POINTER TO VARIABLE VALUE       *
               VALUELEN,       LENGTH OF VARIABLE VALUE        *
               TOKEN),         TOKEN TO VARIABLE ACCESS SERVICE *
               VL=1            CAUSES HI BIT ON IN THE PARM LIST
         B     RET441
```

```
CALL441  CALL  (15),                                           *
               (ECODE,         ENTRY CODE                      *
               NAMEPTR,        POINTER TO VARIABLE NAME        *
               NAMELEN,        LENGTH OF VARIABLE NAME         *
               VALUEPTR,       POINTER TO VARIABLE VALUE       *
               VALUELEN,       LENGTH OF VARIABLE VALUE        *
               TOKEN),         TOKEN TO VARIABLE ACCESS SERVICE *
               VL              CAUSES HI BIT ON IN THE PARM LIST
*
```

Figure 169. Example 2: Return a Variable Value

Figure 'Example 2: Return a Variable Value' **(Continued)**

```
RET441   LTR   R15,R15
         BNZ   ERRORRTN
         L     R7,VALUELEN
         L     R8,VALUEPTR
         LA    R9,L'VALUE
         CR    R7,R9
         BNE   BAD
         CLC   0(L'VALUE,R8),VALUE
         BNE   BAD
*
```

```
*
BAD      DS   0H
ERRORRTN DS   0H
         L    R13,4(,R13)
         L    R14,12(,R13)      RESTORE REGISTER 14
         LM   R0,R12,20(R13)    RESTORE REMAINING REGISTERS
         BR   R14               RETURN TO CALLER, REGISTER 15 CONTAINS
*                               THE RETURN CODE FROM IKJCT441
*
```

```
*
NAME     DC    CL12'VARIABLENAME'  NAME OF THE VARIABLE
NAMELEN  DC    F'12'               LENGTH OF THE VARIABLE NAME
VALUELEN DS    F                   LENGTH OF VARIABLE VALUE
NAMEPTR  DC    A(NAME)             POINTER TO THE VARIABLE NAME
VALUEPTR DS    A                   POINTER TO THE VARIABLE VALUE
VALUE    DC    CL3'YES'            VARIABLE VALUE
TOKEN    DC    F'0'                TOKEN (UNUSED HERE)
ECODE    DC    A(TSVERETR)         ENTRY CODE FOR RETRIEVE
SAVEAREA DS    18F
         END
```

Example 3: Return Variable Value - Do Not Create

Figure 170 on page 425 shows an example of how to invoke IKJCT441 to return a variable value. If the variable does not exist, IKJCT441 does not create it, but returns to the caller with a return code of X'52'.

```
NOIMPM   CSECT
CVTPTR   EQU   16
CVTTVT   EQU   X'9C'
R00      EQU   0
R07      EQU   7
R08      EQU   8
R09      EQU   9
R11      EQU   11
R12      EQU   12
R13      EQU   13
R14      EQU   14
R15      EQU   15
         IKJTSVT
```

```
NOIMP    CSECT
         STM   R14,R12,12(R13)   SAVE CALLER'S REGISTERS
         BALR  R12,0             ESTABLISH ADDRESSABILITY
         USING *,R12             BASE REGISTER OF EXECUTING PROGRAM
         ST    R13,SAVEAREA+4    CALLER'S SAVEAREA ADDRESS
         LA    R15,SAVEAREA      EXECUTING PROGRAM'S SAVEAREA ADDRESS
         ST    R15,8(,R13)       EXECUTING PROGRAM'S SAVEAREA ADDRESS
         LA    R13,SAVEAREA      EXECUTING PROGRAM'S SAVEAREA ADDRESS
*
```

```
         L     R15,CVTPTR              ACCESS THE CVT
         L     R15,CVTTVT(,R15)        ACCESS THE TSVT
         L     R15,TSVTVACC-TSVT(,R15) ACCESS THE VARIABLE ACCESS RTN
*
*        INVOKE THE VARIABLE ACCESS SERVICE
*
         LTR   R15,R15         VERIFY TSVT ADDRESS PRESENT
         BNZ   CALL441         IF PRESENT, CALL IKJCT441
```

```
LINK441  LINK  EP=IKJCT441,                                      *
               PARAM=(ECODE,   ENTRY CODE                        *
               NAMEPTR,        POINTER TO VARIABLE NAME          *
               NAMELEN,        LENGTH OF VARIABLE NAME           *
               VALUEPTR,       POINTER TO VARIABLE VALUE         *
               VALUELEN,       LENGTH OF VARIABLE VALUE          *
               TOKEN),         TOKEN TO VARIABLE ACCESS SERVICE  *
               VL=1            CAUSES HI BIT ON IN THE PARM LIST
         B     RET441
```

```
CALL441  CALL  (15),                                             *
               (ECODE,         ENTRY CODE                        *
               NAMEPTR,        POINTER TO VARIABLE NAME          *
               NAMELEN,        LENGTH OF VARIABLE NAME           *
               VALUEPTR,       POINTER TO VARIABLE VALUE         *
               VALUELEN,       LENGTH OF VARIABLE VALUE          *
               TOKEN),         TOKEN TO VARIABLE ACCESS SERVICE  *
               VL              CAUSES HI BIT ON IN THE PARM LIST
*
```

```
RET441   LTR   R15,R15         CHECK RETURN CODE
         BNZ   ERRORRTN
         L     R7,VALUELEN
         L     R8,VALUEPTR
         LA    R9,L'VALUE
         CR    R7,R9
         BNE   BAD
         CLC   0(L'VALUE,R8),VALUE
         BNE   BAD
*
```

Figure 170. Example 3: Return Variable Value Only

Figure 'Example 3: Return Variable Value Only' **(Continued)**

```
*
BAD      DS   0H
ERRORRTN DS   0H
         LA   R08,TSVRUNDF     OBTAIN NO IMPLICIT RETURN CODE
         CLR  R15,R08          DETERMINE IF UNDEFINED VARIABLE
         BNZ  EXITCODE         IF NOT, THEN EXIT
*
*        ISSUE ERROR MESSAGES OR TAKE ANY APPROPRIATE ACTION
*
```

```
*
EXITCODE L    R13,4(,R13)      CALLER'S SAVEAREA
         L    R14,12(,R13)     RESTORE REGISTER 14
         LM   R00,R12,20(R13)  RESTORE REMAINING REGISTERS
         BR   R14              RETURN TO CALLER, REGISTER 15 CONTAINS
*                              THE RETURN CODE FROM IKJCT441
*
```

```
*
NAME     DC    CL12'VARIABLENAME'  NAME OF THE VARIABLE
NAMELEN  DC    F'12'               LENGTH OF THE VARIABLE NAME
VALUE    DS    CL3                 VARIABLE VALUE WILL BE RETURNED HERE
VALUELEN DS    F                   LENGTH OF THE VARIABLE VALUE WILL BE
                                   RETURNED HERE
```

```
NAMEPTR  DC    A(NAME)             POINTER TO THE VARIABLE NAME
VALUEPTR DC    A(VALUE)            POINTER TO THE VARIABLE VALUE
TOKEN    DC    F'0'                TOKEN (UNUSED HERE)
ECODE    DC    A(TSVNOIMP)         ENTRY CODE FOR NO IMPLICIT SETTING
*                                  OF VALUES.  IF THE SYMBOLIC VARIABLE
*                                  NAME HAD NOT BEEN PREVIOUSLY DEFINED
*                                  IKJCT441 WILL ISSUE THE RETURN CODE
*                                  of 52 (TSVRUNDF).
SAVEAREA DS    18F
         END
```

Example 4: Return All Active Variables and Their Values

Figure 171 on page 427 shows an example of how to invoke IKJCT441 to find all variables and their values.

```
LOCATE   CSECT
CVTPTR   EQU   16
CVTTVT   EQU   X'9C'
R15      EQU   15
R14      EQU   14
R13      EQU   13
R12      EQU   12
R11      EQU   11
R9       EQU   9
R8       EQU   8
R0       EQU   0
         IKJTSVT
```

```
LOCATE   CSECT
         STM   R14,R12,12(R13)  SAVE CALLER'S REGISTERS
         BALR  R12,0            ESTABLISH ADDRESSABILITY
         USING *,R12            BASE REGISTER OF EXECUTING PROGRAM
         ST    R13,SAVEAREA+4   CALLER'S SAVEAREA ADDRESS
         LA    R15,SAVEAREA     EXECUTING PROGRAM'S SAVEAREA ADDRESS
         ST    R15,8(,R13)      EXECUTING PROGRAM'S SAVEAREA ADDRESS
         LA    R13,SAVEAREA     EXECUTING PROGRAM'S SAVEAREA ADDRESS
*
```

```
*
LOOP     DS    0H
         L     R15,CVTPTR              ESTABLISH
         L     R15,CVTTVT(,R15)        ADDRESSABILITY TO THE
         L     R15,TSVTVACC-TSVT(,R15) VARIABLE ACCESS SERVICE
*
```

```
*        INVOKE THE VARIABLE ACCESS SERVICE
*
         LTR   R15,R15         VERIFY TSVT ADDRESS PRESENT
         BNZ   CALL441         IF PRESENT, CALL IKJCT441
LINK441  LINK  EP=IKJCT441,                                    *
               PARAM=(ECODE,   ENTRY CODE                      *
               NAMEPTR,        POINTER TO VARIABLE NAME        *
               NAMELEN,        LENGTH OF VARIABLE NAME         *
               VALUEPTR,       POINTER TO VARIABLE VALUE       *
               VALUELEN,       LENGTH OF VARIABLE VALUE        *
               TOKEN),         TOKEN TO VARIABLE ACCESS SERVICE *
               VL=1            CAUSES HI BIT ON IN THE PARM LIST
         B     RET441
```

```
CALL441  CALL  (15),                                           *
               (ECODE,         ENTRY CODE                      *
               NAMEPTR,        POINTER TO VARIABLE NAME        *
               NAMELEN,        LENGTH OF VARIABLE NAME         *
               VALUEPTR,       POINTER TO VARIABLE VALUE       *
               VALUELEN,       LENGTH OF VARIABLE VALUE        *
               TOKEN),         TOKEN TO VARIABLE ACCESS SERVICE *
               VL              CAUSES HI BIT ON IN THE PARM LIST
*
```

```
RET441   C     R15,NOMORE
         BE    ENDUP
         LTR   R15,R15
         BNZ   ERRORRTN
*
MAINLINE DS    0H
         L     R8,NAMEPTR
         L     R9,VALUEPTR
*
```

Figure 171. Example 4: Return all Active Variables and their Values

Figure Example 4: Return all Active Variables and their Values **(Continued)**

```
*
*        ISSUE 'PUTLINE' TO WRITE VARIABLE NAME AND VALUE
*                        - OR -
*        SAVE THE NAME AND VALUE IN A TABLE
*
*
         B    LOOP
*
*
ERRORRTN DS   0H
*
```

```
*        ANALYZE RETURN CODE
*
         B MAINLINE
*
ENDUP    DS   0H
         L    R13,4(,R13)
         L    R14,12(,R13)      RESTORE REGISTER 14
         LM   R0,R12,20(R13)    RESTORE REMAINING REGISTERS
         BR   R14               RETURN TO CALLER, REGISTER 15 CONTAINS
*                               THE RETURN CODE FROM IKJCT441
*
```

```
*
NAMELEN  DS    F           LENGTH OF NAME WILL BE RETURNED HERE
VALUELEN DS    F           LENGTH OF VALUE WILL BE RETURNED HERE
NAMEPTR  DS    A           ADDRESS OF NAME WILL BE RETURNED HERE
VALUEPTR DS    A           ADDRESS OF VALUE WILL BE RETURNED HERE
TOKEN    DC    F'0'        TOKEN MUST BE ZERO ON THE FIRST CALL
*                            AND NEVER CHANGED BY THE CALLER
ECODE    DC    A(TSVELOC)  ENTRY CODE FOR THE 'LOCATE' SERVICE
NOMORE   DC    A(TSVRNOM)  RETURN CODE FOR NO MORE NAMES
SAVEAREA DS    18F
         END
```

Example 5: Update or Create a List of Variables

Figure 172 on page 429 shows an example of a program that invokes IKJCT441 to update the value of three variables or create the variables if they do not exist. The program exits with the return code from IKJCT441 in register 15.

```
SETLIST  CSECT
CVTPTR   EQU   16
CVTTVT   EQU   X'9C'
R15      EQU   15
R14      EQU   14
R13      EQU   13
R12      EQU   12
R11      EQU   11
R00      EQU   0
         IKJTSVT

SETLIST  CSECT
         STM    R14,R12,12(R13)   Save caller's registers
         BALR   R12,0             Establish addressability
         USING  *,R12             Base register of executing program
         ST     R13,SAVEAREA+4    Caller's savearea address
         LA     R15,SAVEAREA      Executing program's savearea address
         ST     R15,8(,R13)       Executing program's savearea address
         LA     R13,SAVEAREA      Executing program's savearea address
*
         OI     NXTLIS2@,B'10000000'     Turn on the hi bit to show
*                                       the end of the 2nd element
         OI     NXTLIS3@,B'10000000'     Turn on the hi bit to show
*                                       the end of the 3rd element
*
         L      R15,CVTPTR               Access the CVT
         L      R15,CVTTVT(,R15)         Access the TSVT
         L      R15,TSVTVACC-TSVT(,R15)  Access the Variable Access Rtn
*
*  Invoke the variable access service
*
         LTR    R15,R15          Verify TSVT address present
         BNZ    CALL441          If present, call IKJCT441
LINK441  LINK   EP=IKJCT441,
                PARAM=(ECODE,    Entry code
                NAMEPTR,         Pointer to variable name
                NAMELEN,         Length of variable name
                VALUEPTR,        Pointer to variable value
                VALUELEN,        Length of variable value
                TOKEN,           Token to variable access service
                ECTPARM,         Let variable access get the ECT
                RETCODE,         Function return code
                NEXTLIST),       Next element address
                VL=1             Causes hi bit on in the parm list
         B      RET441

CALL441  CALL   (15),
                (ECODE,          Entry code
                NAMEPTR,         Pointer to variable name
                NAMELEN,         Length of variable name
                VALUEPTR,        Pointer to variable value
                VALUELEN,        Length of variable value
                TOKEN,           Token to variable access service
                ECTPARM,         Let variable access get the ECT
                RETCODE,         Function return code
                NEXTLIST),       Next element address
                VL               Causes hi bit on in the parm list
*

RET441   DS     0H
         L      R13,4(,R13)      Caller's savearea
         L      R14,12(,R13)     Restore register 14
         LM    R00,R12,20(R13)    Restore remaining registers
         BR    R14                Return to caller, register 15 contains
                                  the return code from IKJCT441
*
```

Figure 172. Example 5: Update or Create a List of Variables

Figure of 'Example 5: Update or Create a List of Variables' **(Continued)**

```
*
******   First request    ******
*
PARM1    DS    0F                First request follows:
ECODE    DC    A(TSVEUPDT)         1st request type
NAMEPTR  DC    A(NAME)             Pointer to 1st variable name
NAME     DC    CL4'VAR1'           1st variable name
NAMELEN  DC    F'4'                1st variable length
VALUEPTR DC    A(VALUE)            Pointer to 1st variable value
VALUE    DC    CL3'ONE'            1st variable value
VALUELEN DC    F'3'                1st variable value length
TOKEN    DC    F'0'                Token (unused here)
ECTPARM  DC    X'FFFFFFFF'         Let variable access get the ECT
RETCODE  DC    F'0'                Function return code
NEXTLIST DC    A(PARMS2)           Next element address
*
******   Second request   ******
*
PARMS2   DS    0F                Second request follows:
*                                  Second request address list:
         DC    A(ECODE2)           Address of parm 1
         DC    A(NAMEPTR2)         Address of parm 2
         DC    A(NAMELEN2)         Address of parm 3
         DC    A(VALPTR2)          Address of parm 4
         DC    A(VALLEN2)          Address of parm 5
         DC    A(TOKEN2)           Address of parm 6
         DC    A(ECTPRM2)          Address of parm 7
         DC    A(RETCDE2)          Address of parm 8
NXTLIS2@ DC    A(NEXTLIS2)         Address of parm 9
*                                  Second request data:
ECODE2   DC    A(TSVEUPDT)         2nd request type
NAMEPTR2 DC    A(NAME2)            Pointer to 2nd variable name
NAME2    DC    CL4'VAR2'           2nd variable name
NAMELEN2 DC    F'4'                2nd variable length
VALPTR2  DC    A(VALUE2)           Pointer to 2nd variable value
VALUE2   DC    CL3'TWO'            2nd variable value
VALLEN2  DC    F'3'                2nd variable value length
TOKEN2   DC    F'0'                Token (unused here)
ECTPRM2  DC    X'FFFFFFFF'         Let variable access get the ECT
RETCDE2  DC    F'0'                Function return code
NEXTLIS2 DC    A(PARMS3)           Next element address
*
******   Third request    ******
*
PARMS3   DS    0F                Third request follows:
*                                  Third request address list:
         DC    A(ECODE3)           Address of parm 1
         DC    A(NAMEPTR3)         Address of parm 2
         DC    A(NAMELEN3)         Address of parm 3
         DC    A(VALPTR3)          Address of parm 4
         DC    A(VALLEN3)          Address of parm 5
         DC    A(TOKEN3)           Address of parm 6
         DC    A(ECTPRM3)          Address of parm 7
         DC    A(RETCDE3)          Address of parm 8
NXTLS3@  DC    A(NEXTLIS3)         Address of parm 9
*                                  Third request data:
ECODE3   DC    A(TSVEUPDT)         3rd request type
NAMEPTR3 DC    A(NAME3)            Pointer to 3rd variable name
NAME3    DC    CL4'VAR3'           3rd variable name
NAMELEN3 DC    F'4'                3rd variable length
VALPTR3  DC    A(VALUE3)           Pointer to 3rd variable value
VALUE3   DC    CL5'THREE'          3rd variable value
VALLEN3  DC    F'3'                3rd variable value length
TOKEN3   DC    F'0'                Token (unused here)
ECTPRM3  DC    X'FFFFFFFF'         Let variable access get the ECT
RETCDE3  DC    F'0'                Function return code
NEXTLIS3 DC    F'0'                Next element address
```

Chapter 25. Accessing the Information Center Facility names directory

This chapter describes how to use ICQCAL00 in an application program to access the Information Center Facility names directory.

A valid ISPF environment must exist for an application to be able to invoke ICQCAL00.

TSO/E program ICQCAL00 lets application users search the Information Center Facility's names directory and retrieve information such as phone numbers, user IDs, and addresses for specified names. For information about the names directory itself, see *z/OS TSO/E Administration*.

Operation of ICQCAL00

Search Input

An application that uses ICQCAL00 passes names directory search requests to ICQCAL00 through an ISPF table. Applications must define the table in advance and identify it when invoking ICQCAL00. The application can prompt users for search requests and can create the table from the users' responses.

Figure 173. Using ICQCAL00 to Access the Names Directory

1. The user requests information.
2. The application program, if necessary, prompts the user for more information.
3. The application program places the request into the ISPF table and then invokes ICQCAL00.
4. ICQCAL00 gets the requested information from the names directory and returns it to the application program. The application program then returns the information to the user.

Each row of the table provides the input for one search of the names directory, using the following types of variables:

- Variables in the names directory to be searched for and returned
- Variables that control the scope of the search and the results and messages displayed to the user.

Any variable or combination of variables in the names directory can be searched for, such as last name, first name, department, or user ID. For example, one row of the input table can specify a search for all directory entries with the name John Smith, or for all directory entries in a certain department. The variable QCANVARS, set by the non-display panel ICQSIECA, contains all the variables used in the names directory.

The controlling variables limit the search. For example, they can specify the names directories to be searched (master, private or both), the types of directory entries to be searched for (names, groups, or both) and the maximum number of selections allowed.

Other variables specify panels on which to display the search results to users, and messages to appear on the panels to guide the user in selecting the displayed names.

Search Output

When a single name matches a search request, the row of the input table that contained the request is updated with the requested data from the matching names entry. The application can then use the data as needed, such as displaying it to a user.

If more than one names entry matches the request, ICQCAL00 displays them on a list panel for the user to view or select. If the user selects an entry, ICQCAL00 places it in the table.

ICQCAL00 uses the panel in Figure 174 on page 432 as the default panel for listing matching entries.

```
ICQCAE40               NAMES - LIST OF ENTRIES
COMMAND ===>                                          SCROLL ===> PAGE

To view, V, or select, S, an entry, type the letter to the left of the
selected entry.
To save selections, press END; to cancel, type CANCEL on the COMMAND line.

  TYPE  LAST/GROUP           FIRST/NICKNAME USER ID  DEPT./DESCRIPTION
_ NAME  Hollerith            Herman         CARDS    Census
_ NAME  Einstein             Albert         EMC2     Physics Lab
_ GROUP Security                                     D333
 ********************** BOTTOM OF LIST ****************************
```

Figure 174. Default Panel for Listing Names - Panel ICQCAE40

If the search locates the entry for a group of names, ICQCAL00 can return or display either the name of the group, or each of the names in the group. One of the controlling variables in the input row lets you specify that any groups found be expanded into their individual names.

Applications

Using ICQCAL00, application programs can search the Information Center Facility names directories for specific fields and retrieve those entries that match the search requests. The retrieved entries can then be displayed to users, who can view them or select them for further processing.

Applications can use the retrieved entries for mailing lists, phone directories, and memo addressing programs. In addition, when an application accesses the names directory in WRITE mode (using variable QAAMODE), it can scan the names directory and make changes to entries. This capability allows applications to change individual entries or make global changes to the names directory, such as changing department names or addresses.

Invoking ICQCAL00

Applications invoke ICQCAL00 with the following syntax. The parameter INTABLE is required; the others are optional keyword parameters.

