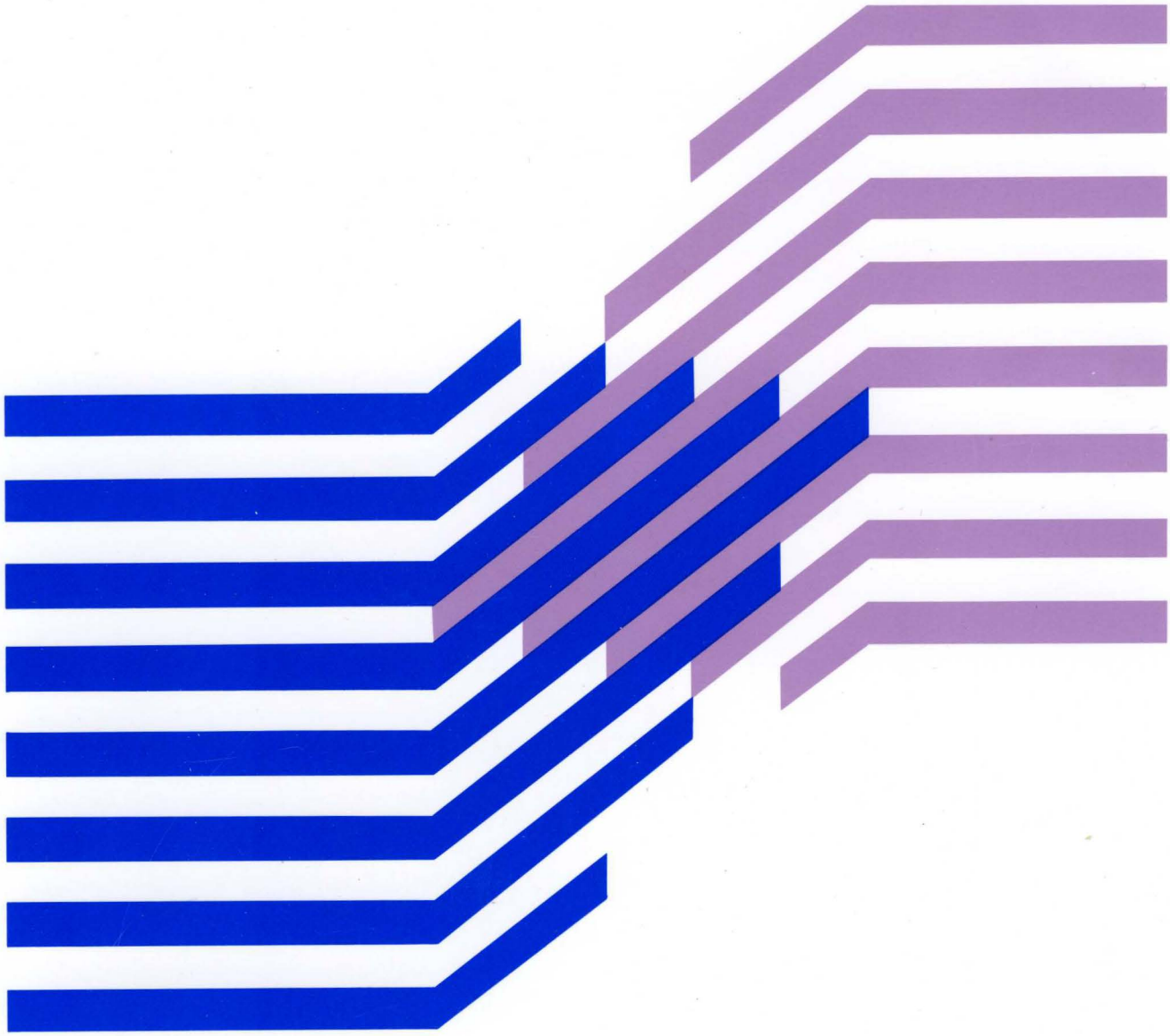
GC28-1565-2

MVS/ESA
TSO Programming

MVS/System Product:
JES2 Version 3
JES3 Version 3



GC28-1565-2

MVS/ESA
TSO Programming

MVS/System Product:
JES2 Version 3
JES3 Version 3

Production of This Book

This book was prepared and formatted using the IBM BookMaster document markup language.