```
ICQCAL00 +
           REMDUPS(Y|N) +
           TBDISP(OPEN|CLOSE) +
           INTABLE(table name) +
           ERRSTOP(Y|N)
```

REMDUPS(Y | N)
specifies whether ICQCAL00 should remove duplicate names found in the search before returning control to the calling application. Duplicates occur when ICQCAL00 finds the same directory ID in both the private and master directories. The default is Y, to remove duplicate names.

ICQCAL00 removes duplicates based on the values of the directory ID (QAAID) and the directory indicator (QAAIND). For duplicates to be removed, the application must enter both of these variables into the input table.

TBDISP(OPEN | CLOSE)
specifies whether to keep the names directory open when control is returned to the calling application. If the application searches the directory successively, leaving the directory open improves

performance. The application is responsible for closing the directories when they receive them open. The default is CLOSE.

INTABLE(*table name*)
specifies the name of the input table that contains the search variables. The application is responsible for creating and maintaining this table. The following section describes the possible variables for the table.

ERRSTOP(Y | N)
specifies whether ICQCAL00 should stop processing requests or continue until all requests have been processed. The default, Y, returns control to the calling application under the following conditions:

- An error occurs, such as a table open error.
- An invocation parameter value is incorrect.
- No match is found for a search request.
- The user types CANCEL on a list of matching entries or presses END without selecting an entry.

Specify N to return control after all the requests have been processed.

Input Table Variables

Each row in the input table may contain any or all of the following QAA variables mentioned. The application must specify the search variables first under the parameter QAAVARS, then list them separately with their contents, as in the following example:

```
QAAVARS(QAALAST,QAAFRST)
QAALAST(Smith)
QAAFRST(John)
```

QAAVARS
specifies the names variables to be searched for. The possible search variables and their contents are the following, as contained in the variable QCANVARS.

Table 129. Search variables and their contents

Variable	Contents
QAALAST	The last name.
QAAFRST	The first name
QAADISP	The display form of the name, which is in the form: *last name, first name, middle initial*.
QAAMIDLE	The middle name.
QAANICK	The nickname.
QAASUFIX	The name suffix, for example Jr., Sr., or III
QAANTITL	The name title, for example, Mr., Mrs., or Ms.
QAANODE	The system node.
QAAUSER	The system user ID.
QAANODE2	The second system node.
QAAUSER2	The second system user ID.
QAADNUM	The department number.
QAADNAME	The department name.
QAAUTYPE	The user type.
QAATITLE	The job title or position.
QAAPHONE	The phone number.
QAAADDR	The first line of the internal address.
QAAADDR2	The second line of the internal address.
QAAXADR1	The first line of the external address.

Table 129. Search variables and their contents (continued)

Variable	Contents
QAAXADR2	The second line of the external address.
QAAXADR3	The third line of the external address.
QAAXADR4	The fourth line of the external address.
QAAID	The directory ID. This string identifies an entry, and must be unique in the directory containing the entry.
QAAIND	The directory indicator. This variable contains: • "#" if the entry is either a master directory entry or a private directory entry that is a modified version of a master directory entry. • "@" if the entry is a private directory entry that is not a modified version of a master directory entry.
QAAPRIV	Private directory. This variable contains: • ">" if the entry is for a private directory. • A blank if the entry is for the master directory.
QAATYPE	The type of entry ("NAME" or "GROUP").

Note: For the variables QAAADDR through QAAXADR4, you must specify values in the same case as they exist in the directory. For example, if you search for *NEW YORK*, you will not match entries of *New York* or *new york* in the directory.

QAAUSE
specifies whether the row should be used for searching. Set this variable to Y if the row is to be searched. When ICQCAL00 processes a row and finds a matching entry in the names directory, it stores the requested data in the row and sets QAAUSE to N, so the row is not searched again. When ICQCAL00 processes a row but finds no matching entry, or when a variable in the row is incorrect, it sets this variable to X. The calling application can then scan for the first row with a QAAUSE value of X before displaying an error message.

QAAGNRIC
specifies that if ICQCAL00 finds no match for the values, it should scan the directory a second time using generic search values. In that case, any names directory entries *beginning* with the passed values are displayed on a list for selection or returned as matches. The default is Y, to allow a generic search.

QAALIST
specifies whether ICQCAL00 should display a single match for the user to view and select. Y specifies that a single match be displayed on the list panel. The default, N, lets ICQCAL00 return a single match to the table row without displaying it on a list.

QAARTYPE
specifies that the search be limited to NAMES or GROUPS, or should include both. The default is BOTH.

QAADIR
specifies that ICQCAL00 should search the master directory (MASTER), the user's private names directory (PRIVATE), or both. The default is BOTH.

If TBDISP is OPEN, ICQCAL00 sets the names of the directories in use in the shared pool variables QAATAB1 (private directory) and QAATAB2 (master directory). The application is responsible for closing these tables later.

QAAMODE
specifies the mode in which the names directory is to be opened. WRITE allows the application to update a matching entry. For WRITE, the value of QAADIR cannot be BOTH. The default is READ, to access the directory in read mode only.

QAAMXSEL
specifies the maximum number of entries to be selected or returned. The number can have up to eight digits. If the user attempts to select more than nn entries from a list, ICQCAL00 displays an error message. An error message is also displayed if a selected group is to be expanded into its individual names (variable QAAEXPGP is set to Y), and the expansion would result in more than nn entries. If this variable is unspecified, or set to null or zero, there is no limit on the number of entries that can be selected or returned. The default is null.

QAAMXMSG
specifies the ID of a message to display when expansion of a group or selection from the list would cause more than QAAMXSEL entries to be returned to the caller. If this variable is unspecified or set to null, ICQCAL00 will display an Information Center Facility message (ICQCA701).

QAAEXPGP
specifies whether a matching group should be expanded. The default, Y, causes ICQCAL00 to return all the names within the group or within included groups to the input table.

QAAFMSG
specifies the ID of a message to display on the list panel the first time it appears. If two or more list requests have been made, this variable allows the application to send a specific message to help the user select the names in each list. For example, in an installation's memo facility, the first list request might be for the names in the TO: section, the next might be for those who receive carbon copies, and the next for the distribution list. ICQCAL00 could display each list with a prompting message to guide the user in selecting the correct names.

QAAPANEL
specifies the name of the panel to be used for displaying the list of names matching the search variables. The default panel, ICQCAE40, is shown in Figure 174 on page 432.

QAAPANGP
specifies the name of the panel to be used for viewing the members in a group that matches the search variables. This panel is used when a user chooses to view a group entry. The default panel, ICQCAE41, is shown as follows.

Note: A panel used instead of the default panel must have similar INIT and PROC sections.

```
ICQCAE41               NAMES - VIEW A GROUP
COMMAND ===>                                                SCROLL ===> PAGE

To view, V, or select, S, an entry, type the letter to the left of the
selected entry.
To save selections, press END; to cancel, type CANCEL on the COMMAND line.

   GROUP NAME.............Security
   DESCRIPTION............
    TYPE  LAST/GROUP NAME     FIRST/NICKNAME USER ID   DEPARTMENT/DESC
    NAME  Goergen             Pavel          GNP       D333
    NAME  Pawlin              Karen          KP        D333
    GROUP Fire                                         D333a
    GROUP Rescue                                       D333b
 ********************** BOTTOM OF LIST ********************************
```

Figure 175. Default Panel for Viewing Groups - Panel ICQCAE41

QAADUPSR
specifies whether to delete duplicates resulting from expansion of a group or selection from the list, before processing the next search request. Deletion ensures that duplicate results are not returned for the name in the current row. The default is Y, to delete duplicates.

ICQCAL00 removes duplicates based on the values of the directory ID (QAAID) and the directory indicator (QAAIND). For duplicates to be removed, the application must enter both of these variables into the input table.

QAASCNIK
specifies that ICQCAL00 search for the value of QAAFIRST among the nicknames in the directory if it is not found among the first names. If you want this nickname search, then do not specify QAANICK in QAAVARS. The default is Y, for searching the nicknames.

QAAMSGID
contains the ID of a message that ICQCAL00 returns if it sets a non-zero return code after processing the row.

QAARETCD
contains the return code that ICQCAL00 sets after processing the row. (Each request row has its own return code in a QAARETCD variable.) The possible return codes are listed in "Return Codes from ICQCAL00" on page 436.

Return Codes from ICQCAL00

ICQCAL00 sets one of the following return codes in the variable &QAARETCD. There is one return code per request from the table.

Table 130. ICQCAL00 return codes

Return code	Meaning	Message ID
0	ICQCAL00 completed successfully.	N/A
4	No names were selected from list.	ICQCA710
8	CANCEL was typed on the list.	ICQCA711
12	The name was not found.	ICQCA712
16	Master/Private directory conflict. The private directory has precedence yet the master directory was selected.	ICQCA720
20	The private group selected consists of all master directory entries. Only private directory entries are available.	ICQCA721
24	Some entries in the group are from the master directory and are not available.	ICQCA722
30	The master directory library was not allocated.	ICQCA713
32	The master directory does not exist.	ICQCA714
34	The master directory is busy.	ICQCA715
36	There was a severe error opening the master directory.	ICQCA716
40	The private directory library was not allocated.	ICQCA717
44	The private directory is busy.	ICQCA718
46	There was a severe error opening the private directory.	ICQCA719
100	INTABLE was not predefined.	N/A
101	REMDUPS is not valid.	N/A
102	TBDISP is not valid.	N/A
103	ERRSTOP is not valid.	N/A
110	No search variables were specified.	N/A
111	There are no rows to be searched.	N/A
112	QAAGNRIC is not valid.	N/A
113	QAALIST is not valid.	N/A
114	QAARTYPE is not valid.	N/A
115	QAADIR is not valid.	N/A

Table 130. ICQCAL00 return codes (continued)

Return code	Meaning	Message ID
116	QAAEXPGP is not valid.	N/A
117	QAADUPSR is not valid.	N/A
118	QAASCNIK is not valid.	N/A
119	QAAMODE is not valid.	N/A
120	QAAMODE/QAADIR combination is not valid.	N/A
121	QAAMXSEL is not valid.	N/A
130	ISPLINK was not found.	N/A
131	Insufficient storage.	N/A
132	Internal error.	N/A
133	Internal error.	N/A
134	QAAVARS contains an incorrect (non-names directory) variable.	N/A
140	A severe error was encountered.	N/A

Example Using ICQCAL00

The PHONE CLIST in Figure 176 on page 438 is a sample application that invokes ISPF dialog management services to display input and output panels. The PHONE CLIST searches the names directory and returns the phone number for a name that the user provides on the input panel. The input panel is shown in Figure 177 on page 440.

The PHONE CLIST creates an input table and loads each phone number request in a row. It then invokes ICQCAL00 to search the names directory and displays the results on an output panel (Figure 178 on page 440). If more than one directory entry matches a request, they are displayed on the list panel shown in Figure 179 on page 441.

```
/*********************************************************************/
/* THIS PROGRAM SEARCHES THE NAMES DIRECTORY FOR PHONE NUMBERS OF     */
/* PERSONS SPECIFIED BY THE USER ON AN INPUT PANEL.  DUPLICATE        */
/* RESULTS ARE RETURNED ON A SELECTION PANEL FOR THE USER TO CHOOSE   */
/* FROM, AND FINAL RESULTS ARE DISPLAYED ON AN OUTPUT PANEL.          */
/*********************************************************************/
PROC 0 TRACE(N)

CONTROL END(ENDO)
CONTROL NOPROMPT NOFLUSH NOMSG
IF &NRSTR(&TRACE) = Y THEN +
  CONTROL LIST CONLIST SYMLIST MSG ASIS
ELSE +
  CONTROL ASIS NOLIST NOCONLIST NOSYMLIST
ISPEXEC CONTROL NONDISPL ENTER
ISPEXEC DISPLAY PANEL(ICQSIECA)   /* DISPLAY DEFAULT NAME VARIABLES
ISPEXEC CONTROL ERRORS RETURN
ISPEXEC TBEND INTABLE             /* END TABLE, IN CASE IT EXISTS
ISPEXEC CONTROL ERRORS CANCEL
/*************************************************************/
/*  CREATE INPUT TABLE OF SEARCH REQUESTS                    */
/*************************************************************/
ISPEXEC TBCREATE INTABLE NAMES (QAAVARS QAAUSE QAAGNRIC +
             QAALIST QAARTYPE QAADIR QAAMODE QAAMXSEL +
             QAAMXMSG QAAEXPGP QAAFMSG QAAPANEL QAAPANGP QAADUPSR +
             QAASCNIK QAAMSGID QAARETCD &QCANVARS) NOWRITE REPLACE
SET ISPCODE = &LASTCC
IF &ISPCODE > 4 THEN +
  DO
    WRITE ERROR IN TBCREATE:  RETURN CODE OF &ISPCODE
    GOTO EXIT
  ENDO
SET NAMES_ENTERED = Y
```

```
DO WHILE &NAMES;_ENTERED = Y
  ISPEXEC TBVCLEAR INTABLE
  ISPEXEC DISPLAY PANEL(JRT1)   /* GET DESIRED INFORMATION
  SET ISPCODE = &LASTCC         /* SAVE RETURN CODE
  IF &ISPCODE ¬= 0 THEN +
    SET NAMES_ENTERED = N
  SET QAAUSE = Y                /* SET UP USE FLAG
  SET QAASCNIK = Y              /* REPEAT WITH FIRST NAME AS NICKNAME
  SET QAAPANEL = &STR(JRT3)     /* WANT LIST TO USE THIS PANEL
  /*********************************************/
  /* USE ALL OTHER DEFAULT VALUES              */
  /*********************************************/
  IF &NRSTR(&QAALAST) ¬= THEN +
    SET QAAVARS = &QAAVARS QAALAST  /* USE LAST NAME IN SEARCH
  IF &NRSTR(&QAAFRST) ¬= THEN +
    SET QAAVARS = &QAAVARS QAAFRST  /* USE FIRST NAME IN SEARCH
  IF &NRSTR(&QAAMIDLE) ¬= THEN +
    SET QAAVARS = &QAAVARS QAAMIDLE /* USE MIDDLE NAME IN SEARCH
  IF &NRSTR(&QAAUSER) ¬= THEN +
    SET QAAVARS = &QAAVARS QAAUSER  /* USE USER ID IN SEARCH
  ISPEXEC TBADD INTABLE           /* ADD ROW TO TABLE
  IF &NAMES;_ENTERED = Y THEN +
    DO
```

Figure 176. A Sample Application Using ICQCAL00 — the PHONE CLIST

Figure of 'A Sample Application Using ICQCAL00 — the PHONE CLIST' (**Continued**)

```
     /*************************************************/
     /* CALL NAMES DIRECTORY PROGRAM ICQCAL00         */
     /* OPTIONS:   REMOVE DUPLICATE ENTRIES           */
     /*            LEAVE DIRECTORY TABLES OPEN        */
     /*************************************************/
      ICQCAL00 REMDUPS(Y) TBDISP(OPEN) INTABLE(INTABLE)
      ISPEXEC TBTOP INTABLE             /* GO TO TOP OF TABLE
      ISPEXEC TBVCLEAR INTABLE
      SET QAAUSE = X                       /* SET FOR ERROR FLAG
      ISPEXEC TBSARG INTABLE
      ISPEXEC TBSCAN INTABLE
      IF &LASTCC = 0 THEN +
        DO                                  /* NAME FOUND
          IF &QAAMSGID ¬= THEN +
             ISPEXEC SETMSG MSG(&QAAMSGID)  /* ID NOT BLANK, USE IT
          ISPEXEC TBDISPL INTABLE PANEL(JRT2) /* DISPLAY RESULTS
        ENDO
      ELSE +
        DO
          ISPEXEC TBVCLEAR INTABLE
          SET QAAUSE = N
          ISPEXEC TBSARG INTABLE
          ISPEXEC TBDISPL INTABLE PANEL(JRT2)
        ENDO
    ENDO
)ATTR DEFAULT(%+_)
      /*  % TYPE(TEXT) INTENS(HIGH)      defaults displayed for     */
      /*  + TYPE(TEXT) INTENS(LOW)       information only           */
      /*  _ TYPE(INPUT) INTENS(HIGH) CAPS(ON) JUST(LEFT)            */
  @ TYPE(INPUT) INTENS(HIGH) PAD(' ') CAPS(OFF) JUST(LEFT)
  ! TYPE(OUTPUT) INTENS(LOW) PAD(' ') CAPS(OFF) JUST(LEFT)
)BODY
+                           PHONE DIRECTORY SEARCH
%COMMAND ===>_ZCMD                                              +
%
+Fill in the name of the person you want information about.
+To continue, press ENTER; to end, press END.
+
+   Last Name         ===>@Z                                     +
+   First Name        ===>@Z                                     +
+   Middle Name       ===>@Z                                     +
+   User Id           ===>@Z       +
)INIT
  .ZVARS = '(QAALAST QAAFRST QAAMIDLE QAAUSER)' /* create input variables */
  .CURSOR = QAALAST
)PROC
)END
```

```
)ATTR DEFAULT(%+_)
/* % TYPE(TEXT) INTENS(HIGH) defaults displayed for */
/* + TYPE(TEXT) INTENS(LOW) information only */
/* _ TYPE(INPUT) INTENS(HIGH) CAPS(ON) JUST(LEFT) */
@ TYPE(INPUT) INTENS(HIGH) PAD(' ') CAPS(OFF) JUST(LEFT)
! TYPE(OUTPUT) INTENS(LOW) PAD(' ') CAPS(OFF) JUST(LEFT)
)BODY
+ PHONE DIRECTORY SEARCH
%COMMAND ===>_ZCMD +
%
+Fill in the name of the person you want information about.
+To continue, press ENTER; to end, press END.
++
Last Name ===>@Z +
+ First Name ===>@Z +
+ Middle Name ===>@Z +
+ User Id ===>@Z +
)INIT
.ZVARS = '(QAALAST QAAFRST QAAMIDLE QAAUSER)' /* create input variables */
.CURSOR = QAALAST
)PROC
)END
```

Figure 177. PHONE CLIST Input Panel Definition (JRT1)

```
)ATTR
 % TYPE(TEXT)   INTENS(HIGH) CAPS(OFF)
 + TYPE(TEXT)   INTENS(LOW)  CAPS(OFF)
 _ TYPE(INPUT)  INTENS(HIGH) CAPS(ON)  JUST(LEFT)
 $ TYPE(INPUT)  INTENS(HIGH) CAPS(OFF) JUST(LEFT)  PAD(_)
 # TYPE(INPUT)  INTENS(HIGH) CAPS(ON)  JUST(LEFT)  PAD(_)
 ! TYPE(OUTPUT) INTENS(HIGH) CAPS(OFF) JUST(LEFT)
 @ TYPE(OUTPUT) INTENS(LOW)  CAPS(OFF) JUST(LEFT)  PAD(' ')

)BODY
+                    PHONE DIRECTORY LIST OF RESULTS
%COMMAND ===>_ZCMD                                        %SCROLL ===>_Z  +
%
+The information you requested is shown below.
+Press ENTER or END to continue.
+
%    TYPE  LAST/GROUP NAME      FIRST NAME     USER ID  PHONE NUMBER
)MODEL ROWS(SCAN)
  @Z@Z    @Z                  @Z             @Z       @Z
)INIT
 .ZVARS = '(ZSCML,QAAPRIV,QAATYPE,QAALAST,+
            QAAFRST,QAAUSER,QAAPHONE)'          /*display variables */
 IF (&ZSCML = ' ')
   &ZSCML = 'PAGE'
)PROC
)END
```

Figure 178. PHONE CLIST Output Panel Definition (JRT2)

```
)ATTR
 % TYPE(TEXT)   INTENS(HIGH) CAPS(OFF)
 + TYPE(TEXT)   INTENS(LOW)  CAPS(OFF)
   TYPE(INPUT)  INTENS(HIGH) CAPS(ON)  JUST(LEFT)
 $ TYPE(INPUT)  INTENS(HIGH) CAPS(OFF) JUST(LEFT)  PAD(_)
 # TYPE(INPUT)  INTENS(HIGH) CAPS(ON)  JUST(LEFT)  PAD(_)
 ! TYPE(OUTPUT) INTENS(HIGH) CAPS(OFF) JUST(LEFT)
 @ TYPE(OUTPUT) INTENS(LOW)  CAPS(OFF) JUST(LEFT)  PAD(' ')

)BODY
+                     PHONE DIRECTORY LIST OF ENTRIES
%COMMAND ===>_ZCMD                                        %SCROLL ===>_Z    +
%
+More than one match was found, to view an entry type V next to it.
+To select one, type S.
+To save selections, press END; to cancel, type CANCEL on the COMMAND line.
+
%    TYPE  LAST/GROUP NAME      FIRST NAME     USER ID  PHONE NUMBER
)MODEL ROWS(SCAN)
#Z@Z@Z    @Z                  @Z             @Z       @Z
)INIT
 .ZVARS = '(ZSCML,QAANSEL,QAAPRIV,QAATYPE,QAALAST,+
            QAAFRST,QAAUSER,QAAPHONE)'
 IF (&ZSCML = ' ')
   &ZSCML = 'PAGE'
)PROC
 &ICQCMD = &ZCMD          /* save command for display in msg ICQGC036  */
 &QAANTSEL = &QAANSEL     /* save selection character for msg ICQCA700 */
 &QAAVCHAR = 'V'
 &QAASCHAR = 'S'
 &ZCMD = TRANS(&ZCMD
       ' ',' '
    CANCEL,CANCEL
      MSG = ICQGC036) /* validate command, blank, or CANCEL only   */
 &QAANSEL = TRANS(&QAANSEL
         &QAAVCHAR;,'V'
         &QAASCHAR;,'S'
       ' ',' '
         MSG = ICQCA700)  /* validate selection char: S, V, or blank   */
 IF (&ZCMD = 'CANCEL')/* if CANCEL is typed                          */
   &QACAN = 'Y'
   VPUT (QACAN) SHARED    /* set CANCEL flag and save it*/
)END
```

Figure 179. PHONE CLIST List Panel Definition (JRT3)

Chapter 26. Using the printer support CLISTs

This chapter describes how to use the printer support CLISTs, ICQCPC00, ICQCPC10 and ICQCPC15, in application programs.

A valid ISPF environment must exist for an application to be able to invoke the printer support CLISTs.

Overview of Using the Printer Support CLISTs

The TSO/E printer support service provides a standard interface between application programs and printers. With printer support, your interactive print applications do not have to be programmed to access specific printers. Instead, applications can invoke printer selection CLIST ICQCPC00 to display lists of printers for users to select. ICQCPC00 can display printers for selection based on their location, print format, and printer type.

Printer support also lets print applications be independent of the print routines that actually print the output. The application or ICQCPC00 can invoke print functions such as CLISTs ICQCPC10 and ICQCPC15 to print a data set on a selected printer.

Figure 180 on page 444 shows the interaction between an application program and the printer support service.

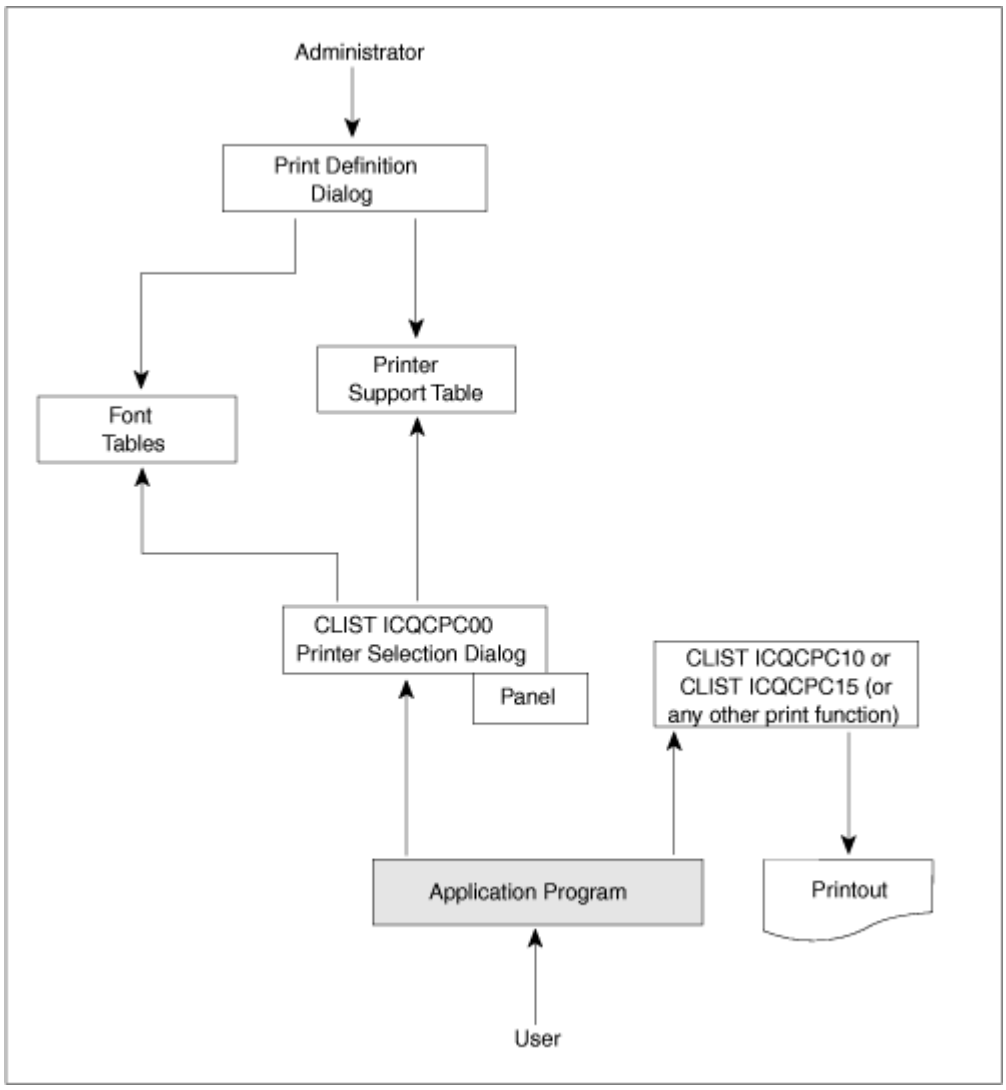


Figure 180. Overview of Printer Support Processing

The following are some examples of the processing that applications can perform using CLISTs ICQCPC00, ICQCPC10 and ICQCPC15. Each example includes an outline of the tasks involved. For more details on how to perform the tasks, read the sections immediately following this overview.

Using the printer support CLISTs, an application can:

1. Prompt the user for a printer location; display a list of printers at that location; when the user selects a printer, display a list of fonts available for the printer; when the user selects fonts, perform printing immediately.

   Tasks involved:

   - Define printers and their characteristics using the print definition dialog of the Information Center Facility. (See *z/OS TSO/E Administration*.)
   - From the application (a text processing program, for example) call CLIST ICQCPC00 with the following:
     - The subset of printers to be displayed (by printer location, format, and/or type). Your application can prompt the user for a location, then invoke ICQCPC00 to display printer types and formats available at that location.
     - The font selection panel to be displayed. (The administrators who define the printers can provide a choice of fonts for each print definition.)

   - The PRINT option. Data will be printed immediately using the print function (CLIST, command, or program) that the administrator may have specified when defining the selected printer. This print function could be ICQCPC10 or ICQCPC15.

2. Allow a user to select a printer and fonts from the displayed list; perform some kind of processing, such as formatting; verify the formatting with the user; then print the data if the user approves it.

   Tasks involved:

   - Define printers as described in *z/OS TSO/E Administration*.
   - Call ICQCPC00 as described in example "1" on page 444 above, leaving out the PRINT option.
   - When the user selects the printer and fonts, the application can do other processing, such as text formatting with the selected fonts, and present the formatting to the user for verification.
   - After the user verifies the formatting, your application can call the print function explicitly. To send the data to the printer selected by the user, invoke ICQCPC10 or ICQCPC15 with the name of the data set to be printed. To override the selected printer:
     - You can send the data to another defined printer by invoking ICQCPC10 or ICQCPC15 with the parameters PLOC and PFORM, which identify the print definition. You can override the printer's defined print characteristics by specifying them on the call to ICQCPC10 or ICQCPC15.
     - You can send data to an undefined printer by invoking ICQCPC10 or ICQCPC15 with the NOTABLE parameter and supplying print characteristics on the CLIST invocation.

3. Save the selected printer so it becomes a default printer; thereafter, instead of displaying a list of printers, direct the data to the default printer.

   Tasks involved:

   - Perform processing as in example "2" on page 445 above. When the user selects a printer, your application can save the information in shared variables in the user profile.
   - Later, when your application is used again, it can call ICQCPC10 or ICQCPC15 to print the data to the default location.
     - To verify that the default print definition still exists, invoke ICQCPC00, specifying the parameters PLOC, PFORM, VERIFY, and PRINT. If the printer or character set is unavailable (has been deleted) your application will receive a return code to that effect.
     - To print without verifying the definition, invoke ICQCPC10 or ICQCPC15 with the parameters PLOC and PFORM only.

Printer Selection CLIST, ICQCPC00

Your print applications can invoke CLIST ICQCPC00 to access printers that are defined in the TSO/E printer support table, ICQAPT10. The print definitions in the table contain variables that associate the printers with optional characteristics, such as print parameters, fonts, and print routines. Information Center Facility administrators create and maintain the definitions using a print definition dialog. See *z/OS TSO/E Administration* for information on creating and maintaining the print definitions.

Applications can invoke ICQCPC00 to display all or a subset of the print definitions to application users. The users can then select a printer from the list. Applications can also specify a single print definition when invoking ICQCPC00, in which case ICQCPC00 can verify the printer's existence and print to it without displaying a selection list.

When ICQCPC00 is used to access a printer, the defined print parameters, fonts, and print function become available for the application to use in printing the output on the printer.

To display a subset of the print definitions for selection, applications can specify certain print definition variables in the call to ICQCPC00. These indexing variables contain the desired printer location, print format, and printer type. For example, a print application could invoke ICQCPC00 to display all the printers that administrators had defined with the locations "Building 1-1" or "Building 1-2", the print format "memo", and the printer type "3800". The user would then be able to choose a printer from a list of those that met these criteria.

The print definition dialog requires that administrators define each printer with a unique combination of location and print format. Thus you can specify a single printer in the call to ICQCPC00 by specifying that printer's location and print format (PLOC and PFORM).

When one or more print definitions meet the search criteria, CLIST ICQCPC00 displays them on a list panel for an application user to select. Figure 181 on page 446 shows a sample of the printer list panel.

```
ICQCPE00      INFORMATION CENTER FACILITY - LIST OF PRINTERS
COMMAND ===>                                          SCROLL ===> PAGE

To select a printer, type S to the left of the selected printer.

  LOCATION         FORMAT   DESCRIPTION
_ Building 1-1     MEMO     3800, hand-delivered output
_ Building 1-2     MEMO     3800
***************************** END OF LIST *********************
```

Figure 181. Printer List Panel

When the user selects a printer, ICQCPC00 can display a font list panel, if the printer has fonts defined for it and the definition allows font selection. Figure 182 on page 446 shows a sample of the font list panel.

```
ICQCPE10               PRINTER - LIST OF FONTS
COMMAND ===>                                          SCROLL ===> PAGE

To select the order of font(s), type the desired number to the left of
the selected font(s).  To restore the pre-defined font order, type R on
the COMMAND line.  When finished, press END; to cancel, type CANCEL on
the COMMAND line.

   LOCATION........... Building 1-1
   FORMAT............. MEMO
   DESCRIPTION........ 3800, hand-delivered output
   SELECTABLE FONTS... 2

   NAME       FONT DESCRIPTION
1_ GOTHIC     Gothic, 12 characters per inch
2_ ITALIC     Italic, 10 characters per inch
__ GOTHIC     Gothic, 10 characters per inch
******************************** END OF LIST **************************
```

Figure 182. Font List Panel

When the application accesses a single printer by uniquely specifying the printer format and location, ICQCPC00 may not display the printer or font list panels. Instead, ICQCPC00 can verify the print definition and any fonts specified.

Whether you specify a single printer in the call to ICQCPC00 or allow user selection, ICQCPC00 lets you:

- Invoke a print function, if the selected printer has a print function defined for it.
- Specify a data set to be printed on the printer, and the number of copies to be printed.

In summary, applications can use ICQCPC00 to:

- Access a printer, either directly or by user selection from a displayed list.
- Let the user select fonts for the printer, from among those named in the print definition.
- Invoke the print function associated with the printer (if a print function is defined).
- Specify fonts to be used in printing.
- Specify the name of a data set to be printed.

- Specify the number of copies to be printed. This overrides any default number of copies that may have been specified in the print definition itself.

Syntax and Parameters

Applications can invoke ICQCPC00 with the following syntax. All the parameters are optional keyword parameters.

```
%ICQCPC00 +
           VERIFY +
           PLOC('loc1 loc2 … locn') +
           PFORM('frm1 frm2 … frmn') +
           PTYPE('typ1 typ2 … typn') +
           OFFLINE +
           CHARS('set1 set2 … setn') +
           CHARSEL +
           CHARSPNL(panelid) +
           PRINT +
           DSNAME(dsname(member)) +
           DDNAME(ddname) +
           COPIES(n) +
```

VERIFY
verifies the existence of a print definition and (optionally) specified fonts. To specify a single printer, use VERIFY with unique values in PLOC and PFORM. To verify that certain fonts are defined for the printer, specify the fonts in the CHARS parameter. With VERIFY, ICQCPC00 does not display the printer for user selection.

PLOC('*loc1 loc2 ... locn*') | PLOC(*loc*) | PLOC((*lo c*))
specifies printer location(s) desired. Use double parentheses around values that contain embedded blanks. Values should match the locations that the administrator defined for the printers. The default value (*) specifies all defined locations. Characters followed by an asterisk specify all locations beginning with those characters. To specify a single print definition, give the printer's location in this parameter and its print format in the PFORM parameter.

PFORM('*frm1 frm2 ... frmn*') | PFORM(*frm*)
specifies print format(s) desired. Values should match the print formats that the administrator defined for the printers. The default value (*) specifies all defined print formats. Characters followed by an asterisk specify all print formats beginning with those characters. To specify a single print definition, give the printer's print format in this parameter and its location in the PLOC parameter.

PTYPE('*typ1 typ2 ... typn*') | PTYPE(*typ*)
specifies the printer type(s) desired. Values should match the printer types that the administrator defined for the printers. The default value (*) specifies all defined printer types. Characters followed by an asterisk specify all printer types beginning with those characters.

Note: PLOC, PFORM, and PTYPE values are combined to determine which print definitions are displayed. Thus, if you specify:

```
%ICQCPC00 PLOC('93* 94*') PFORM('MEMO FOIL') PTYPE(3800)
```

ICQCPC00 displays all print definitions that have the following: locations beginning with 93 or 94, *and* print formats of MEMO or FOIL, *and* a printer type of 3800.

OFFLINE
specifies that printers listed as offline by the administrator be included for display to the user. The administrator sets the printer to offline by typing N in the ONLINE field on panel ICQAPE30 of the print definition dialog.

CHARS('*set1 set2 ... setn*') | CHARS(*set*)
specifies character sets (fonts) to be used with the print function. Use this parameter only when PLOC and PFORM specify a unique printer. Values should match the displayed names or device names of fonts that the administrator defined for the printer. With VERIFY, ICQCPC00 checks that the fonts are defined for the printer. With VERIFY and PRINT, ICQCPC00 invokes the printer with the fonts if they are defined. If PRINT is requested without VERIFY, the CHARS parameter is ignored.

CHARSEL
displays the default font selection panel (ICQCPE10), if the selected print definition includes fonts and allows users to select them. Font selection is allowed when the administrator types Y in the ALLOW FONT SELECTION field of the print definition.

CHARSPNL(*panel ID*)
specifies an alternate font selection panel to display. If CHARSPNL is specified, CHARSEL is assumed and need not be specified.

PRINT
specifies that if the selected print definition includes a print function, the print function be invoked immediately. The administrator can specify a print function on the PRINT FUNCTION panel of the print definition.

DSNAME(*dsname*(*member*))
specifies the data set or member to be printed. This operand is passed to the calling application or to the print function that is invoked when PRINT is specified.

To specify a fully-qualified data set name, enclose it in three sets of single quotes. For example, to print ‘userid.CLIST’, specify:

```
DSNAME('userid.CLIST'')
```

DDNAME(*ddname*)
specifies the ddname associated with the data set to be printed.

COPIES(*n*) | + COPIES(‘,(*n*,*n*,*n*,*n*)’)
specifies the number of printed copies or copy groups (for the 3800). This overrides any number of copies specified in the print definition.

Return Codes from ICQCPC00

For all return codes other than 0, a message ID is stored in the shared pool variable QCPMSGID. The calling application may issue the stored message.

Table 131 on page 448 lists the return codes set by ICQCPC00.

Table 131. Return codes from ICQCPC00

Return code	Meaning
0	Printer and character sets (if specified) were verified and/or selected. If PRINT was requested, printing was successful.
4	A printer was selected, but PRINT was requested and no print function was defined for the printer.
8	A list of printers was displayed to the user, but none was selected.
12	The data set was unavailable (name not valid or access not allowed).
16	No print definitions match the search criteria. The name of the first parameter (PLOC, PFORM, or PTYPE, in that order) not found is in the shared pool variable QCPFARG1. Variable QCPFARG2 contains the value of the parameter that was not found. If a print definition matched the criteria but was inaccessible because OFFLINE was not specified in the call to ICQCPC00, QCPFARG1 and QCPFARG2 are null.
20	A VERIFY request found that a character set specified in the CHARS parameter was not defined to the specified printer. The undefined character set(s) are identified in the shared pool variable QCPBCHAR.
24	The print function set a non-zero return code. The name of the print function is in shared pool variable QCPPRF and the return code is in variable QCPPRC.

Table 131. Return codes from ICQCPC00 (continued)	
Return code	**Meaning**
28	The printer selection panel ICQCPE00 could not be displayed.
32	The fonts selection panel ICQCPE10 could not be displayed.
36	There was a parameter syntax error. The call to ICQCPC00 contained incorrect or conflicting parameters, such as VERIFY with CHARSEL or VERIFY without unique values in PLOC and PFORM.

Variables

When a user selects a print definition, ICQCPC00 makes the data in the definition available to the calling application or print function to use in printing the output. This print definition data is stored in variables prefixed with QAP. These variables are available in the ISPF shared pool and in a temporary table in virtual storage, which ICQCPC00 creates when a printer is selected. The name of the temporary table is in the shared pool variable QCPPRINT. If the print definition has a font table associated with it, ICQCPC00 copies the data in the font table to another temporary table, whose name is in variable QCPFONT.

If ICQCPC00 sets a return code greater than 4, it sets the print definition variables to nulls in the shared pool, and the temporary printer and font tables do not exist in the address space.

Retrieving the Variable Data

There are several ways to retrieve the print definition data from the table variables and make it available for the print function or calling application to use.

- If you use ICQCPC10 or ICQCPC15 as the print function, it retrieves variables directly from the print definition and uses them in printing the output. The variables that ICQCPC10 uses are those that contain parameters of the TSO/E ALLOCATE command; the variables that ICQCPC15 uses are those that contain parameters of the TSO/E PRINTDS command.
- If the print function is not ICQCPC10 or ICQCPC15, your application program must specify any variables that the print function is to retrieve from the definition. The administrator can specify the variables as parameters of the function in the print definition. Figure 183 on page 449 shows where the administrator can specify variables. The variables shown, &QAPTSYSO and &QAPDFCB, contain the parameters created from the SYSOUT CLASS and FCB fields of the print definition.

```
ICQAPE80                        Print Function
COMMAND ===>                                                SCROLL ===> PAGE

   Printer Location .....NJ/324
   Print Format .........REPORT
   Printer Type .........6670
   Description ..........Central Computer site

Indicate whether the Print Function uses the PRINTDS command.
To continue press ENTER.  To exit without saving, press END.

   PRINTDS used      ===> _        If Y, ICQCPC15 can be entered as CLIST Name.
                                   If N, ICQCPC10 can be entered as CLIST Name.

Enter or change CLIST, Command or Program name.
   CLIST Name        ===> ________ If invoked by a CLIST
   Command Name      ===> ________ If invoked by a command
   Program Name      ===> ________ If invoked by a program name

   Parameters     ===>  &QAPTSYSO &QAPDFCB;_____________________
_________________________________
_________________________________
   Test              ===> _        (Y/N) Y to test the function
```

Figure 183. Entering Variables as Parameters on the Print Function Panel

- When an application specifies the parameters for a print function to use, it can also use variables from a print definition. Applications can obtain the variables from the ISPF shared pool by using the ISPF VGET

service or can automatically set these variables in their ISPF function pool by using the ISPF TBGET service as in the following CLIST statement:

```
ISPEXEC TBGET &QCPPRINT /* retrieve printer variables
```

For a list of the print definition variables and the corresponding print definition fields from which they are formed, see Table 132 on page 450 and Table 133 on page 462.

Print Definition Variables

Table 132 on page 450 lists the print definition variables that the printer selection CLIST sets in the ISPF shared pool and in the temporary printer table, &QCPPRINT, when a printer is selected. The table lists the variables in the order of their corresponding print definition fields.

Naming Convention for Printer Support Variables

All printer variables begin with the standard prefix, QAP. Variables that contain print parameters for keywords for JCL statements and the TSO/E ALLOCATE and PRINTDS commands are further identified as follows. In those variables, the fourth letter indicates the command or statement to which the parameter belongs.

Fourth letter
Command or statement

T
TSO/E ALLOCATE command

P
TSO/E PRINTDS command

O
OUTPUT JCL statement

D
JCL DD statement.

The remaining letters are the first letters of the parameter itself. For example, in QAPTSYSO, QAP is the standard prefix, T indicates the TSO/E ALLOCATE command, and SYSO indicates the SYSOUT parameter.

Table 132. Printer definition variables - table

Variable name	Panel name and field name or derivation	Format/description of variable contents
QAPTSO	Derived from all the variables in this table that contain TSO/E ALLOCATE values.	Can be used in the invocation of a print command or program on panel ICQAPE80 to reference all the TSO/E ALLOCATE parameters that are stored in this table. Do not use QAPTSO when calling a CLIST; parsing errors can result if any of the table variables contain TSO/E ALLOCATE parameters with multiple values, such as QAPTCOPI with the syntax COPIES(b1(g1,g2,...)), or QAPTCHAR with the syntax CHARS(f1 f2 ...).
QAPDSN	Obtained from the application that invoked the printer selection CLIST, ICQCPC00.	Contains the fully-qualified name of the data set to be printed.

Table 132. Printer definition variables - table (continued)

Variable name	Panel name and field name or derivation	Format/description of variable contents
QAPDDN	Obtained from the application that invoked the printer selection CLIST, ICQCPC00.	Contains the name of the file to be printed.
QAPLOC	ICQAPE30 - LOCATION field	Required. Can contain any characters, including embedded blanks, except for an asterisk, single quote, or parenthesis. Displayed to the user on the printer selection panel, ICQCPE00. Can be used as a subsetting display argument in invocation of ICQCPC00. Must form unique combination with QAPFORM.
QAPFORM	ICQAPE30 - PRINT FORMAT field	Required. Can contain A-Z, 0-9, @, #, and $, with the first character not numeric. Displayed to the user on the printer selection panel ICQCPE00. Can be used as a subsetting display argument in invocation of ICQCPC00. Must form unique combination with QAPLOC.
QAPTYPE	ICQAPE30 - PRINTER TYPE field	Can contain A-Z, 0-9, and a dash (-). Embedded blanks are valid. Can be used as a subsetting display argument in invocation of ICQCPC00.
QAPDESC	ICQAPE30 - DESCRIPTION field	Can contain any combination of characters, including embedded blanks. Displayed to the user on the printer selection panel ICQCPE00.
QAPONLIN	ICQAPE30 - ONLINE field	Required. Can contain Y or N. Used as a display criteria by printer selection CLIST. Printers that contain an N in this field are not displayed to users unless the CLIST was called with OFFLINE specified.
QAPDSTND	ICQAPE30 - SYSTEM NAME field	Can contain A-Z, 0-9, @, #, and $. If QAPDSTID (PRINTER NAME) is specified, this value becomes c1 in the variables QAPDDEST, QAPODEST, and QAPTDEST. Without QAPDSTID, this value cannot be assigned.

Table 132. Printer definition variables - table (continued)

Variable name	Panel name and field name or derivation	Format/description of variable contents
QAPDSTID	ICQAPE30 - PRINTER NAME field	Required if SYSTEM NAME specified. If SYSTEM NAME is specified, this variable can contain A-Z, 0-9. If SYSTEM NAME is null, this variable can contain A-Z, 0-9, @, #, and $. This value always becomes c1 in the variable QAPTDEST. If there is no value in QAPDSTND, this value becomes c1 in the variables QAPDDEST, QAPODEST, and QAPTDEST. If there is a value in QAPDSTND, this value becomes c2 in the variables QAPDDEST, QAPODEST, and QAPTDEST.
QAPTDEST	Derived from QAPDSTID	DEST parameter on TSO/E ALLOCATE command. **Format:** DEST(c2)
QAPDDEST	Derived from QAPDSTND and QAPDSTID	DEST parameter on JCL DD statement. **Format:** DEST=(c1,c2)
QAPODEST	Derived from QAPDSTND and QAPDSTID	DEST parameter on JCL OUTPUT command. **Format:** DEST=(c1.c2)
QAPOUTDE	ICQAPE50 and ICQAPE53 - OUTPUT DESCRIPTOR field	Can contain A-Z, 0-9, @, #, or $, with the first character non-numeric. Becomes c1 in the variable QAPTOUTD.
QAPTOUTD	Derived from QAPOUTDE	OUTDES parameter on TSO/E ALLOCATE command. **Format:** OUTDES (c1)
QAPOUTC	ICQAPE50 and ICQAPE54 - SYSOUT CLASS field	Can contain A-Z, 0-9, or *. Becomes c1 in the variables QAPTSYSO, QAPDSYSO, and QAPOCLAS.
QAPOUTP	ICQAPE50 - SYSOUT PROGRAM field	Can contain A-Z, 0-9, @, #, and $, with the first character not numeric. Becomes c2 in the variables QAPDSYSO, QAPOWRIT, and QAPTWRIT.
QAPOUTF	ICQAPE50 and ICQAPE53 - SYSOUT FORM field	Can contain A-Z, 0-9, @, #, and $. Becomes c3 in the variable QAPDSYSO. Becomes c1 in the variable QAPTFORM.
QAPTSYSO	Derived from QAPOUTC	SYSOUT parameter on TSO/E ALLOCATE command. **Format:** SYSOUT(c1)

Table 132. Printer definition variables - table (continued)

Variable name	Panel name and field name or derivation	Format/description of variable contents
QAPDSYSO	Derived from QAPOUTC, QAPOUTP, and QAPOUTF	SYSOUT parameter on the JCL DD statement. **Format:** SYSOUT(c1,c2,c3)
QAPTOUTF	Derived from QAPOUTF	FORMS parameter on TSO/E ALLOCATE command. **Format:** FORMS(c1)
QAPOCLAS	Derived from QAPOUTC	CLASS parameter on JCL OUTPUT command. **Format:** CLASS=c1
QAPOWRIT	Derived from QAPOUTP	WRITER parameter on JCL OUTPUT command. **Format:** WRITER=c2
QAPTWRIT	Derived from QAPOUTP	WRITER parameter on TSO/E ALLOCATE command. **Format:** WRITERc2
QAPFORMS	ICQAPE50 - OUTPUT FORMS field	Can contain A-Z, 0-9. Becomes c1 in the variables QAPOFRMS and QAPTFORM.
QAPOFRMS	Derived from QAPFORMS	FORMS parameter on JCL OUTPUT command. **Format:** FORMS=c1
QAPCTRL	ICQAPE50 - DATA CONTROL field	Can contain A or M for ANSI or Machine. Indicates to the print function whether the data contains carriage control characters.
QAPUCS	ICQAPE50 - UCS NAME field	Can contain A-Z, 0-9. Becomes c1 in the variables QAPDUCS and QAPOUCS.
QAPDUCS	Derived from QAPUCS	UCS parameter on JCL DD statement. **Format:** UCS=c1
QAPOUCS	Derived from QAPUCS	UCS parameter on JCL OUTPUT command. **Format:** UCS=c1
QAPTUCS	Derived from QAPUCS	UCS parameter on TSO/E ALLOCATE command. **Format:** UCS(c1)
QAPFCB	ICQAPE50 - FCB NAME field	Can contain A-Z, 0-9, @, #, and $. Becomes c1 in the variables QAPDFCB and QAPOFCB.

Table 132. Printer definition variables - table (continued)

Variable name	Panel name and field name or derivation	Format/description of variable contents
QAPDFCB	Derived from QAPFCB	FCB parameter on JCL DD statement. **Format:** FCB=c1
QAPOFCB	Derived from QAPFCB	FCB parameter on JCL OUTPUT command. **Format:** FCB=c1
QAPHOLD	ICQAPE50 and ICQAPE53 - HOLD field	Can contain Y or N. Used to generate the variables QAPTHOLD and QAPDHOLD.
QAPTHOLD	Derived from QAPHOLD	HOLD or NOHOLD parameter on TSO/E ALLOCATE command. **Format:** HOLD (QAPHOLD = Y) or NOHOLD (QAPHOLD = N)
QAPDHOLD	Derived from QAPHOLD	HOLD parameter on JCL DD statement. **Format:** HOLD=Y (QAPHOLD = Y) or HOLD=N (QAPHOLD = N)
QAPLINEC	ICQAPE50 - LINE COUNT field	Can contain 0 to 255. Becomes n1 in the variable QAPOLINE.
QAPOLINE	Derived from QAPLINEC	LINECT parameter on JCL OUTPUT command. **Format:** LINECT=nt
QAPMODN	ICQAPE51 - MODULE NAME field	Required if TRANSLATE CODE is specified in print definition. May contain A-Z, 0-9, @, #, and $. Becomes c1 in the variables QAPTMODI, QAPDMODI, and QAPOMODI.
QAPMODX	ICQAPE51 - TRANSLATE CODE field	Can contain 0, 1, 2, or 3. Becomes n1 in the variables QAPTMODI, QAPDMODI, and QAPOMODI.
QAPTMODI	Derived from QAPTMODN and QAPTMODX.	MODIFY parameter on TSO/E ALLOCATE command. **Format:** MODIFY(c1,n1)
QAPDMODI	Derived from QAPTMODN and QAPTMODX.	MODIFY parameter on JCL DD statement. **Format:** MODIFY=(c1,n1)

Table 132. Printer definition variables - table (continued)

Variable name	Panel name and field name or derivation	Format/description of variable contents
QAPOMODI	Derived from QAPTMODN and QAPTMODX.	MODIFY parameter on JCL OUTPUT command. **Format:** MODIFY=(c1,n1)
QAPOPTCD	ICQAPE51 - OPTCD J field	Can contain Y or N. Used to generate the variables QAPTOPTC, QAPDOPTC, and QAPOTRC.
QAPTOPTC	Derived from QAPOPTCD	OPTCD parameter on TSO/E ALLOCATE command. **Format:** OPTCD(J)
QAPDOPTC	Derived from QAPOPTCD	OPTCD parameter on JCL DD command. **Format:** OPTCD=(J)
QAPOTRC	Derived from QAPOPTCD	TRC parameter on JCL OUTPUT command. **Format:** TRC=Y
QAPFLN	ICQAPE51 - FLASH NAME field	Can contain A-Z, 0-9, @, #, and $. Becomes c1 in the variables QAPTFLAS, QAPDFLAS, and QAPOFLAS.
QAPFLC	ICQAPE51 - FLASH COUNT field	Can contain 1 to 255. Becomes n1 in the variables QAPTFLAS, QAPDFLAS, and QAPOFLAS.
QAPTFLAS	Derived from QAPFLN and QAPFLC	FLASH parameter on TSO/E ALLOCATE command. **Format:** FLASH(c1,n1)
QAPDFLAS	Derived from QAPFLN and QAPFLC	FLASH parameter on JCL DD statement. **Format:** FLASH=(c1,n1)
QAPOFLAS	Derived from QAPFLN and QAPFLC	FLASH parameter on JCL OUTPUT command. **Format:** FLASH=(c1,n1)
QAPBURST	ICQAPE51 - BURST field	Can contain Y or N. Used to generate the entries: QAPTBURS, QAPDBURS, and QAPOBURS.

Table 132. Printer definition variables - table (continued)

Variable name	Panel name and field name or derivation	Format/description of variable contents
QAPTBURS	Derived from QAPBURST	BURST or NOBURST parameter on TSO/E ALLOCATE command. **Format:** BURST (QAPBURST = Y) or NOBURST (QAPBURST = N)
QAPDBURS	Derived from QAPBURST	BURST parameter on JCL DD statement. **Format:** BURST=Y (QAPBURST = Y) or BURST=N (QAPBURST = N)
QAPOBURS	Derived from QAPBURST	BURST parameter on JCL OUTPUT command. **Format:** BURST=Y (QAPBURST = Y) or BURST=N (QAPBURST = N)
QAPCPYB	ICQAPE52 - TOTAL field (Number of copies)	Required if PER GROUP field has an entry. May contain 1 - 255. Becomes b1 in the variables QAPTCOPI, QAPDCOPI, and QAPOCOPI.
QAPCPYG1 through QAPCPYG8	ICQAPE52 - PER GROUP field (8 subfields)	Subfields must be filled in from left to right. Each subfield can contain numbers 1-255. Becomes g1-g8 in the variables QAPTCOPI, QAPDCOPI, and QAPOCOPI.
QAPTCOPI	Derived from QAPTCPYB and QAPCPYG1 through QAPCPYG8	COPIES parameter on TSO/E ALLOCATE command. **Format:** COPIES (b1,(g1,g2,...,g8))
QAPDCOPI	Derived from QAPTCPYB and QAPCPYG1 through QAPCPYG8	COPIES parameter on JCL DD statement. **Format:** COPIES=(b1,(g1,g2,...,g8))
QAPOCOPI	Derived from QAPTCPYB and QAPCPYG1 through QAPCPYG8	COPIES parameter on JCL OUTPUT command. **Format:** COPIES=(b1,(g1,g2,...,g8))
QAPGROUP	ICQAPE52 - GROUPID NAME field	Can contain A-Z, 0-9. Becomes c1 in the variable QAPOGROU.
QAPOGROU	Derived from QAPGROUP	GROUPID parameter on JCL OUTPUT command. **Format:** GROUPID=c1

Table 132. Printer definition variables - table (continued)

Variable name	Panel name and field name or derivation	Format/description of variable contents
QAPFORMD	ICQAPE52 - FORMDEF NAME field	Can contain A-Z, 0-9, @, #, and $. Becomes c1 in the variable QAPOFORM.
QAPOFORM	Derived from QAPFORMD	FORMDEF parameter on JCL OUTPUT command. **Format:** FORMDEF=c1
QAPPAGED	ICQAPE52 - PAGEDEF NAME field	Can contain A-Z, 0-9, @, #, and $. Becomes c1 in the variable QAPOPAGE.
QAPOPAGE	Derived from QAPPAGED	PAGEDEF parameter on JCL OUTPUT command. **Format:** PAGEDEF=c1
QAPINDEX	ICQAPE52 - INDEX field	Can contain 1 - 31. Becomes n1 in the variable QAPOINDE.
QAPOINDE	Derived from QAPINDEX	INDEX parameter on JCL OUTPUT command. **Format:** INDEX=n1
QAPLINDE	ICQAPE52 - LEFT INDEX field	Can contain 1 - 31. Becomes n1 in the variable QAPOLIND.
QAPOLIND	Derived from QAPLINDE	LINDEX parameter on JCL OUTPUT command. **Format:** LINDEX=n1
QAPINDCF	ICQAPE53 - DCF field	Can contain Y or N. Used to generate QAPPDCF.
QAPPDCF	Derived from QAPINDCF	DCF or NODCF parameter on TSO/E PRINTDS command. **Format:** DCF (QAPINDCF = Y) or NODCF (QAPINDCF = N)
QAPINMEM	ICQAPE53 - MEMBERS field	Can contain Y or N. Used to generate QAPPMEMB. Used to generate QAPPMEMB.
QAPINDIR	ICQAPE53 - DIRECTORY field	Can contain Y or N. Used to generate QAPPMEMB.

Table 132. Printer definition variables - table (continued)

Variable name	Panel name and field name or derivation	Format/description of variable contents
QAPPMEMB	Derived from QAPINMEM and QAPINDIR	MEMBERS, DIRECTORY or ALL parameter on TSO/E PRINTDS command. **Format:** MEMBERS (QAPINMEM = Y, QAPINDIR = N), or DIRECTORY (QAPINMEM = N, QAPINDIR = Y), or ALL (QAPINMEM = Y, QAPINDIR = Y)
QAPINTOD	ICQAPE53 - TO DATASET field	Can contain a fully- or partially-qualified data set name. Becomes t1 in the variable QAPPTODS.
QAPPTODS	Derived from QAPINTOD	TODATASET parameter on TSO/E PRINTDS command. **Format:** TODATASET(t1)
QAPINPLN	ICQAPE54 - PAGE LENGTH field	Can contain 6 - 4095. Becomes n1 in the variable QAPPPLEN.
QAPPPLEN	Derived from QAPINPLN.	PAGELEN parameter on TSO/E PRINTDS command. **Format:** PAGELEN(n1)
QAPINTIT	ICQAPE54 - TITLE field	Can contain Y or N. Used to generate QAPPTITL. Used to generate QAPPTITL.
QAPPTITL	Derived from QAPINTIT	TITLE or NOTITLE parameter on TSO/E PRINTDS command. **Format:** TITLE (QAPINTIT =Y) or NOTITLE (QAPINTIT = N)
QAPINTMR	ICQAPE54 - TOP MARGIN field	Can contain numbers 0 through 6 less than the value of PAGELEN. Becomes n1 in the variable QAPPTMAR.
QAPPTMAR	Derived from QAPINTMR	TMARGIN parameter on TSO/E PRINTDS command. **Format:** TMARGIN(N1)
QAPINBMR	ICQAPE54 - BOTTOM MARGIN field	Can contain numbers 0 through 6 less than the value of PAGELEN. Becomes n1 in the variable QAPPBMAR.
QAPPBMAR	Derived from QAPINBMR	BMARGIM parameter on TSO/E PRINTDS command. **Format:** BMARGIN(N1)
QAPINLMR	ICQAPE54 - LEFT MARGIN field	Can contain 0 - 255. Becomes n1 in the variable QAPPLMAR.
QAPPLMAR	Derived from QAPINLMR	LMARGIN parameter on TSO/E PRINTDS command. **Format:** LMARGIN(N1)

Table 132. Printer definition variables - table (continued)

Variable name	Panel name and field name or derivation	Format/description of variable contents
QAPINMAX	ICQAPE54 - MAXIMUM LENGTH field	A number. Becomes n1 in the variable QAPPFOLD.
QAPINFOL	ICQAPE54 - EXCESS LENGTH field	Can contain FOLD or TRUN. Used to generate the variable QAPPFOLD.
QAPPFOLD	Derived from QAPINMAX and QAPINFOL	FOLD or TRUNCATE parameter on TSO/E PRINTDS command. **Format:** FOLD (n1) (QAOUNFIK = FOLD) TRUNCATE(n1) (QAPINFOL = TRUN)
QAPINSPA	ICQAPE54 - LINE SPACING field	Can contain 1, 2, 3 or C. Used to generate QAPPSPAC.
QAPPSPAC	Derived from QAPINSPA	CCHAR, SINGLE, DOUBLE or TRIPLE parameter on TSO/E PRINTDS command. **Format:** CCHAR (QAPINSPA = C) SINGLE (QAPINSPA = 1) DOUBLE (QAPINSPA = 2) TRIPLE (QAPINSPA = 3)
QAPINLFR	ICQAPE55 - LINES - FROM field (ignore embedded line numbers)	A number. Becomes n1 in the variable QAPPLINE.
QAPINLTO	ICQAPE55 - LINES - TO field (ignore embedded line numbers)	A number. Becomes n2 in the variable QAPPLINE.
QAPINEFR	ICQAPE55 - LINES - FROM field (use embedded line numbers)	A number. Becomes n1 in the variable QAPPLINE.
QAPINETO	ICQAPE55 - LINES - TO field (use embedded line numbers)	A number. Becomes n2 in the variable QAPPLINE.
QAPPLINE	Derived from QAPINLFR and QAPINLTO, or from QAPINEFR and QAPINETO.	LINES parameter on TSO/E PRINTDS command. **Format:** LINES(n1:n2)
QAPINNUM	ICQAPE55 - LOC field	A number. Becomes n1 in the variable QAPPNUMS.
QAPINNLE	ICQAPE55 - LENGTH field	A number less than 8. Becomes n2 in the variable QAPPNUMS.

Table 132. Printer definition variables - table (continued)

Variable name	Panel name and field name or derivation	Format/description of variable contents
QAPPNUMS	Derived from QAPINLFR/ QAPINLTO or QAPINEFR/ QAPINETO, QAPINNUM, and QAPINNLE	NUM, SNUM or NONUM parameter on TSO/E PRINTDS command. **Format:** NUM(n1,n2) (QAPINLFR/QAPINLTO set) NONUM (QAPINEFR/QAPINETO set)
QAPINFC1 - QAPINFCA	ICQAPE56 - FROM COLUMN 1 through FROM COLUMN 10 fields	A number. Becomes s1 through s10 in the variable QAPPCOLS.
QAPINTC1 - QAPINTCA	ICQAPE56 - TO COLUMN 1 through TO COLUMN 10 fields	A number. Becomes e1 through e10 in the variable QAPPCOLS.
QAPPCOLS	Derived from QAPINFC1 - QAPINFCA and QAPINTC1 - QAPINTCA	COLUMNS parameter on TSO/E PRINTDS command. **Format:** COLUMNS(s1:e1, s2:e2,... s10:e10)
QAPPTRC	ICQAPE57 - TRC field	Can contain Y or N. Y indicates OPTCD(J).
QAPATT1 - QAPATT6	ICQAPE60 - ATTRIBUTE 1 through ATTRIBUTE 6 fields	Can contain any character combination including embedded blanks. Not used to generate any MVS system parameters. Can be used to store information required for a local print function or for locally supported system keywords.
QAPATT7 - QAPATT20	Not displayed on any panel	These variables are available but do not appear on any TSO/E panel. A copy of panel ICQAPE60 may be used to externalize more of these variables if your installation needs them.
QAPNFONT	ICQAPE70 - NUMBER OF FONTS field	Can contain numbers 1 - 99. Denotes the maximum number of fonts that the user can select. A blank defaults to the maximum of 99.
QAPFONL	ICQAPE70 - PERMIT SELECTION field	Can contain Y or N. Y indicates that the user should be presented with the font selection panel, ICQCPE10, if CHARSEL was also specified on the call to the printer selection CLIST, ICQCPC00.
QAPCHARD	Derived from the QAPFDEV field of the fonts that were selected by the administrator on panel ICQAPE70.	Contains the internal names of the fonts that were selected by the administrator. It contains as many fonts as were selected. **Format:** font1 font2 font*n* Four of these fonts become f1 - f4 in the variables QAPTCHAR, QAPDCHAR, and QAPOCHAR.

Table 132. Printer definition variables - table (continued)

Variable name	Panel name and field name or derivation	Format/description of variable contents
QAPCHARS	Derived from the QAPFSCR field of the fonts that were selected by the administrator on panel ICQAPE70.	Contains the script names of the fonts selected by the administrator. It contains as many fonts as were selected. **Format:** font1 font2 font*n*
QAPTCHAR	Derived from QAPCHARD. A maximum of 4 fonts are extracted from QAPCHARD to create the character string in this variable.	CHARS parameter on TSO/E ALLOCATE command. **Format:** CHARS(f1 f2 f3 f4)
QAPDCHAR	Derived from QAPCHARD. A maximum of 4 fonts are extracted from QAPCHARD to create the character string in this variable.	CHARS keyword on JCL DD card. **Format:** CHARS=(f1,f2,f3,f4)
QAPOCHAR	Derived from QAPCHARD. A maximum of 4 fonts will be extracted from QAPCHARD to create the character string in this variable.	CHARS keyword on JCL DD statement. **Format:** CHARS=(f1,f2,f3,f4)
QAPINFUN	ICQAPE80 - PRINTDS USED field	Can contain Y or N. Y indicates PRINTDS is used. N indicates PRINTDS is not used.
function names	ICQAPE80 - print function names	Only one of the following three print function name fields can be specified.
QAPCLIST	ICQAPE80 - CLIST NAME field	Can contain A-Z, 0-9, @, #, and $, with the first character non-numeric. Contains the name of the CLIST that you want to print the data associated with this printer. If you specify PRINT on the call to the printer selection CLIST, the named CLIST is automatically invoked when the user selects the printer.
QAPCOMM	ICQAPE80 - COMMAND NAME field	May contain A-Z, 0-9, @, #, and $, with the first character non-numeric. Contains the name of the command that you want to print the data associated with this printer. If you specify PRINT on the call to the printer selection CLIST, the named command is automatically invoked when the user selects the printer.
QAPPGM	ICQAPE80 - PROGRAM NAME field	May contain A-Z, 0-9, @, #, and $, with the first character non-numeric. Contains the name of the program that you want to print the data associated with this printer. If you specify PRINT on the call to the printer selection CLIST, the named program is automatically invoked when the user selects the printer.

Table 132. Printer definition variables - table (continued)		
Variable name	**Panel name and field name or derivation**	**Format/description of variable contents**
QAPPARM	ICQAPE80 - PARAMETERS field	May contain any combination of characters including embedded blanks. Contains the parameter string that you want passed to the print function when a user selects this printer. The print function scans this field once. You may want to pass variables from this table, such as &QAPSYSO or, if the print function is a program or command, &QAPTSO. (Do not use &QAPTSO with a CLIST). The variables &QAPATT1 through &QAPATT20 can also be passed. The characters &,; /, *, -, or + in these variables will be treated as data.
QAPFTABL	Generated dynamically	If a font table is associated with this printer, this variable contains its name. The name syntax is ICQSP*nnn*, where *nnn* is a number from 0 to 999.
QAPITEM	Always contains "1"	Used internally for table searching.
QAPID	Unique identifier	Used for control purposes. Assigned to the table row when it is created and remains unchanged when other contents are modified by the administrator.

Font Definition Variables

The following table describes the variables that are in the font table. The font table is named in variable &QAPFONT from the shared pool.

Table 133. Font definition variables - table		
Variable name	**Panel name and field name or derivation**	**Format/description of variable contents**
QAPFDISP	ICQAPE70 - DISPLAYED NAME field	Can contain A-Z, 0-9, @, #, and $. Displayed to the user on panel ICQCPE10.
QAPFDEV	ICQAPE70 - DEVICE NAME field	Can contain A-Z, 0-9, @, #, and $. Contains the name of a font as known by the device. For example, if the font table is for a 3800 printer, the font "gothic bold 12 pitch" would be GB12. The contents of this variable specify a font in the variables QAPTCHAR, QAPDCHAR, and QAPOCHAR.
QAPFSCR	ICQAPE70 - SCRIPT NAME field	Can contain A-Z, 0-9, @, #, and $. Contains the font name to be used in the CHARS parameter when the SCRIPT/VS module is invoked.
QAPFOTHR	ICQAPE70 - OTHER field	Can contain any combination of characters. Used for local installation requirements.
QAPFDESC	ICQAPE70 - FONT DESCRIPTION field	Can contain any combination of characters, including imbedded blanks. Contains a description of the font's characteristics. Displayed to the user on font selection panel ICQCPE10.

Additional Variables

In addition to the printer and font table variables, ICQCPC00 sets additional variables for a selected printer in the ISPF shared pool. The additional variables are prefixed with QCP. Some contain error information and are identified with the return codes in Table 131 on page 448 and Table 134 on page 466.

Other QCP variables store the contents of a printer's QAPLOC, QAPFORM, and QAPTYPE variables when the printer is selected. The application can then use these variables (QCPLOC, QCPFORM, QCPTYPE) to redisplay a former selection. Another variable, QCPID, contains a unique identifier for the print definition's row in the printer table.

Print CLIST, ICQCPC10

IBM provides CLIST ICQCPC10 to serve as a print function for printer support. CLIST ICQCPC00 can call ICQCPC10 to print a sequential data set or a member of a partitioned data set on a selected printer. Your print applications can also call ICQCPC10 directly to print data on a specified printer.

Functions

Print CLIST ICQCPC10 provides several options for printing data sets on defined printers. The primary function of ICQCPC10 is to print a data set using the ALLOCATE parameters already specified in a print definition. However, you may pass alternative ALLOCATE parameters in the call to ICQCPC10. Any of these ALLOCATE parameters in the call to ICQCPC10 overrides the same parameter from the print definition.

You can specify the name of a data set to be printed. If you do not specify a data set in the call to ICQCPC10, ICQCPC10 takes the data set name from ICQCPC00 or from the calling application. A data set name in ICQCPC10 overrides one specified in ICQCPC00.

The ability to specify ALLOCATE parameters and a data set name for ICQCPC10 allows you to invoke ICQCPC10 in different ways. The different applications are described below.

Applications

You can use ICQCPC10 in the following ways:

- As the print function of a print definition, ICQCPC10 can be automatically invoked when the user selects the definition.
- As part of an application program, ICQCPC10 can:
  - Specify a print definition and print with it when invoked by the application.
  - Specify a SYSOUT CLASS to use a printer independent of any print definition.

ICQCPC10 as the Print Function of a Printer Definition

Using the Information Center Facility print definition dialog, an administrator can define ICQCPC10 as the print function of a print definition. When a user selects a printer with ICQCPC10 defined as its print function, there are two possible results:

- If the printer selection CLIST ICQCPC00 specified PRINT and a data set name, ICQCPC10 automatically prints the data set at the defined printer. It prints using TSO/E ALLOCATE parameters from the temporary table &QCPPRINT that ICQCPC00 creates when the user selects the printer.
- If ICQCPC00 did not specify PRINT, then ICQCPC10 is available for the calling application to invoke. ICQCPC10 can extract variables from the temporary table, and the calling application can specify a data set and any TSO/E ALLOCATE parameters to be used.

ICQCPC10 as Part of the Calling Application

If you do not want users to select a printer, you can have the calling application invoke ICQCPC10 directly. Bypassing the printer selection CLIST, ICQCPC00, the application can do one of the following:

- Invoke ICQCPC10 with the PLOC and PFORM parameters specifying a single defined printer. In that case, ICQCPC10 can obtain ALLOCATE parameters from the print definition.
- Invoke ICQCPC10 using the NOTABLE parameter, without PLOC and PFORM but specifying a SYSOUT(class) that has a printer associated with it.

In either case, use the DSNAME option of ICQCPC10 to identify a data set to be printed on the specified printer. Because the printer is not selected, there is no temporary table of ALLOCATE values to retrieve. Instead, ICQCPC10 uses any ALLOCATE parameters specified at invocation, or else retrieves ALLOCATE parameters directly from the print definition.

Considerations

Print CLIST ICQCPC10 supports only parameters of the TSO/E ALLOCATE command. ICQCPC10 ignores any other parameters that may be specified in a print definition. ICQCPC10 uses the ALLOCATE parameters to create the print data set.

Syntax and Parameters

Applications can invoke ICQCPC10 with the following syntax. All the parameters are optional keyword parameters. These invocation parameters override any conflicting parameters in the print definition table.