Third Edition (May 1991)

This is a major revision of, and obsoletes, GC28-1565-1, and Technical Newsletters GN28-1548 and GN28-1486. See the Summary of Changes regarding new and changed information made to this publication. Technical changes and additions to the text and illustrations are indicated by a vertical line to the left of the change.

This edition applies to Version 3 of MVS/System Product 5685-001 or 5685-002 and to all subsequent releases until otherwise indicated in new editions or Technical Newsletters. Changes are made periodically to the information herein; before using this publication with the operation of IBM systems, consult the latest *IBM System/370 Bibliography*, GC20-0001, for the editions that are applicable and current.

References in this publication to IBM products or services do not imply that IBM intends to make these available in all countries in which IBM operates. References to IBM products in this document do not imply that functionally equivalent products may be used. The security certification of the trusted computing base that includes the products discussed herein covers certain IBM products. Please contact the manufacturer of any product you may consider to be functionally equivalent for information on that product's security classification. This statement does not expressly or implicitly waive any intellectual property right IBM may hold in any product mentioned herein.

Publications are not stocked at the address given below. Requests for IBM publications should be made to your IBM representative or to the IBM branch office serving your locality.

A form for reader's comments is provided at the back of this publication. If the form has been removed, comments may be addressed to IBM Corporation, Information Development, Department D58, Building 921, PO Box 950, Poughkeepsie, New York 12602

IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

PROGRAMMING INTERFACES

The information in this book is provided to allow a customer installation to write programs that use the services of MVS/System Product Version 3 in a TSO environment. The majority of this book consists of general-use programming interface information. However, this book also contains guidance information for programmers who design and write programs that run under TSO. Such information should never be used as programming interface information.

General-Use Programming Interfaces: General-use programming interfaces do not have significant dependencies on detailed product design or implementation.

General-use programming interface information is identified by brackets before and after the information, as follows:

GENERAL-USE PROGRAMMING INTERFACE

Description of the interface.

End of GENERAL-USE PROGRAMMING INTERFACE

About This Book

MVS/ESA TSO Programming describes the services and commands that TSO provides for use in writing system and application programs. This book also describes how to write, install, and execute a command processor.

Trademarks

The following are trademarks of International Business Machines Corporation.

- MVS/ESA™
- MVS/SP™
- MVS/XA™

Who This Book Is For

This book is for the following audience:

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

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

How This Book Is Organized

This book is divided into three parts:

- **Part I** describes how to write, install and execute a command processor. It discusses the TSO services that you can use in a command processor, and refers you to Part II of this book for more information, when needed.
- **Part II** describes the programming services that you can use in system or application programs.
- **Part III** describes the syntax and function of the TSO commands, and provides examples of how to use them.

How to Use This Book

If you have never used this book, read Chapter 1, "Introduction" on page 1 to become familiar with command processors and the programming services and commands that TSO provides. Then refer to the individual chapter that discusses a particular topic.

Related Information

You need the following publications for reference:

MVS Publications

MVS/ESA Application Development Guide, GC28-1821

MVS/ESA Application Development Macro Reference, GC28-1822

MVS/ESA JCL Reference, GC28-1829

MVS/ESA JCL User's Guide, GC28-1830

MVS/ESA Message Library: System Codes, GC28-1815

MVS/ESA Message Library: System Messages, Volumes 1 and 2, GC28-1812 and GC28-1813

MVS/ESA System Programming Library: Application Development — 31-Bit Addressing, GC28-1820

MVS/ESA System Programming Library: Application Development — Extended Addressability, GC28-1854

MVS/ESA System Programming Library: Application Development Guide, GC28-1852

MVS/ESA System Programming Library: Application Development Macro Reference, GC28-1857

MVS/ESA System Programming Library: Initialization and Tuning, GC28-1828

Contents

Figures

Summary of Changes

Summary of Changes
for GC28-1565-2
MVS/System Product Version 3 Release 1.3

Changed Information:

Service updates.