```
%ICQCPC10 +
           NOTABLE +
           DSNAME(dsname(member)) +
           PLOC(loc) +
           PFORM(form) +
           BURST/NOBURST +
           CHARS('set1 set2 … setn') +
           COPIES(number) +
           DEST('destination.userid') +
           FCB('image-id[,ALIGN,VERIFY]') +
           FORMS(forms name) +
           FLASH('overlay-name[,count]') +
           HOLD/NOHOLD +
           MODIFY('module-name[,trc]') +
           OPTCD('[J],[U]') +
           OUTDES(name) +
           SYSOUT(class)
           UCS(ucs name) +
           WRITER(external writer name)
```

NOTABLE
specifies that ICQCPC10 will not retrieve ALLOCATE parameters from the temporary table created by printer selection. NOTABLE is required if the calling application invokes ICQCPC10 without printer selection and without PLOC and PFORM coded to specify a print definition. In that case, (NOTABLE required), a printer must be identified by an entry in the SYSOUT parameter.

DSNAME(*dsname*(*member*))
specifies a data set or member to be printed. This overrides any data set named in the temporary table from the printer selection CLIST, ICQCPC00.

To specify a fully-qualified data set name, enclose it in three sets of single quotes. For example, to print ‘userid.CLIST’, specify:

```
DSNAME('userid.CLIST'')
```

PLOC(*loc*) | PLOC((*lo c*))
specifies printer location. Use double parentheses if the location contains embedded blanks. The location must match the LOCATION field of the desired print definition. If you specify PLOC, you must also specify PFORM to identify a unique print definition. NOTABLE is assumed.

PFORM(*form*)
specifies print format or style. Must match the PRINT FORMAT field of the desired print definition. If you specify PFORM, you must also specify PLOC to identify a unique print definition. NOTABLE is assumed.

BURST | NOBURST
specifies whether output from the 3800 printer is to be burst, trimmed, and stacked. Specify BURST or NOBURST.

CHARS('*set1 set2*') | CHARS(*set*)
specifies the character sets (fonts) to be used.

COPIES(*n*) | COPIES('(*n*,*n*,*n*,*n*)')
specifies the number of printed copies or copy groups (for the 3800).

DEST(*destination*) | DEST('*destination*.*userid*')
specifies the destination to which the output is to be sent.

FCB('*image-id*[,ALIGN,VERIFY]')
specifies a forms-control buffer. You can specify the ID of the forms control image and, optionally, specify that the operator align the printer forms and verify that the image displayed on the printer is the correct one.

FLASH('*overlay-name*[,*count*]')
specifies a forms overlay for use on the 3800 printer and the number of copies to be printed with the overlay.

FORMS(*forms name*)
specifies that the output data set should be printed on a special output form.

HOLD | NOHOLD
specifies whether the data set is to be placed on a HOLD queue before printing.

MODIFY('*module-name*[,*trc*]')
specifies a copy modification module for the 3800 printer. The module contains data, such as headings, and information specifying where and on which copies to print them. The Table Reference Character (TRC) specifies character sets for use with the module. TRC corresponds to fonts specified in CHARS, which is required for use of TRC.

OPTCD('[J],[U]')
specifies formatting and data checking options. 'J' applies to the 3800 printer, and specifies that each line of output data begins with a print control character followed by a table reference character for the font required. 'U' applies to the 1403 or 3211 printers with the UCS feature, and the 3800 printer. It permits data checks and allows analysis by an error analysis routine.

OUTDES(*name*) | OUTDES('*name*,*name*,...')
specifies that the output be printed using an output statement or statements named in the user's logon procedure.

SYSOUT(*class*)
specifies the system output data set and class. Required if you specify NOTABLE without PLOC and PFORM.

UCS(*name*)
specifies a universal character set to be used in printing the output.

WRITER(*name*)
specifies a name for use in processing or selecting a SYSOUT data set. The writer name can contain 1 to 8 alphanumeric or special characters #, $, or @.

Return Codes from ICQCPC10

For all return codes other than 0, ICQCPC10 stores a message ID in the shared pool variable QCPMSGID. The calling application may issue the stored message. ICQCPC10 stores error messages from TSO/E ALLOCATE in shared pool variables QCPERR1 through QCPERR8.

Table 134 on page 466 lists the return codes set by ICQCPC10.

Table 134. Return codes from ICQCPC10	
Return code	**Meaning**
0	Printing completed successfully with no error return codes from TSO/E ALLOCATE or IEBGENER.
8	The print definition indicated by PLOC and PFORM does not exist.
12	The data set or member specified in the DSNAME parameter does not exist or could not be allocated to SYSUT1 of IEBGENER. Variables QCPERR1-QCPERR8 contain the output from IEBGENER.
16	The TSO/E ALLOCATE command set a non-zero return code. Check the TSO/E ALLOCATE parameters for correct syntax.
24	Printing using IEBGENER was unsuccessful. The print function return code is in the shared pool variable QCPPRC. See *OS2/VS2 MVS Utilities* for information about IEBGENER.
28	The printer support table could not be opened.
32	NOTABLE was not specified. ICQCPC10 tried to access the temporary table, &QCPPRINT, but did not find it.
36	There was a parameter syntax error. Conflicting parameters were specified, such as BURST and NOBURST, HOLD and NOHOLD, or PLOC without PFORM.

Print CLIST, ICQCPC15

IBM provides CLIST ICQCPC15 to serve as a print function for printer support. CLIST ICQCPC00 can call ICQCPC15 to print one or more sequential or partitioned data sets or members of a partitioned data set on a selected printer. Your print applications can also call ICQCPC15 directly to print data on a specified printer.

Functions

Print CLIST ICQCPC15 provides several options for printing data sets on defined printers. The primary function of ICQCPC15 is to print a data set using the PRINTDS parameters already specified in a print definition. However, you may pass alternative PRINTDS parameters in the call to ICQCPC15. Any of these PRINTDS parameters in the call to ICQCPC15 overrides the same parameter from the print definition.

You can specify a list of the names of data sets to be printed. If you do not specify a data set in the call to ICQCPC15, ICQCPC15 takes the data set name from ICQCPC00 or from the calling application. A data set name in ICQCPC15 overrides one specified in ICQCPC00.

The ability to specify PRINTDS parameters and a list of data set names for ICQCPC15 allows you to invoke ICQCPC15 in different ways. The different applications are described below.

Applications

You can use ICQCPC15 in the following ways:

- As the print function of a print definition, ICQCPC15 can be automatically invoked when the user selects the definition.
- As part of an application program, ICQCPC15 can:
  - Specify a print definition and print with it when invoked by the application.
  - Specify a SYSOUT CLASS to use a printer independent of any print definition.

ICQCPC15 as the Print Function of a Printer Definition

Using the Information Center Facility print definition dialog, an administrator can define ICQCPC15 as the print function of a print definition. When a user selects a printer with ICQCPC15 defined as its print function, there are two possible results:

- If the printer selection CLIST ICQCPC00 specified PRINT and a data set name, ICQCPC15 automatically prints the data set at the defined printer. ICQCPC15 prints using TSO/E PRINTDS parameters from the temporary table &QCPPRINT that ICQCPC00 creates when the user selects the printer.
- If ICQCPC00 did not specify PRINT, then ICQCPC15 is available for the calling application to invoke. ICQCPC15 can extract variables from the temporary table, and the calling application can specify data sets and any TSO/E PRINTDS parameters to be used.

ICQCPC15 as Part of the Calling Application

If you do not want users to select a printer, you can have the calling application invoke ICQCPC15 directly. Bypassing the printer selection CLIST, ICQCPC00, the application can do one of the following:

- Invoke ICQCPC15 with the PLOC and PFORM parameters specifying a single defined printer. In that case, ICQCPC15 can obtain PRINTDS parameters from the print definition.
- Invoke ICQCPC15 using the NOTABLE parameter, without PLOC and PFORM but specifying a SYSOUT(class) that has a printer associated with it.

In either case, use the DSNAME option of ICQCPC15 to identify the data sets to be printed on the specified printer. Because the printer is not selected, there is no temporary table of PRINTDS values to retrieve. Instead, ICQCPC15 uses any PRINTDS parameters specified at invocation, or else retrieves PRINTDS parameters directly from the print definition.

Considerations

Print CLIST ICQCPC15 supports only parameters of the TSO/E PRINTDS command. ICQCPC15 ignores any other parameters that may be specified in a print definition. ICQCPC15 uses the PRINTDS parameters to invoke the TSO/E PRINTDS command, which prints the output.

Syntax and Parameters

Applications can invoke ICQCPC15 with the following syntax. All the parameters are optional keyword parameters. These invocation parameters override any conflicting parameters in the print definition table.

```
%ICQCPC15 +
           NOTABLE +
           DSNAME(dsname(member),dsname(member),…)/
               DDNAME(ddname) +
           PLOC(loc) +
           PFORM(form) +
           BIND(columns)/LMARGIN(columns) +
           BMARGIN(lines) +
           BURST/NOBURST +
           CCHAR/SINGLE/DOUBLE/TRIPLE +
           CHARS('set1 set2 … setn') +
           COLUMNS('start1:end1,start2:end2,…') +
           COPIES(number) +
           DCF/NODCF +
           DEST('destination.userid') +
           FCB('image-id[,ALIGN,VERIFY]') +
           FLASH('overlay-name[,count]') +
           FOLD(width)/TRUNCATE(width) +
           FORMS(forms name) +
           HOLD/NOHOLD +
           LINES(line-num1:line-num2) +
           MEMBERS/DIRECTORY/ALL +
           MODIFY('module-name[,trc]') +
           NUM('loc,len')/SNUM('loc,len')/NONUM +
           OUTDES(name) +
           PAGELEN(lines) +
           SYSOUT(class)/CLASS(class)
           TITLE/NOTITLE +
           TMARGIN(lines) +
```

```
            TODATASET(dsname)/TODSNAME(dsname) +
            TRC/NOTRC +
            UCS(ucs name) +
            WRITER(external writer name)
```

NOTABLE
specifies that ICQCPC15 will not retrieve ALLOCATE parameters from the temporary table created by printer selection. NOTABLE is required if the calling application invokes ICQCPC15 without printer selection and without PLOC and PFORM coded to specify a print definition. In that case, (NOTABLE required), a printer must be identified by an entry in the SYSOUT parameter.

DSNAME(*dsname*(*member*),+ *dsname*(*member*),...)
specifies one or more data sets or members to be printed. DSNAME overrides any data set named in the temporary table from the printer selection CLIST, ICQCPC00. DATASET can be specified as an alias of DSNAME.

To specify a fully-qualified data set name, enclose it in three sets of single quotes. For example, to print 'userid.CLIST', specify:

```
DSNAME('userid.CLIST'')
```

DDNAME(*ddname*)
specifies the file to be printed. Specifying DDNAME causes the data sets in the file concatenation to be printed in the same way as specifying DSNAME followed by a list of the data set names that make up the file. This overrides any data set named in the temporary table from the printer selection CLIST, ICQCPC00. FILE can be specified as an alias of DDNAME.

PLOC(*loc*) | PLOC((*lo c*))
specifies printer location. Use double parentheses if the location contains embedded blanks. The location must match the LOCATION field of the desired print definition. If you specify PLOC, you must also specify PFORM to identify a unique print definition. NOTABLE is assumed.

PFORM(*form*)
specifies print format or style. Must match the PRINT FORMAT field of the desired print definition. If you specify PFORM, you must also specify PLOC to identify a unique print definition. NOTABLE is assumed.

BIND(*columns*) | LMARGIN(*columns*)
specifies the number of columns to the right that the output should be shifted on the paper.

BMARGIN(*lines*)
specifies the number of blank lines to be left at the bottom of each printed page.

BURST | NOBURST
specifies whether output from the 3800 printer is to be burst, trimmed, and stacked. Specify BURST or NOBURST. NOBURST is the default.

CCHAR | SINGLE | DOUBLE | TRIPLE
CCHAR specifies that ANSI or machine code spacing control characters in the data set should be used for inter-record spacing. SINGLE, DOUBLE and TRIPLE specify that all non-blank lines in the data set should be printed with single, double and triple spacing, respectively.

CHARS('*set1 set2*') | CHARS(*set*)
specifies the character sets (fonts) to be used.

COLUMNS('*start1*:*end1*,*start2*:*end2*,...')
specifies the columns of the data set to be printed. Specify the columns to be printed as pairs of numbers in the format "start-column:end column".

COPIES(*n*) | COPIES(',(*n*,*n*,*n*,*n*)')
specifies the number of printed copies or copy groups (for the 3800).

DCF | NODCF
specifies that if the data set being printed has been formatted by DCF, the font information in the first line of the data set should be extracted. This font information is used when the data set is printed. NODCF specifies that font information should not be extracted from the data set. DCF is the default.

DEST(*destination*) | DEST('*destination*.*userid*')
specifies the destination to which the output is to be sent.

FCB('*image-id*[,ALIGN,VERIFY]')
specifies a forms-control buffer. You can specify the ID of the forms control image and, optionally, specify that the operator align the printer forms and verify that the image displayed on the printer is the correct one.

FLASH('*overlay-name*[,*count*]')
specifies a forms overlay for use on the 3800 printer and the number of copies to be printed with the overlay.

FOLD(*width*) | TRUNCATE(*width*)
specifies the maximum length of the printed line. FOLD indicates that lines longer than the maximum length should be wrapped onto the following line. TRUNCATE indicates that lines longer than the maximum length should be truncated to fit on the line.

FORMS(*forms name*)
specifies that the output data set should be printed on a special output form.

HOLD | NOHOLD
specifies whether the data set is to be placed on a HOLD queue before printing. NOHOLD is the default.

LINES(*line-num1*:*line-num2*)
specifies the range of lines to be printed, either in terms of embedded line-number fields (NUM and SNUM parameters) or relative records (NONUM parameter).

MEMBERS | DIRECTORY | ALL
specifies what portion of a partitioned data set is to be printed. MEMBERS indicates that only the data contained in the members of the partitioned data set should be printed. DIRECTORY indicates that only a list of the members should be printed. ALL indicates that the data contained in the members should be printed, followed by a list of the members in the partitioned data set. ALL is the default.

MODIFY('*module-name*[,*trc*]')
specifies a copy modification module for the 3800 printer. The module contains data, such as headings, and information specifying where and on which copies to print them. The Table Reference Character (TRC) specifies character sets for use with the module. TRC corresponds to fonts specified in CHARS, which is required for use of TRC.

NUM('*loc*,*len*')+ /SNUM('*loc*,*len*')/NONUM
specifies whether line numbers should assumed to be embedded.

NUM
Indicates that the data set contains a line-number field that is to be printed. The first value indicates the column location of the beginning of the line-number field. The second value indicates the number of columns that the line-number field occupies.

SNUM
Indicates that the data set contains a line-number field that is *not* to be printed. The first value indicates the column location of the beginning of the line-number field. The second value indicates the number of columns that the line-number field occupies.

NONUM
Indicates that the records should be treated as though there are no embedded line-numbers.

OUTDES(*name*) | OUTDES('*name*,*name*,...')
specifies that the output be printed using an output statement or statements named in the users logon procedure.

PAGELEN(*lines*)
specifies the number of lines in a printed page.

SYSOUT(*class*)/CLASS(*class*)
specifies the system output data set class. Required if you specify NOTABLE without PLOC and PFORM.

TITLE | NOTITLE
TITLE specifies that a title, including the name of the data set being printed and the page number, should appear on every page of printed output. NOTITLE specifies that the title should be suppressed.

TMARGIN(*lines*)
specifies the number of blank lines to be left at the top of every printed page.

TODATASET(*dsname*) | TODSNAME(*dsname*)
specifies the name of the data set into which the formatted data is to be copied. If this operand is specified, a SYSOUT data set is not created. If the indicated data set does not exist, the TSO/E PRINTDS command creates the data set.

To specify a fully-qualified data set name, enclose it in three sets of single quotes. For example, to copy the formatted output into 'userid.OUTPUT', specify:

```
TODSNAME('userid.OUTPUT'')
```

TRC | NOTRC
specifies whether the data records contain TRC codes that identify the font to be used for printing each record.

UCS(*name*)
specifies a universal character set to be used in printing the output.

WRITER(*name*)
specifies a name for use in processing or selecting a SYSOUT data set. The writer name can contain 1 to 8 alphanumeric or special characters #, $, or @.

Return Codes from ICQCPC15

For all return codes other than 0, ICQCPC15 stores a message ID in the shared pool variable QCPMSGID. The calling application may issue the stored message. ICQCPC15 stores error messages from TSO/E PRINTDS in shared pool variables QCPERR1 through QCPERR8.

Table 135 on page 470 lists the return codes set by ICQCPC15.

Table 135. Return codes from ICQCPC15

Return code	Meaning
0	Printing completed successfully with no error return codes from TSO/E PRINTDS.
4	Printing completed with a warning message from TSO/E PRINTDS. Variables QCPERR1-QCPERR8 contain the output from PRINTDS.
8	The print definition indicated by PLOC and PFORM does not exist.
12	The data set or member specified in the DSNAME parameter does not exist.
16	The DSNAME/DDNAME parameter (or an alias) was specified more than once.
24	Printing using PRINTDS was unsuccessful. The print function return code is in the shared pool variable QCPPRC. Variables QCPERR1-QCPERR8 contain the output from PRINTDS.
28	The printer support table could not be opened.
32	NOTABLE was not specified. ICQCPC15 tried to access the temporary table, &QCPPRINT, but did not find it.
36	There was a parameter syntax error. Conflicting parameters were specified, such as BURST and NOBURST, HOLD and NOHOLD, or PLOC without PFORM

Examples Using Printer CLISTs

The following examples illustrate sample applications for using printer CLISTs.

Example 1: The Printer List CLIST

A user or an application program can invoke the application in Figure 184 on page 471 to print a note or data set. The application invokes CLIST ICQCPC00 using input from a series of WRITE and READ statements; this input could also be obtained from an input panel. If the input is a list request, ICQCPC00 displays a list of printers. If the user requests a specific printer, ICQCPC00 verifies that the printer is defined, then sends the data set to it. Printing would be done by the print function defined for the selected printer. The application displays any messages set by ICQCPC00.

```
/*********************************************************************/
/* THIS CLIST PROMPTS THE USER TO NAME A DATA SET TO BE PRINTED     */
/* AND A PRINTER LOCATION AND FORMAT.  THE USER CAN SELECT THE LAST */
/* LOCATION USED OR SPECIFY ANOTHER.  THE USER CAN SELECT FROM A    */
/* LIST OF PRINTERS.  THE PRINTER SELECTION CLIST SENDS THE DATA SET*/
/* TO THE PRINT FUNCTION DEFINED FOR THE SELECTED PRINTER.          */
/*********************************************************************/
PROC 0
CONTROL NOPROMPT NOFLUSH NOMSG END(ENDO)
CONTROL CAPS
WRITENR DATA SET TO BE PRINTED ===>
READ
SET &PRINTDSN = &STR(&SYSDVAL)               /* save data set name    */
IF &SUBSTR(1:1,&NRSTR(&PRINTDSN)) = &STR(') THEN +
 SET PRINTDSN = &STR('&PRINTDSN'') /* insert 2 quotes for clist call */
ISPEXEC VGET (SELLOC SELFRM) PROFILE         /* get previous location */
IF &NRSTR(&SELLOC) ¬=  THEN +
 DO
  WRITE Previously used printer was &NRSTR(&SELLOC) &NRSTR(&SELFRM)
  WRITE To reuse, press ENTER; to use another printer,
  WRITENR enter another location or * for list ===>
  READ &QMPRTLOC
  IF &NRSTR(&QMPRTLOC) =  THEN     /* if old location requested */ +
   DO
    SET QMPRTLOC = &SELLOC         /* set loc = old location    */
    SET QMPRTFRM = &SELFRM         /* set frm = old format      */
   ENDO
  ELSE +
   DO                                /* request another format */
    WRITENR enter another format or * for list ===>
    READ &QMPRTFRM
   ENDO                              /* request another format */
 ENDO
ELSE +
 DO
  WRITENR PRINTER LOCATION OR * FOR LIST ===>
  READ &QMPRTLOC
  WRITENR PRINT FORMAT OR * FOR LIST ===>
  READ &QMPRTFRM
 ENDO
WRITENR NUMBER OF COPIES ===>
READ &QMPRTNO
IF &NRSTR(&QMPRTNO) = THEN +
 SET &QMPRTNO = 1
CONTROL ASIS
```

Figure 184. Example 1: The Printer List CLIST

Figure of 'Example 1: The Printer List CLIST' **(Continued)**

```
/*********************************************************************/
/* If LOCATION and FORMAT have single values, not null or *,         */
/* then verify that the printer is defined and use it.               */
/* (Do not display a list of printers.)                              */
/*********************************************************************/
   SET VERIFY = VERIFY        /* assume a specific printer was selected
   SET N = &LENGTH(&NRSTR(&QMPRTLOC))  /* check LOCATION value */
   IF &NRSTR(&QMPRTLOC) =  OR +
      (&SUBSTR(&N:&N,&NRSTR(&QMPRTLOC)) = &STR(*) THEN +
       SET VERIFY =       /* VERIFY = null; display a list of printers */
   SET N = &LENGTH(&NRSTR(&QMPRTFRM))            /* check FORMAT value */
   IF &NRSTR(&QMPRTFRM) =  OR +
      (&SUBSTR(&N:&N,&NRSTR(&QMPRTFRM)) = &STR(*) THEN +
       SET VERIFY =       /* VERIFY = null; display a list of printers */
    IF &NRSTR(&QMPRTLOC) =  THEN +
     SET QMPRTLOC = &STR(*)
    IF &NRSTR(&QMPRTFRM) = THEN +
     SET &QMPRTFRM = &STR(*)
    %ICQCPC00 PLOC('(&NRSTR(&QMPRTLOC))') +
              PFORM('(&QMPRTFRM)') +
              PRINT +
              DSNAME(&PRINTDSN) +
              COPIES(&QMPRTNO) &VERIFY
    SET RC = &LASTCC
      WRITE RETURN CODE = &RC
    IF &RC = 0 THEN              /* if printing was successful */ +
     DO
      ISPEXEC VGET (QCPLOC QCPFORM) SHARED /* retrieve printer used    */
      SET SELLOC = &NRSTR(&QCPLOC)         /* load variable next time  */
      SET SELFRM = &NRSTR(&QCPFORM)        /* load variable next time  */
      ISPEXEC VPUT (SELLOC SELFRM) PROFILE
     ENDO                                  /* error printing data set  */
    ELSE +
     DO                                    /* error printing data set  */
      ISPEXEC VGET (QCPMSGID) SHARED       /* get message identifier   */
      ISPEXEC GETMSG MSG(&QCPMSGID) LONGMSG(MESSAGE)
      WRITE &MESSAGE
     ENDO                                 /* error printing data set   */
  EXIT CODE(&RC)                          /* return to calling program */
```

Example 2: The Print Function CLIST

The application in Figure 185 on page 473 expands upon the example in Figure 184 on page 471. It performs the same printer selection function but also invokes either ICQCPC10 or ICQCPC15 to give the data set a SYSOUT CLASS of A and send it to the HOLD queue before printing it. The SYSOUT CLASS and HOLD keyword override anything else specified in the print definition. The application displays any messages set by the printer selection and print CLISTs.

```
/*********************************************************************/
/* THIS CLIST PROMPTS THE USER TO NAME A DATA SET TO BE PRINTED      */
/* AND A PRINTER LOCATION AND FORMAT.  THIS CLIST INVOKES ICQCPC00   */
/* TO ALLOW THE USER TO SELECT A PRINTER FROM A LIST OF PRINTERS.    */
/* THIS CLIST THEN INVOKES EITHER PRINT CLIST ICQCPC10 or ICQCPC15   */
/* TO EFFECT PRINTING ON THE SELECTED PRINTER.                       */
/*********************************************************************/
PROC 0
/*
CONTROL NOPROMPT NOFLUSH NOMSG END(ENDO)
/*
CONTROL CAPS
WRITENR DATA SET TO BE PRINTED  ===>
READ
SET &PRINTDSN = &SYSDVAL              /* check if fully qualified; if
                                      /* data set is not fully qualified,
IF &SUBSTR;(1:1,&PRINTDSN); ¬= &STR(') THEN +
 SET &PRINTDSN = &STR;('&SYSPREF..&PRINTDSN'')  /* add prefix */
ELSE +
 SET &PRINTDSN = &STR('&PRINTDSN'')              /* add quotes */
WRITE PRINTER LOCATION OR * FOR LIST (IF LOCATION CONTAINS A BLANK
WRITENR SURROUND IT WITH PARENTHESES) ===>
READ &QMPRTLOC
WRITENR PRINT FORMAT OR * FOR LIST ===>
READ &QMPRTFRM
WRITENR NUMBER OF COPIES ===>
READ &QMPRTNO
/*
CONTROL ASIS
IF &NRSTR(&QMPRTLOC) =  THEN +
 SET QMPRTLOC = &STR(*)             /* If location is null, set to *
IF &NRSTR(&QMPRTFRM) =  THEN +
 SET QMPRTFRM = &STR(*)             /* If format is null, set to *
 SET VERIFY = &STR(VERIFY)          /* verify only if named printer
 SET TEMP = &SYSINDEX(&STR(*),&QMPRTLOC,0)
 IF &TEMP ¬= 0 THEN +
  SET VERIFY =                /* list (not verify) when unnamed printer
 SET TEMP = &SYSINDEX(&STR(*),&QMPRTFRM,0) /* check for an asterisk
 IF &TEMP ¬= 0 THEN +
  SET VERIFY =                /* list (not verify) when unnamed printer
```

Figure 185. The Print Function CLIST

Figure of 'The Print Function CLIST' (**Continued**)

```
/*********************************************************************/
/* Display printer or list of printers                               */
/*********************************************************************/
    %ICQCPC00 PLOC('(&NRSTR(&QMPRTLOC))') +
              PFORM('(&QMPRTFRM)') &VERIFY
    SET RC = &LASTCC
    IF &RC ¬= 0 THEN +
     DO                               /* error selecting a printer
      ISPEXEC VGET (QCPMSGID) SHARED  /* get message identifier
      ISPEXEC SETMSG MSG(&QCPMSGID)   /* display message on next screen
     ENDO                             /* error selecting a printer
    ELSE +
     DO                               /* user has selected a printer
/*********************************************************************/
/* Using the printer data selected by the user, invoke either        */
/* print routine ICQCPC10 or ICQCPC15.  If the user requests that    */
/* PRINTDS be used, ICQCPC15 is invoked.  Otherwise, ICQCPC10 is     */
/* invoked.  In either case, specify SYSOUT CLASS A and HOLD.        */
/*********************************************************************/
      VGET (QAPINFUN) SHARED          /* Obtain print function type
      IF &QAPINFUN = Y THEN +
        %ICQCPC15 DSNAME(&NRSTR(&PRINTDSN)) SYSOUT(A) HOLD /* PRINTDS
      ELSE +
        %ICQCPC10 DSNAME(&NRSTR(&PRINTDSN)) SYSOUT(A) HOLD /*Not PRINTDS
      SET RC = &LASTCC
      IF &RC ¬= 0 THEN +
       DO                                /* error printing data set
        ISPEXEC VGET (QCPMSGID) SHARED   /* get message identifier
        ISPEXEC SETMSG MSG(&QCPMSGID)    /* display message
       ENDO                              /* error printing data set
     ENDO
 EXIT CODE(&RC)
```

Chapter 27. Invoking an Information Center Facility application

This chapter describes how to use the application invocation function, ICQAMLI0, in a program to invoke an application that is defined to the Information Center Facility.

Operation of ICQAMLI0

ICQAMLI0 allows a program to invoke an application that has been defined with the Application Manager dialog to the Information Center Facility. A program that uses ICQAMLI0 specifies, as input operands, the name of the application to be invoked and a list of parameters to be passed to the application. A program can also use ICQAMLI0 to invoke a tutorial that is associated with an application, instead of the application itself.

Output from ICQAMLI0 is the return code from ICQAMLI0 and an ISPF table that contains the return code from the invoked application or tutorial.

Invoking ICQAMLI0

Invoke ICQAMLI0 from your application program using either ISPEXEC SELECT or ISPSTART. To invoke ICQAMLI0 using ISPEXEC SELECT, a valid ISPF environment must exist. Otherwise, you must use ISPSTART. For information on ISPEXEC SELECT and ISPSTART, see *z/OS ISPF User's Guide Vol I* and *z/OS ISPF User's Guide Vol II*.

The syntax of ICQAMLI0 and a description of each operand follows:

```
ISPEXEC SELECT
   CMD(ICQAMLI0

       [APPLNAME(application-name) ]
       [KEYWORD(keyword)           ]

        [INIT(init-command)]

        [NEXTOPT(option)]

        [TUTORIAL]

        [NOERRPAN]

        [PARM(parameters)] )

   NEWAPPL(application-id)

   PASSLIB
```

APPLNAME
specifies the name of the application to be invoked. The maximum length of the name is 12 characters. The first character must be alphabetic and the remaining characters must be alphanumeric or the special characters $, #, @.

KEYWORD
specifies the keyword that identifies the application to be invoked. If more than one application matches the specified keyword, ICQAMLI0 displays a selection panel to the user, who can then choose the application to be invoked.

INIT
specifies a command string used to invoke an initialization routine. You can use INIT instead of, or in addition to, defining a start-up routine. Use INIT for initialization that is required only under special circumstances, such as the first time an application is invoked. The routine specified by INIT is executed before the start-up routine specified in the application definition.

The command string specified by INIT must be a quoted string with a maximum length of 256 bytes. Its format must be suitable for use with an ISPEXEC SELECT statement.

NEXTOPT
specifies the next option for fast path processing. The maximum length of the string is 80 characters. This string is passed to the dynamic panel display or the application function to support fast path processing. The default is blanks.

TUTORIAL
specifies that the tutorial (if one exists) for the application is to be invoked instead of the application itself.

NOERRPAN
specifies that error messages and panels should not be displayed for error conditions other than a severe error (return code 20). If ICQAMLI0 issues a return code of 20, an error message and an error panel are displayed, regardless of whether NOERRPAN is specified.

The default is to display an error message and panel whenever ICQAMLI0 encounters a error.

PARM
specifies a list of invocation parameters to be passed to the application. The maximum length of this string is 256 bytes. The default is null.

Note: PARM *must* be the last operand specified.

NEWAPPL
specifies a 1- to 4-character code for the application being invoked. The application ID can be one of the following:

- The ISPF application ID assigned by the Information Center Facility administrator to the application being invoked.
- The application ID of the application that is currently in effect. Use the same application ID as the current application because the setting of PF keys and variables common to ISPF and PDF are not copied from one profile pool to another. To retrieve the current application ID, examine the ISPF shared pool variable ZAPPLID.
- ICQ, the application ID for the Information Center Facility.

For information on ISPF application IDs, see *z/OS ISPF User's Guide Vol I* and *z/OS ISPF User's Guide Vol II*.

PASSLIB
specifies that the current set of application-level ISPF libraries, if any exist, are to be used by the application being invoked. You should specify PASSLIB to insure that any library allocated using the ISPF LIBDEF service is available to the application being invoked.

Output Table Variables

ICQAMLI0 returns an ISPF table, ICQAMTRC, that has one row containing the variable QAMRC. QAMRC is the return code from the invoked application or tutorial.

Return Codes from ICQAMLI0

Table 136 on page 476 lists the return codes that ICQAMLI0 passes to your program in register 15. If you invoke ICQAMLI0 from a CLIST, the variable &LASTCC contains the return code from ICQAMLI0.

Table 136. ICQAMLI0 return codes

Return code	Meaning
0	ICQAMLI0 completed successfully.
2	ICQAMLI0 completed successfully. However, no application was invoked because the user did not select an application from the keyword selection panel.

Table 136. ICQAMLI0 return codes (continued)	
Return code	**Meaning**
4	Extended return from successful application.
6	The pre-application installation exit failed.
8	The startup/initialization routine failed.
10	The application or tutorial failed.
12	The application or tutorial was not available because it was locked.
14	The environment for the requested function was not available.
16	The application or tutorial is not defined, or the keyword specified to identify the application is not defined.
20	A severe error occurred. Error panel ICQAME70 is displayed to the user and the corresponding error message shows a reason code. The reason codes are described in Table 137 on page 477.

Reason Codes from ICQAMLI0

ICQAMLI0 sets one of the following reason codes when a return code of 20 is issued. ICQAMLI0 does *not* return the reason code to its caller. Instead, this information is provided to the user of your application program when the error panel ICQAME70 is displayed.

Table 137. ICQAMLI0 reason codes	
Reason code	**Meaning**
100	An error occurred in the Parse Service Routine.
104	Both the APPLNAME and KEYWORD operands were specified. These operands are mutually exclusive.
108	The string specified on the PARM operand is longer than 256 bytes.
110	PQUERY failed.
130	General information needed to invoke the application could not be found. (A DATA row is missing from the table).
134	The application type is not valid. An application must be a panel or a function.
138	The invocation command for this application is not defined.
140	Too many variables are defined to the function and environment, if one exists.
142	The ISPF SELECT command failed.
144	The value of NEXTOPT will not fit in the invocation command.
146	The value of PARM will not fit in the invocation command.
148	Too many data sets are defined for a function library.
150	The LIBDEF service failed while allocating the libraries for the application.
199	A severe error occurred in the KEYWORD search CLIST, ICQAMCL0.

Example Using ICQAMLI0

The CLIST in Figure 186 on page 478 is a sample application that uses ICQAMLI0 to invoke applications that are defined to the Information Center Facility. The CLIST prompts the user to choose the application to be invoked and then uses ICQAMLI0 to invoke the indicated application. This CLIST requires an ISPF environment.