Chapter 1. Introduction

Command processors are a specific type of program that you can write to run in the TSO environment. You can write your own command processors to add to the set of commands provided by TSO.

TSO provides programming services that support a wide range of functions. You can use the programming services described in this book in system or application programs, including command processors.

Executing TSO Commands

To execute TSO commands, you must write JCL statements and submit them to the operating system. Use the SYSTSIN DD statement in your JCL to control input to your job and indicate which commands are to be executed. Chapter 9, "Executing a Command Processor" on page 39 describes the JCL statements you must write to execute a command.

Identifying Authorized Programs and Commands

To allow TSO users to execute authorized and unauthorized programs within a single job step, a system programmer must maintain the access lists in CSECTs IKJEFTE2 and IKJEFTE8.

The IBM-supplied lists for APFCTABL (in IKJEFTE2) and APFPTABL (in IKJEFTE8) contain blank entries which inhibit the execution of APF-authorized programs. The APFCTABL list contains the names of authorized command processors executed by the TMP, and the APFPTABL list contains the names of authorized programs to be executed by the CALL command. The modules that are attached for these names must be link edited with APF authorization. If a name does not appear in these lists, the program is attached without authorization. If a program is to be executed by both the TMP and the CALL command, then its name must appear in both lists.

The format of the list is a sequence of eight-character command name entries. This list is terminated by an entry consisting of eight blanks. Command name entries of less than eight characters must be left-justified and padded to the right with blanks to fill the eight-character entry.

The first entry to be examined by the TMP in either IKJEFTE2 or IKJEFTE8 will be that entry associated with the respective ENTRY name APFCTABL or APFPTABL. If a command has an abbreviation, it must appear as a separate entry. A null list consists of just the final eight blanks.

For example, if commands R1USER with abbreviation R1 and P3SRCH are to be executed with authorization, then the list should look like:

```
          ENTRY     APFCTABL
IKJEFTE2  CSECT
          DC        CL8'IKJEFTE2'
          DC        CL8'  76.133'     DATE MAY CHANGE
APFCTABL  DC        CL8'R1USER  '
          DC        CL8'R1      '
          DC        CL8'P3SRCH  '
          DC        CL8'        '
          END
```

If an installation wishes to allow access to IEBCOPY through CALL, then the list should look like:

```
          ENTRY     APFPTABL
IKJEFTE8  CSECT
          DC        CL8'IKJEFTE8'
          DC        CL8'  76,133'     DATE MAY CHANGE
APFPTABL  DC        CL8'IEBCOPY '
          DC        CL8'        '
          END
```

The lists in APFCTABL and APFPTABL must contain only the eight-character strings. The installation can reserve extra space by additional terminal blank strings. Nonblank entries following a blank entry are not examined.

You can replace the IBM-supplied modules IKJEFTE2 and IKJEFTE8 by link editing installation-supplied modules with these names into TMP load module IKJEFT02 in SYS1.LPALIB.

Consult the output from stage 1 for correct link edit information. Any program that depends upon a job step environment such as the TMP should not be placed in the lists.

Writing Command Processors

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

A command processor is a program written in assembler language that receives control when a command name is specified in the input data controlled by the SYSTSIN DD statement in a user's JCL. It is given control by the terminal monitor program (TMP), a program that provides an interface between TSO 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 through the job output data set and obtain input data, as needed. Command processors can recognize subcommand names specified in the input data 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 services to use, and how to install and execute the command processor, refer to Part I.

Overview of TSO Programming Services

TSO provides services that your programs can use to perform the tasks described below. Figure 1 summarizes the services provided by TSO.