```
 PROC 0
:
 WRITE  ************************************************
 WRITE
 WRITE  Calculations completed.  Please select one of the
 WRITE  following:
 WRITE
 WRITE    1 - Data Base Data Entry
 WRITE    2 - Data Base Reports
 WRITE
 WRITENR  Enter your selection ===>
 READ SELECT
 ISPEXEC VGET (ZAPPLID) SHARED
 IF &SELECT = 1 THEN +
   ISPEXEC SELECT  +
     CMD(ICQAMLI0 APPLNAME(DBENTRY)) +
     NEWAPPL(&ZAPPLID) PASSLIB     /* invoke data base entry +
                                      application
 ELSE +
   IF &SELECT = 2 THEN +
     ISPEXEC SELECT  +
       CMD(ICQAMLI0 APPLNAME(DBREPRT)  +
           PARM(DA(DBREPRT.DAT)))      +
       NEWAPPL(&ZAPPLID) PASSLIB   /* invoke data base reporting    +
                                   application.  Pass data set name +
                                   as parm.
```

Figure 186. A Sample Application Using ICQAMLI0

Chapter 28. Using the GETMSG service

Application programs can use the GETMSG service to retrieve system messages issued during a console session. TSO/E provides the GETMSG service as both a programming service and a REXX function. This chapter describes how to use the GETMSG programming service. See *z/OS TSO/E REXX Reference* for information about the TSO/E REXX GETMSG function.

You can invoke GETMSG from an assembler program or as an assembler service from a program written in a high-level language. For more information about invoking an assembler service from a high-level language, see your language reference.

Functions of GETMSG

There are two types of system messages issued during a console session:

Solicited message
Any message issued in response to an MVS system or subsystem command

Unsolicited message
Any message that is not a direct response to an MVS system or subsystem command (for example, a message sent from another user)

If the user of the CONSOLE command has specified that messages (solicited and/or unsolicited) are not to be displayed, you can use GETMSG to retrieve messages that have been routed to the user's console. You can request to retrieve:

- A specific solicited message (response to a specific command)
- The oldest solicited message
- The oldest unsolicited message
- The oldest message (regardless of whether it is solicited or unsolicited)

There may be times when the message you request has not yet been routed to the user's console. To allow for these situations, you can request that GETMSG wait a specified time for the message to arrive.

Considerations for Using GETMSG

A console session must be active before your program invokes GETMSG. The settings in your console profile must indicate that system messages (solicited and/or unsolicited) are not to be displayed at the terminal. You can use the CONSPROF command to change the settings in the console profile.

To ensure that the proper message is delivered to your application program, use the command and response token (CART), and its mask. The CART is a token that lets you associate MVS system commands and subcommands with their responses. The mask is a search argument that GETMSG uses with the CART parameter for obtaining a message. The CART and mask parameters are valid only if you are retrieving solicited messages.

You cannot selectively retrieve unsolicited messages. Unsolicited messages are not associated with CARTs and are shared by applications.

Multiple Applications

If multiple applications are using GETMSG in a TSO/E user's address space, the following guidelines should be followed to ensure that your application retrieves only the solicited messages destined for it:

- Solicited and unsolicited messages should be explicitly requested. If you request only the oldest message, and the oldest message is solicited, it may not be destined for your application.
- Solicited messages should be requested with a mask that contains X'FF' for at least the first 4 bytes and a CART that begins with a 4-character application identifier.

The user of the CONSOLE command must have also specified a CART that begins with a 4-character application identifier. See *z/OS TSO/E System Programming Command Reference* for specific guidelines about using the CART on the CONSOLE command to associate commands with their responses.

Note: At least one application should allow receiving of unsolicited messages.

Invoking GETMSG

You can invoke GETMSG from an assembler program or as an assembler service from a program written in a high-level language. The method you use to invoke GETMSG depends on the language your program is written in. For more information about invoking GETMSG as an assembler service from a high-level language, see your language reference.

GETMSG must be invoked in the key and execution state of the calling program. GETMSG must be invoked in 31-bit addressing mode. The caller's parameters must be in the primary address space. It can accept input above or below 16 MB in virtual storage.

Figure 187 on page 480 shows the standard parameter list structure for GETMSG.

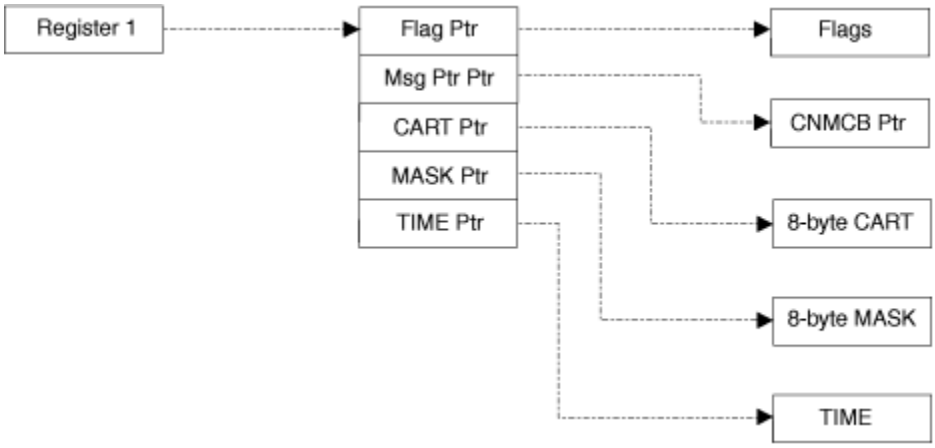


Figure 187. Parameter List Structure for GETMSG Service Parameters

When the GETMSG service is called, the MVS registers contain the following values:

Register 0
Unpredictable

Register 1
The address of the parameter list

Registers 2–12
Unpredictable

Register 13
The problem program save area

Register 14
The return address

Register 15
The entry point address of the programming service

GETMSG Parameters

The GETMSG parameters are described in Table 138 on page 481. The parameters must be specified in the order shown in the figure; however, they do not have to be contiguous in storage.

Table 138. Parameters for GETMSG	
Parameter	**Description**
Flags	See Table 139 on page 481 for a description of the flags. This is a 4-byte field.
CNMCB Address	Specifies a field to contain the address of the control block containing the message and associated information (CNMCB) returned to the user. This is a 4-byte field. See Table 141 on page 482 for details.
CART	Specifies a search argument that is compared with the CARTs propagated with the messages routed to the user's console. If this value is not specified (CART flag is off), the oldest message is returned. This is an 8-byte field.
Mask	Specifies a mask that will be ANDed with the CART specified in the CART parameter and the CART propagated with the messages routed to the user's console. The two ANDed values are then compared, and if a match occurs, the message is returned. If this value is not specified (mask flag is off), the oldest solicited message with a matching CART is returned. This is an 8-byte field.
Time	Specifies a time (in seconds) that the service waits for the message if it has not been routed to the user's console. This is a 4-byte field.

The flags that are passed as input to GETMSG are described in Table 139 on page 481.

Table 139. Flags for GETMSG

Flag	Meaning
X'80000000'	A solicited message that has been routed to the user's console is requested. If this bit is on, the CART and mask flags may be used to request specific messages.
X'40000000'	The oldest unsolicited message routed to the user's console is requested.
X'20000000'	A solicited or unsolicited message is requested. The oldest message routed to the user's console is returned.
X'10000000'	Indicates that a CART is specified in the parameter at offset X'08'.
X'08000000'	Indicates that a mask is specified in the parameter at offset X'10'.
X'04000000'	Indicates that a time value is specified in the parameter at offset X'18'.
X'03FFFFFF'	Reserved.

Output from GETMSG

GETMSG passes a return code to the calling program. See “Return Codes from GETMSG” on page 482 for explanations of the return codes.

If processing was successful and a message was found, GETMSG also returns the address of a chain of console message control blocks (CNMCBs) in the CNMCB parameter. Each CNMCB built by GETMSG contains a pointer to the next CNMCB (if one exists) and a message data block (MDB). There is one CNMCB for each MDB. Each MDB contains one or more lines of message text and related message information, such as the CART and message ID. If the retrieved message is a multi-line message, each line of the message has information pertaining to that line only.

If processing was not successful or a message was not found, GETMSG sets the CNMCB parameter to zero.

Table 140 on page 482 shows the format and contents of the CNMCB. Use the IKJCNMCB macro, which is provided in SYS1.MACLIB, to map the fields of the CNMCB. Use the IEAVM105 macro, which is provided in SYS1.MACLIB, to map the individual fields of the MDB.

Table 140. The console message control block

Offset dec(Hex)	Number of bytes	Field name	Contents or meaning
0(0)	8	CNMCB_ID	CNMCB identifier 'IKJCNMCB'
8(8)	2	CNMCB_VERS	CNMCB version number
10(A)	2	CNMCB_LEN	The length of the CNMCB
12(C)	4	CNMCB_NEXT_MCB	The address of the next CNMCB, if one exists. If this CNMCB is the last in the chain, this field contains nulls.
16(10)	variable	CNMCB_MDB_AREA	The message data block (MDB).

After using the retrieved message, your program must free the CNMCBs associated with the message. The CNMCBs are in key 8 subpool 78 storage.

Return Codes from GETMSG

GETMSG returns one of the following return codes to the calling program. The return code is passed in general register 15.

Table 141. Return codes from GETMSG

Return code (decimal)	Message issued	Description
0	None	Processing successful; a message has been returned.
4	None	Processing successful; the message was not found. If a time value was specified, the timer expired before the message arrived.
8	None	Processing successful; the user pressed the Attention key to end GETMSG.
16	IKJ55324I	Processing unsuccessful; recovery could not be established.
20	IKJ55323I	Processing unsuccessful; a console session is not active.
24	IKJ55325I	Processing unsuccessful; console deactivation is in progress. GETMSG cannot process the request.
28	IKJ55327I	Processing unsuccessful; the CONSOLE command encountered an unrecoverable error (an abend occurred).
32	IKJ55321I	Processing unsuccessful; the input parameter list is not valid.
36	IKJ55322I	Processing unsuccessful; an error occurred while attempting to obtain a message.

Displaying the Retrieved Message

After retrieving the message using GETMSG, your program can decide whether to display the message to the user. Your program may want to display only certain messages to the user, for example, those that require responses. If the message is a WTOR message, the MDB in the CNMCB contains a reply ID.

If your program decides to display the retrieved message, the console command control block (CNCCB) contains message format (MFORM) settings that indicate how the message should be displayed. You can use the IKJCNCCB macro, which is provided in SYS1.MACLIB, to map the fields in the CNCCB.

Example Using GETMSG

Figure 188 on page 483 is an example that shows how an assembler program invokes GETMSG to retrieve the response to a specific system command invocation. The program uses the CART and mask options to ensure that it retrieves only a message destined for it. The CART begins with the identifier of the program, ABCD. The program also specifies a timer value of 5 seconds. If the message requested has not yet been routed to the user's console, GETMSG will wait 5 seconds for it to arrive before returning to the calling program.

```
GETAMSG  CSECT
GETAMSG  AMODE 31
GETAMSG  RMODE ANY
*
* (NOTE: THIS MODULE IS NOT REENTRANT)
* ENTRY LINKAGE
         STM   14,12,12(13)
         LR    12,15
         USING GETAMSG,12
         LA    2,SAVEAREA
         ST    2,8(13)
         ST    13,SAVEAREA+4
         LR    13,2
*
* SET THE PARAMETERS
         SR    2,2
         ST    2,GPL_MSG_PTR     ZERO THE POINTER
*
* INVOKE GETMSG
         LA    1,GPL             POINT TO THE PARAMETER LIST
         LINK  EP=GETMSG         LINK TO GETMSG
*
* PROCESS THE RESULT
*        .
*        .
*        .
*
* EXIT LINKAGE
         L     13,SAVEAREA+4
         L     14,12(13)
         LM    0,12,20(13)
         BR    14
SAVEAREA DS    18F               SAVEAREA
GPL_MSG_PTR DS   A               MESSAGE POINTER
GPL_FLAGS DC   XL4'9C000000'     GETMSG SERVICE REQUEST FLAGS
*                                SOLICITED MESSAGE, CART, MASK, AND
*                                TIME SPECIFIED
GPL_CART DC    CL8'ABCD    '     CART VALUE
GPL_MASK DC    XL8'FFFFFFFF00000000'  MASK VALUE
GPL_TIME DC    F'5'              TIME TO WAIT FOR THE MESSAGE
GPL      EQU   *                 STANDARD PARAMETER LIST
         DC    A(GPL_FLAGS)
         DC    A(GPL_MSG_PTR)
         DC    A(GPL_CART)
         DC    A(GPL_MASK)
         DC    A(GPL_TIME)
         END
```

Figure 188. Invoking GETMSG from an Assembler Language Program

Chapter 29. Using the Unauthorized Resource Processor Service IKJURPS

IKJURPS is a service that allows applications that execute in a TSO/E environment to get control within the TSO/E terminal monitor program (TMP). When the application gets control, it can share and manage its resources.

To share and manage resources, the application:

1. Invokes the IKJURPS service naming a module known as an unauthorized resource processor.
2. Receives control from the TSO/E TMP in the unauthorized resource processorwhere the application shares and manages its resources.

Using the IKJURPS service can be an attractive alternative to other implementations such as:

- Creating a static environment for managing resources before further processing in the application.
- Using authorized services to create an asynchronous exit routine to create and execute an interrupt request block (IRB).

The IKJURPS service is designed for unauthorized callers. By invoking the IKJURPS service and providing an unauthorized resource processor, an application can dynamically create an environment to manage its resources. This is similar to that created by an IRB, and the application remains entirely unauthorized.

Overview of the TSO/E Unauthorized Resource Processor Service

Your unauthorized resource processors, the IKJURPS service, and the IKJEFT01 TMP need to fit together to allow your applications to make use of these services. The following provides an overview of this processing.

The IKJEFT01 TMP is capable of processing both authorized and unauthorized commands and programs. It does this by creating separate control layers from which it invokes authorized or unauthorized commands and subtasks. The following descriptions and Figure 189 on page 486 explain these terms.

The **Authorized Control Layer** invokes authorized commands and programs as well as initiates the unauthorized control layer.

The authorized control layer invokes authorized commands in an authorized environment. From this environment TSO/E commands and programs can use MVS system services that are limited to authorized invokers. This environment is isolated from the unauthorized control layer.

The **Unauthorized Control Layer** invokes unauthorized commands and programs as well as unauthorized resource processors. Note that the life of the unauthorized control layer is the same as the life of the unauthorized command that this layer invokes. For example, the life of the unauthorized control layer is the same as the life of the unauthorized command invoked from READY.

The unauthorized control layer gives control to the unauthorized resource processors that you write. By receiving control in an unauthorized resource processor, an unauthorized application can share and manage its resources within the unauthorized control layer of the TSO/E TMP. This unauthorized resource processor manages the initialization, resource maintenance, and cleanup for the unauthorized resources within the TSO/E TMP.

In the following figure, AA and BB denote unauthorized commands and subtasks initiated from those commands that execute within the TSO/E address space. These applications can invoke the IKJURPS service that in turn give control to the unauthorized resource processor that you name.

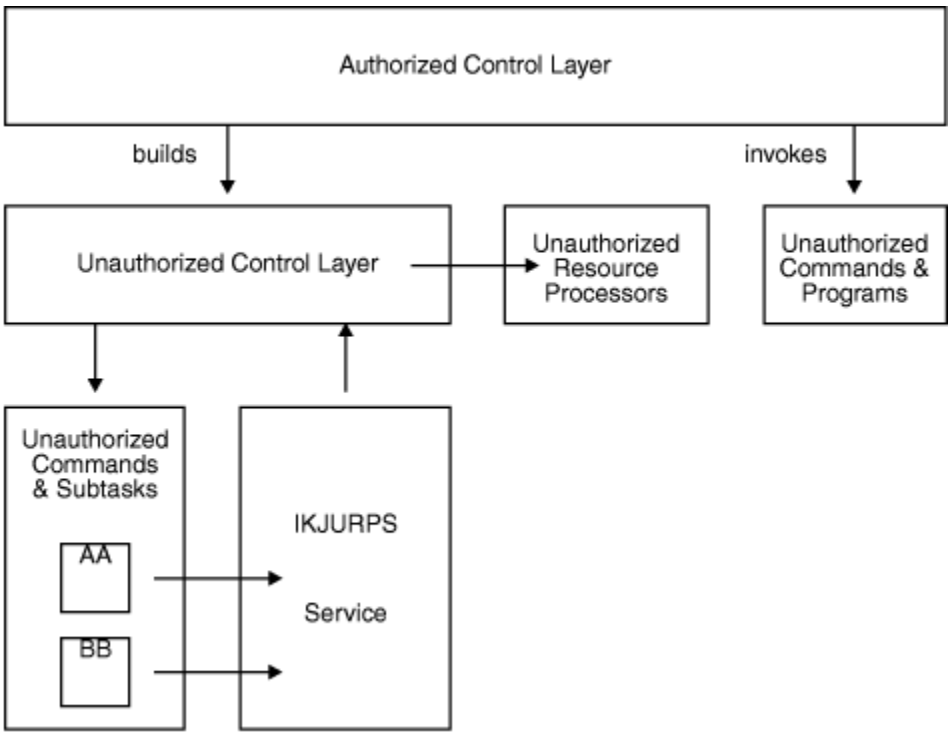


Figure 189. How Unauthorized Resource Processing Fits Into a TSO/E Address Space

Note that the IKJURPS service is an implementation that is specific to the IKJEFT01 TMP, that is, it is available in a foreground TSO/E environment (started though the TSO/E LOGON command) or in a background TSO/E environment (started through EXEC PGM=IKJEFT01). IKJURPS is not available in a non-TSO/E address space.

Application Routine Versus the unauthorized resource processor

As noted above, access to the unauthorized resource processor is through the TSO/E IKJURPS service and requires two main interfaces within this flow of control:

- The application interface to the IKJURPS service
- The unauthorized control layer interface to the unauthorized resource processor.

The following sections provide a detailed view of:

- The interface between the application program and the IKJURPS service
- The interface between the TSO/E TMP unauthorized control layer and the unauthorized resource processors
- How to install unauthorized resource processors.

Passing Control to IKJURPS

Your application needs to pass control to the IKJURPS service. To do so, it:

- Sets up parameters for the IKJURPS service to use
- Invokes the IKJURPS service
- Interprets the return information.

The IKJURPS Parameter List

Use the IKJURPS parameter list to communicate with:

1. The unauthorized control layer; you tell the unauthorized control layer the name of the unauthorized resource processor to invoke.
2. The unauthorized resource processor; you can give the unauthorized resource processor installation-defined data so the unauthorized resource processor can interrogate the data.

Figure 190 on page 488 describes the parameter list passed between the application and the TSO/E IKJURPS service. You must first create the IKJURPS parameter list and place its address into general register 1. Be certain:

1. To turn on the high-order bit in the address of the last parameter to indicate the end of the parameter list
2. All other high-order bits for all other parameters are set off.

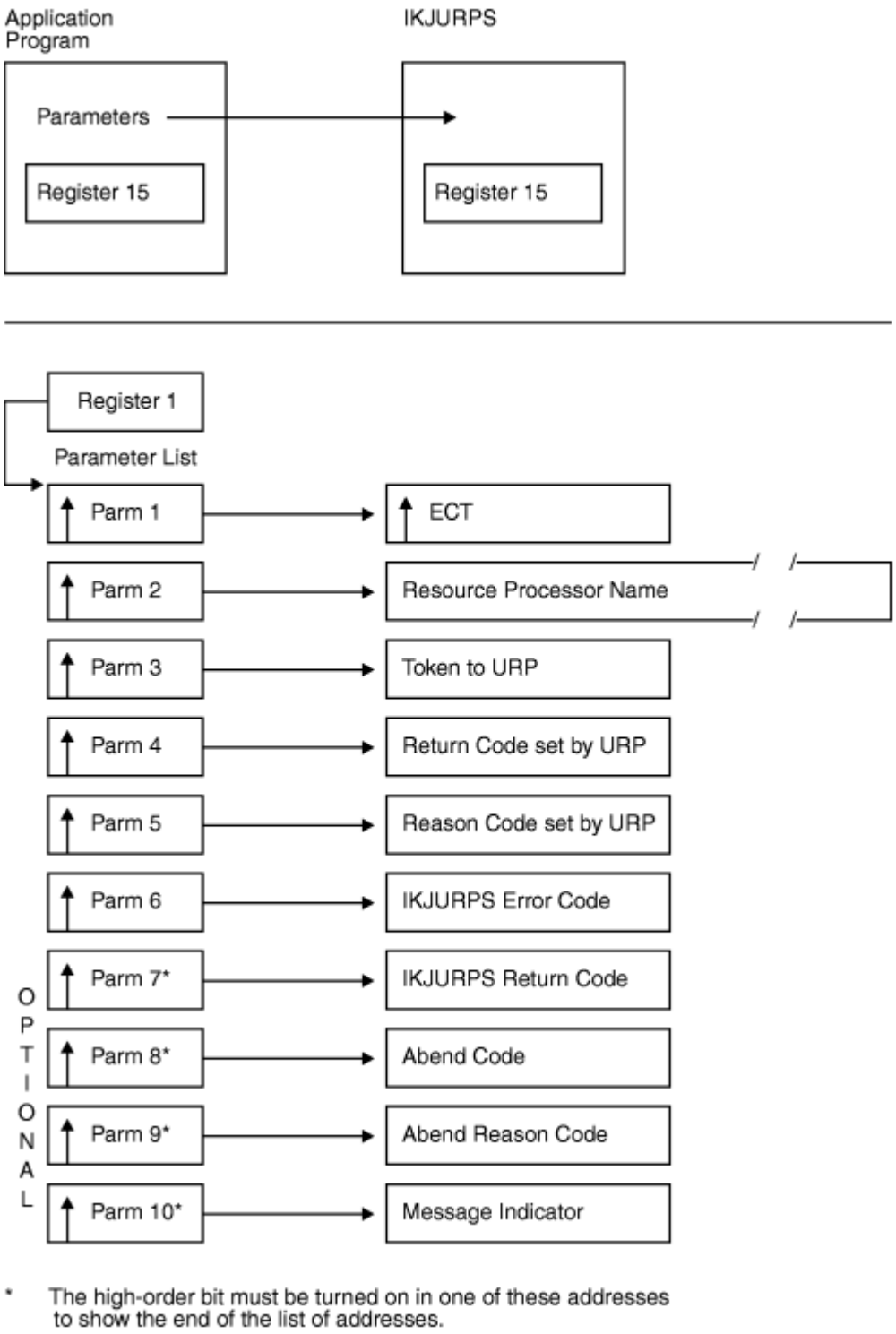


Figure 190. Parameter List for IKJURPS

The application uses register 1 to point to a list of addresses, of which each of these addresses points to a specific parameter. The parameters are:

Parameter 1 (input)
A fullword containing a pointer to an environment control table (ECT). The IKJURPS service uses this ECT when it requires the use of the TSO/E I/O service routines; for example, when IKJURPS issues a message.

Note that this ECT might not be the same ECT that the unauthorized resource processor receives in the Command Processor parameter list (CPPL) passed to it. For more information, refer to the discussion of the CPPL on entry to the resource processor and "Receiving Control in an unauthorized resource processor" on page 492.

Parameter 2 (input)
An 8-byte field containing the entry point name of the unauthorized resource processor. The unauthorized control layer uses this as the name of the module it is to invoke.

Parameter 3 (input)
A fullword containing a token passed to the unauthorized resource processor. You can use the token to communicate with the resource processor.

Parameter 4 (output)
A fullword containing a return code from the unauthorized resource processor. The invoker of the IKJURPS service can use this return code as a more detailed diagnostic; for example, if the unauthorized resource processor does not complete processing successfully.

Parameter 5 (output)
A fullword containing a reason code from the unauthorized resource processor. The invoker of the IKJURPS service can use this reason code as a more detailed diagnostic; for example, if the unauthorized resource processor does not complete processing successfully.

Parameter 6 (output)
A fullword containing an error code. The IKJURPS service sets the error code to indicate detailed diagnostics regarding completion of the IKJURPS service.

All subsequent parameters are optional. If you do not code any of the following parameters, the high-order bit of the address of this parameter must be on to indicate the end of the parameter list.

Parameter 7 (output) - optional
A fullword containing a return code. The IKJURPS service sets the return code in this parameter and register 15 when IKJURPS completes. The IKJURPS service sets this return code to indicate the type of error, if any. For example, an error due to a parameter error or an environment error. All subsequent parameters are optional. If you do not code any of the following parameters, the high-order bit of the address of this parameter must be on to indicate the end of the parameter list.

Parameter 8 (output) - optional
A fullword containing an abend code in the event of abnormal termination. The IKJURPS service returns the abend code as defined by the SWAABCC field of the SDWA. All subsequent parameters are optional. If you do not code any of the following parameters, the high-order bit of the address of this parameter must be on to indicate the end of the parameter list.

Parameter 9 (output) - optional
A fullword containing an abend reason code in the event of an abnormal termination. The IKJURPS service returns the abend reason code as defined by the SDWAHRC field of the SDWA. All subsequent parameters are optional. If you do not code any of the following parameters, the high-order bit of the address of this parameter must be on to indicate the end of the parameter list.

Parameter 10 (input) - optional
A fullword containing an indicator to the IKJURPS service that specifies whether to issue error messages. Set this parameter as follows:

X'00000000'
The IKJURPS service is not to issue error messages. This is the default if the parameter is not specified.

X'00000001'
The IKJURPS service is to issue (if appropriate).

If you code this parameter, the high-order bit of the address of this parameter must be on to indicate the end of the parameter list.

Invoking the IKJURPS Service

You can invoke the TSO/E IKJURPS service with one of the following methods:

- The CALLTSSR macro instruction, specify IKJURPS as the entry point name
- The LINK macro instruction, specify IKJURPS as the entry point name
- The address of IKJURPS that is in the TSVTURPS field of the TSO/E vector table (TSVT).

On entry to the IKJURPS service, it expects the following register linkage conventions:

Register 0
n/a

Register 1
Address of the parameter list

Registers 2–12
n/a

Register 13
Key 8 save area

Register 14
Return address

Register 15
Entry point address.

IKJURPS must be invoked in:

- 31-bit addressing mode
- Primary address space mode.

Understanding the Environment in which IKJURPS Operates

The following considerations are useful when choosing whether to use the IKJURPS service to share and manage application resources:

Applications must invoke the IKJURPS service from within an unauthorized TSO/E environment. The IKJURPS service does not complete successfully in the following environments:

- A non-TSO/E environment (that is, MVS batch)
- An authorized TSO/E environment (for example, an authorized TSO/E command or program)
- A dynamic TSO/E environment (that is, an environment created by the IKJTSOEV service)
- Any environment in which the TSO/E TMP is unable to process an IKJURPS request (for example, during LOGON processing)
- When the TSO/E TEST command is active.

For more information about the environment in which the IKJURPS service cannot successfully process the request, refer to the error codes described for the IKJURPS service return code RC=20 in "Interpreting the Return Information from the IKJURPS Service" on page 490.

Interpreting the Return Information from the IKJURPS Service

Upon return from the IKJURPS service, your application receives the following information to indicate successful or unsuccessful completion.

Table 142. Return codes from IKJURPS

Return code (decimal)	Error code (decimal)	Description
0	0	The unauthorized resource processor was invoked successfully. For more detailed diagnostics from the unauthorized resource processor, refer to the unauthorized resource processor return code and reason codes parameters (parameters 4 and 5) in the IKJURPS parameter list.

Table 142. Return codes from IKJURPS (continued)

Return code (decimal)	Error code (decimal)	Description
12 [1]	not set	Processing unsuccessful. Either the high-order bit in the address of one of the parameters other than the last was set on or the high-order bit in the address of the last parameter was not set on.
	1	Processing unsuccessful. A non-valid ECT was passed.
16 [2]	16	Processing unsuccessful; unable to load the unauthorized resource processor.
	17	Processing unsuccessful; the unauthorized resource processor abnormally ended. The abend and reason parameters, if supplied, contain the abend and reason codes.
20 [3]	20	Processing unsuccessful; the IKJURPS service was invoked in a non-TSO/E environment.
	21	Processing unsuccessful; the IKJURPS service was invoked in an authorized TSO/E environment.
	22	Processing unsuccessful; the IKJURPS service was invoked in a dynamic TSO/E environment.
	23	Processing unsuccessful; the IKJURPS service was invoked in an environment in which the TSO/E TMP cannot process an IKJURPS request. For example, during LOGON processing.
	24	Processing unsuccessful; the IKJURPS service was invoked when TEST is active.
92 [4]	20	Processing unsuccessful; the IKJURPS service could not establish a recovery environment.
	21	Processing unsuccessful; the unauthorized control layer could not establish a recovery environment for the unauthorized resource processor.
96	not set	Processing unsuccessful; a parameter is not accessible. The abend and reason parameters (parameters 8 and 9), if supplied, contain the abend and abend reason codes for further diagnosis.
100	not set	Processing unsuccessful; abnormal end. The abend and reason parameters (parameters 8 and 9), if supplied, contain the abend and abend reason codes for further diagnosis.

1

RC=12 error codes can be grouped together to indicate unsuccessful processing due to not valid parameters.

2

RC=16 error codes can be grouped together to indicate unsuccessful processing due to not valid parameters.

3

RC=20 error codes can be grouped together to indicate unsuccessful processing due to environmental errors.

4

RC=92 error codes can be grouped together to indicate unsuccessful processing due to recovery processing.

Receiving Control in an unauthorized resource processor

Your unauthorized resource processor receives control to perform application-dependent resource processing. It needs to:

1. Save and restore register contents from the unauthorized control layer.
2. Determine the type of processing it is to provide:
   a. Activate a resource. That is, to dynamically initialize resources.
   b. Make subsequent calls to further process a resource (that is, either further process that resource or terminate and cleanup that resource).
   c. Handle a terminating invocation to clean up its resources
3. Process the application's resources
4. Return information to the unauthorized control layer and the application that invoked the IKJURPS service.

Process the Application's Resources

In processing the application's resources, it is important that you understand the environment in which the unauthorized resource processor operates. Note the following restrictions:

- Because the unauthorized resource processor receives control from within the unauthorized control layer of the TSO/E TMP where ISPF is not active, ISPF services are not available within the resource processor.
- Because the unauthorized control layer fetches the resource processor, task libraries such as ISPLLIB are not available at that time. For more information concerning the search order for resource processors, refer to “Installing Resource Processors” on page 494.
- In an environment where an application is using its own I/O environment, that is, an environment created by the STACK ENVIRON=CREATE service, it is the application's responsibility to pass this ECT between the application invoking the IKJURPS service and the resource processor.
- If you prompt for input at the terminal, that is, invoke the GETLINE or PUTGET services, you must do so from within the unauthorized commands and subtasks, *not from within the unauthorized resource processor*. The unauthorized control layer defers attention processing until the resource processor completes. Therefore, if the resource processor prompts for input at the terminal and the user presses the attention key, the user has no means to interrupt or break out of the prompt. Note that the attention interrupt is not processed until the IKJURPS service is returning control to the application. Such processing can provide unexpected results for an interactive user.

Provide Return Information to the IKJEFT01 TMP Unauthorized Control Layer

Each time an unauthorized resource processor completes, it returns control to the unauthorized control layer. At this time, the unauthorized resource processor informs the unauthorized control layer whether it is either finished processing and will no longer need to be invoked or is not finished processing and will again need to be invoked.

After the unauthorized control layer invokes an unauthorized resource processor for a termination request, it makes no later calls for termination.

The unauthorized resource processor Parameter List

An unauthorized resource processor receives control with the parameter list shown in Figure 191 on page 493. It uses this parameter list to communicate with:

- The unauthorized control layer:
  - The unauthorized control layer tells you the type of request for which the unauthorized resource processor was given control, which either:

- An activate or process request
- A termination (cleanup) request.

For more information, see parameter 1.

– You tell the unauthorized control layer whether resources have been obtained. If the unauthorized resource processor obtained resources, the unauthorized control layer later invokes the unauthorized resource processorwith a termination (cleanup) request when the unauthorized control layer is terminating. For more information, see parameter 7.

- The application invoking the IKJURPS service:
  – The application invoking the IKJURPS service gives you installation-defined data in a token. For more information, see parameter 2.
  – You can pass the application invoking the IKJURPS service return and reason codes to indicate the status or your processing. For more information, see parameters 4 and 5.
- Later invocations of the same unauthorized resource processor.

  TSO/E provides you with a field where you can store and retrieve user-defined data. For more information, see parameter 3.

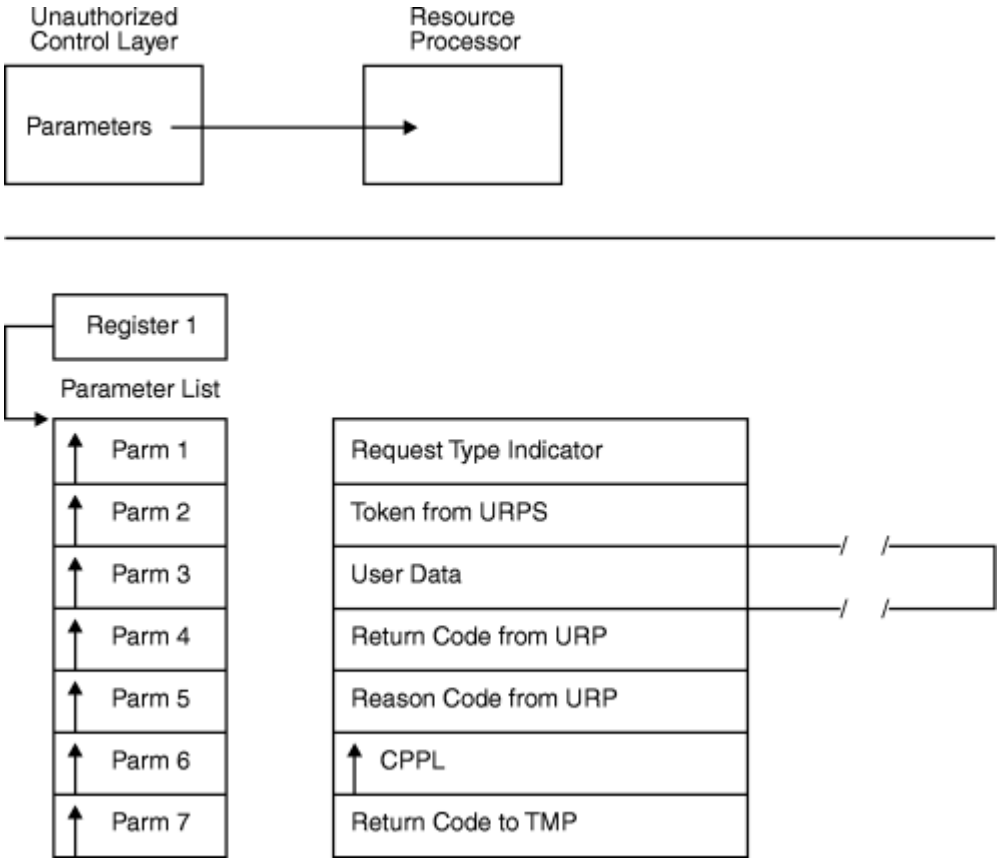


Figure 191. Parameter List Passed To An unauthorized resource processor

The unauthorized control layer uses register 1 to point to a list of addresses, of which each of these addresses points to a specific parameter. The parameters are:

Parameter 1 (input)
A fullword indicating the type of request:

X'00000001'
The application invoking the IKJURPS service requests that the unauthorized resource processor either:

1. Activate its resources. This is an initial request to the IKJURPS service.
2. Process its resources. This is a subsequent request to the IKJURPS service.

X'00000002'
The unauthorized control layer requests that the unauthorized resource processor cleanup its resources. This is a termination request.

Parameter 2 (input)
A fullword containing a copy of the token that the application that invoked the IKJURPS service passes to the unauthorized resource processor.

Parameter 3 (input/output)
An 8-byte field containing user data for the unauthorized resource processor. The unauthorized resource processor can use this parameter, for example, to hold a count of the number of times it was invoked to distinguish the difference between an activate request and a process request.

Parameter 4 (output)
A fullword containing a return code. The unauthorized resource processor sets this field to communicate with the application that invoked the IKJURPS service. The unauthorized control layer passes this value to the invoker of the IKJURPS service in parameter 4 of its parameter list.

Parameter 5 (output)
A fullword containing a reason code. The unauthorized resource processor sets this field to communicate with the application that invoked the IKJURPS service. The unauthorized control layer passed this value to the invoker of the IKJURPS service in parameter 5 of its parameter list.

Parameter 6 (input)
A fullword containing the address of a Command Processor parameter list (CPPL). The unauthorized resource processor can use the fields in the CPPL when it invokes TSO/E services that require these values. Note that this is the same CPPL that the unauthorized control layer passes to the command that it invokes.

This CPPL might not contain the ECT address that the invoker of the IKJURPS service is currently using. If this ECT address is required, you need to communicate the ECT address between the application invoking the IKJURPS service and the unauthorized resource processor. Communicate this address using parameter 2 above.

Parameter 7 (output)
A fullword containing the return code from the unauthorized resource processor to the unauthorized control layer. This parameter is recognized for an activate or process request; it is ignored for a termination request. Use this parameter to indicate to the unauthorized control layer whether to invoke the unauthorized resource processor for a later termination request. Specify one of the following:

Return code	Description
0	The unauthorized resource processor owns active resources.
4	The unauthorized resource processor no longer owns active resources.

The unauthorized control layer sets the high-order bit in the address of this parameter on to indicate the end of the parameter list.

Installing Resource Processors

You must write and install whatever unauthorized resource processors you require; they are not supplied by TSO/E nor do they have pre-determined names. You pass the name of the resource processor in parameter 2 when your application invokes the IKJURPS service.

You can link-edit all resource processors in a separate load library that is exclusively for TSO/E resource processors or in an existing library that contains other routines. Resource processors can reside in:

STEPLIB
A step library is helpful for limited use and for testing the exit before you integrate it into your system. In this case, you can easily change the exit. However, the use of a STEPLIB is not suggested for all of your users because of the extra search time required to locate and invoke the exit.

LPA
The link pack area makes the resource processors available to all of your users. However, if you need to change the processor and make the changes available in LPA for all of your users, you must re-IPL your system.

LNKLST
The linklist concatenation makes the resource processors available to all of your users while maintaining the ability to easily change the processor if required.

The search order is STEPLIB, LPA, and then LNKLST. For more information about STEPLIB, LPA, and LNKLST, refer to *z/OS MVS Initialization and Tuning Guide*.

You might also consider using System Modification Program Extended (SMP/E) when installing unauthorized resource processors. SMP/E allows you to maintain a record of the resource processors you have installed. To use SMP/E you must generate your own functional module ID (FMID) and be certain that is does not duplicate any IBM-defined FMID. For more information about SMP/E, refer to *z/OS SMP/E Commands* or *z/OS SMP/E User's Guide*.

Environment

unauthorized resource processors require the following environment:

State:
Problem program

Key:
8

AMODE
Not restricted

RMODE
Not restricted

Sample IKJURPS Invocation and unauthorized resource processor

TSO/E provides a pair of sample routines in SYS1.SAMPLIB that you can use to pattern your application's use of IKJURPS. The two routines are:

Routine Name
Description

IKJTOURP
Sample IKJURPS Invocation

This routine is a reentrant Command Processor written in assembler language. It shows you how to:

1. Receive control in a Command Processor.
2. Use the TSO/E PUTLINE service to display a message.
3. Use the CALLTSSR macro to invoke the IKJURPS service.

IKJFRURP
Sample unauthorized resource processor

This routine is a reentrant unauthorized resource processor written in assembler language. It shows you how to:

1. Receive control in an unauthorized resource processor.
2. Use the TSO/E PUTLINE service to display a message.
3. Set an appropriate return code to the unauthorized control layer of the TSO/E TMP.

Appendix A. Limits for TSO/E service routines

Services provided by TSO/E generally impose the following limits. Violation of the limits may produce unpredictable results. Other products such as ISPF and considerations such as the terminals and other media to be supported by an application may produce more restrictive limits.

Table 143. Limits

Interface	Reference	Description
Command buffer	See IKJSCAN (Chapter 5, “Verifying subcommand names with IKJSCAN,” on page 35), IKJPARS (Chapter 6, “Verifying command and subcommand operands with parse,” on page 43), I/O Service Rtns. (Chapter 9, “Using the TSO/E I/O service routines for terminal I/O,” on page 173), and IKJCAF (Chapter 13, “Using the CLIST attention facility,” on page 293) for more information.	A command buffer may be from 4 to 32767 bytes in length. The first four required bytes contain control information. The remaining bytes contain command text.
Message buffer	See I/O Service Rtns. (Chapter 9, “Using the TSO/E I/O service routines for terminal I/O,” on page 173) for more information.	A message buffer may be from 4 to 32767 bytes in length. The first four required bytes contain control information. The remaining bytes contain message text.
Text insertion buffer	See I/O Service Rtns. (Chapter 9, “Using the TSO/E I/O service routines for terminal I/O,” on page 173) for more information.	A text insertion buffer may be from 4 to 32767 bytes in length. The first four required bytes contain control information. The remaining bytes contain message text. The length is further limited by the consideration that the message composed by incorporating the text from the buffer must be no longer than 32767 bytes in length.
Message template described by IKJTSMSG macro.	See IKJEFF02 (Chapter 11, “Using the TSO/E message handling routine IKJEFF02,” on page 275) for more information.	A message template consists of 1. From 1 to 255 parts, counting each block of text in the message template as one part and each insertion position as one part. 2. Blocks of text from 1 to 255 bytes in length.
IKJPARS parameter control list (PCL)	See IKJPARS (Chapter 6, “Verifying command and subcommand operands with parse,” on page 43) for more information.	A PCL may be from 8 to 32767 bytes in length.
IKJPARS parameter descriptor list (PDL)	See IKJPARS (Chapter 6, “Verifying command and subcommand operands with parse,” on page 43) for more information.	A PDL may be from 8 to 32767 bytes in length.
CLIST variable name	See IKJCT441 (Chapter 24, “Using the variable access routine IKJCT441,” on page 409) for more information.	The name of a CLIST variable can be from 1 to 252 bytes in length.
CLIST variable value	See IKJCT441 (Chapter 24, “Using the variable access routine IKJCT441,” on page 409) for more information.	The value of a CLIST variable can be from 0 to 32,756 bytes in length.
REXX variable name	See IKJCT441 (Chapter 24, “Using the variable access routine IKJCT441,” on page 409) for more information.	The name of a REXX variable can be from 1 to 250 bytes in length.

Table 143. Limits (continued)

Interface	Reference	Description
REXX variable value	See IKJCT441 (Chapter 24, "Using the variable access routine IKJCT441," on page 409) for more information.	The value of a REXX variable can be from 0 to 16,777,215 bytes in length.

Appendix B. Accessibility

Accessible publications for this product are offered through IBM Documentation (www.ibm.com/docs/en/zos).

If you experience difficulty with the accessibility of any z/OS information, send a detailed message to the Contact the z/OS team web page (www.ibm.com/systems/campaignmail/z/zos/contact_z) or use the following mailing address.

IBM Corporation
Attention: MHVRCFS Reader Comments
Department H6MA, Building 707
2455 South Road
Poughkeepsie, NY 12601-5400
United States

Notices

This information was developed for products and services that are offered in the USA or elsewhere.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
United States of America

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

This information could include missing, incorrect, or broken hyperlinks. Hyperlinks are maintained in only the HTML plug-in output for IBM Documentation. Use of hyperlinks in other output formats of this information is at your own risk.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Site Counsel
2455 South Road

Poughkeepsie, NY 12601-5400
USA

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

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

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Terms and conditions for product documentation

Permissions for the use of these publications are granted subject to the following terms and conditions.

Applicability

These terms and conditions are in addition to any terms of use for the IBM website.

Personal use

You may reproduce these publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative work of these publications, or any portion thereof, without the express consent of IBM.

Commercial use

You may reproduce, distribute and display these publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these publications, or

reproduce, distribute or display these publications or any portion thereof outside your enterprise, without the express consent of IBM.

Rights

Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the publications or any information, data, software or other intellectual property contained therein.

IBM reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the publications is detrimental to its interest or, as determined by IBM, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations.

IBM MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.

IBM Online Privacy Statement

IBM Software products, including software as a service solutions, ("Software Offerings") may use cookies or other technologies to collect product usage information, to help improve the end user experience, to tailor interactions with the end user, or for other purposes. In many cases no personally identifiable information is collected by the Software Offerings. Some of our Software Offerings can help enable you to collect personally identifiable information. If this Software Offering uses cookies to collect personally identifiable information, specific information about this offering's use of cookies is set forth below.

Depending upon the configurations deployed, this Software Offering may use session cookies that collect each user's name, email address, phone number, or other personally identifiable information for purposes of enhanced user usability and single sign-on configuration. These cookies can be disabled, but disabling them will also eliminate the functionality they enable.

If the configurations deployed for this Software Offering provide you as customer the ability to collect personally identifiable information from end users via cookies and other technologies, you should seek your own legal advice about any laws applicable to such data collection, including any requirements for notice and consent.

For more information about the use of various technologies, including cookies, for these purposes, see IBM's Privacy Policy at ibm.com®/privacy and IBM's Online Privacy Statement at ibm.com/privacy/details in the section entitled "Cookies, Web Beacons and Other Technologies," and the "IBM Software Products and Software-as-a-Service Privacy Statement" at ibm.com/software/info/product-privacy.

Policy for unsupported hardware

Various z/OS elements, such as DFSMSdfp, JES2, JES3, and MVS, contain code that supports specific hardware servers or devices. In some cases, this device-related element support remains in the product even after the hardware devices pass their announced End of Service date. z/OS may continue to service element code; however, it will not provide service related to unsupported hardware devices. Software problems related to these devices will not be accepted for service, and current service activity will cease if a problem is determined to be associated with out-of-support devices. In such cases, fixes will not be issued.

Minimum supported hardware

The minimum supported hardware for z/OS releases identified in z/OS announcements can subsequently change when service for particular servers or devices is withdrawn. Likewise, the levels of other software products supported on a particular release of z/OS are subject to the service support lifecycle of those

products. Therefore, z/OS and its product publications (for example, panels, samples, messages, and product documentation) can include references to hardware and software that is no longer supported.

- For information about software support lifecycle, see: IBM Lifecycle Support for z/OS (www.ibm.com/software/support/systemsz/lifecycle)
- For information about currently-supported IBM hardware, contact your IBM representative.

Programming Interface Information

This document describes intended Programming Interfaces that allow the customer to write programs to obtain the services of z/OS TSO/E.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at Copyright and Trademark information (www.ibm.com/legal/copytrade.shtml).

Index

Numerics

A

B

C

D

E

F

G

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Product Number: 5650-ZOS

SA32-0973-50