Figure 1. Summary of TSO Services

Task	Service	Chapter Reference
Invoking TSO service routines	CALLTSSR macro instruction	Chapter 11
Checking the syntax of subcommand names	Command scan service routine	Chapter 12
Checking the syntax of command and subcommand operands	Parse service routine	Chapter 13
Processing I/O	TSO I/O service routines TSO Message Handling Routine	Chapter 14 Chapter 15
Allocating, concatenating and freeing data sets	Dynamic allocation interface routine	Chapter 16
Analyzing return codes	DAIRFAIL GNRLFAIL/VSAMFAIL	Chapter 17 Chapter 18
Retrieving information from the system catalog	Catalog information routine	Chapter 19

Invoking TSO Service Routines

To pass control to certain TSO service routines, use the CALLTSSR macro instruction. See Chapter 11, "Invoking TSO Service Routines with the CALLTSSR Macro Instruction" on page 53.

Checking the Syntax of Subcommand Names

Use the command scan service routine in your command processors to validate a subcommand name. See Chapter 12, "Verifying Subcommand Names with the Command Scan Service Routine" on page 55.

Checking the Syntax of Command and Subcommand Operands

Use the parse service routine to validate command or subcommand operands. See Chapter 13, "Verifying Command and Subcommand Operands with the Parse Service Routine" on page 63.

Processing I/O

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

- You can use the TSO 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. The I/O service routines can be used to issue messages. See Chapter 14, "Using the TSO I/O Service Routines" on page 153.

- Your command processors can use the TSO message issuer routine (IKJEFF02) to issue messages to the output data set. See Chapter 15, "Using the TSO Message Handling Routine (IKJEFF02)" on page 201.

Processing Data Sets

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

Allocating, Concatenating and Freeing Data Sets

TSO 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, using SVC 99.* For a complete discussion of dynamic allocation, see *SPL: Application Development Guide*. DAIR is discussed in Chapter 16, "Using the Dynamic Allocation Interface Routine (DAIR)" on page 207.

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 ID. See Chapter 19, "Using IKJEHCIR to Retrieve System Catalog Information" on page 239.

Analyzing Return Codes

Use the DAIRFAIL routine (IKJEFF18) to analyze return codes from dynamic allocation (SVC 99) or DAIR and issue appropriate error messages. See Chapter 17, "Using the DAIRFAIL Routine (IKJEFF18)" on page 231.

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 18, "Analyzing Error Conditions with the GNRLFAIL/VSAMFAIL Routine (IKJEFF19)" on page 235.

Overview of TSO Commands

The following commands are provided by TSO:

Command	Function
CALL	Loads and executes a program.
TIME	Provides the date and time of day.
WHEN/END	Tests return codes from programs invoked from an immediately preceding CALL command, and takes a prescribed action if the return code meets a specified condition.

For information about the TSO commands, refer to Part III of this book.

Part I: Writing and Executing a Command Processor

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

A *command processor* is a program that is given control by the terminal monitor program (TMP) when you specify the command name as input data to a job that executes the TMP. The TMP provides an interface between TSO users and command processors and has access to many system services.

If you choose to write your own command processors, you can use the programming services provided by TSO to perform many of the functions required by a command processor. The programming services available in TSO consist of service routines and macros, and are discussed in "Part II: TSO Programming Services" on page 41.

Part I of this book contains several chapters that describe what you must do to write, install, and execute a command processor. Chapter 2 presents the concepts and terminology that you must understand before you read the later chapters. Chapter 3 outlines the steps to follow when writing a command processor and refers you to later chapters for the details of each step. Read all of chapters 2 and 3 and then selectively read the subsequent chapters.

Chapter 2. What is a Command Processor?

A command processor is a program invoked by the terminal monitor program (TMP) when you specify the command name as input data to a job that executes the TMP. The TMP is a program that accepts and interprets commands, and causes the appropriate command processor to be scheduled and executed. The TMP also communicates with the user through the output data set and responds to abnormal terminations.

The TSO Environment

The TMP determines whether data in the input stream is a command name. If a command is specified, the TMP attaches the requested command processor and the command processor then performs the functions requested by the user. When the command processor completes and returns control to the TMP, the TMP detaches the command processor.

The Command Processor Parameter List (CPPL)

The CPPL is a four-word parameter list that is located in subpool 1. The control block interface between the TMP and an attached command processor is shown in Figure 2.

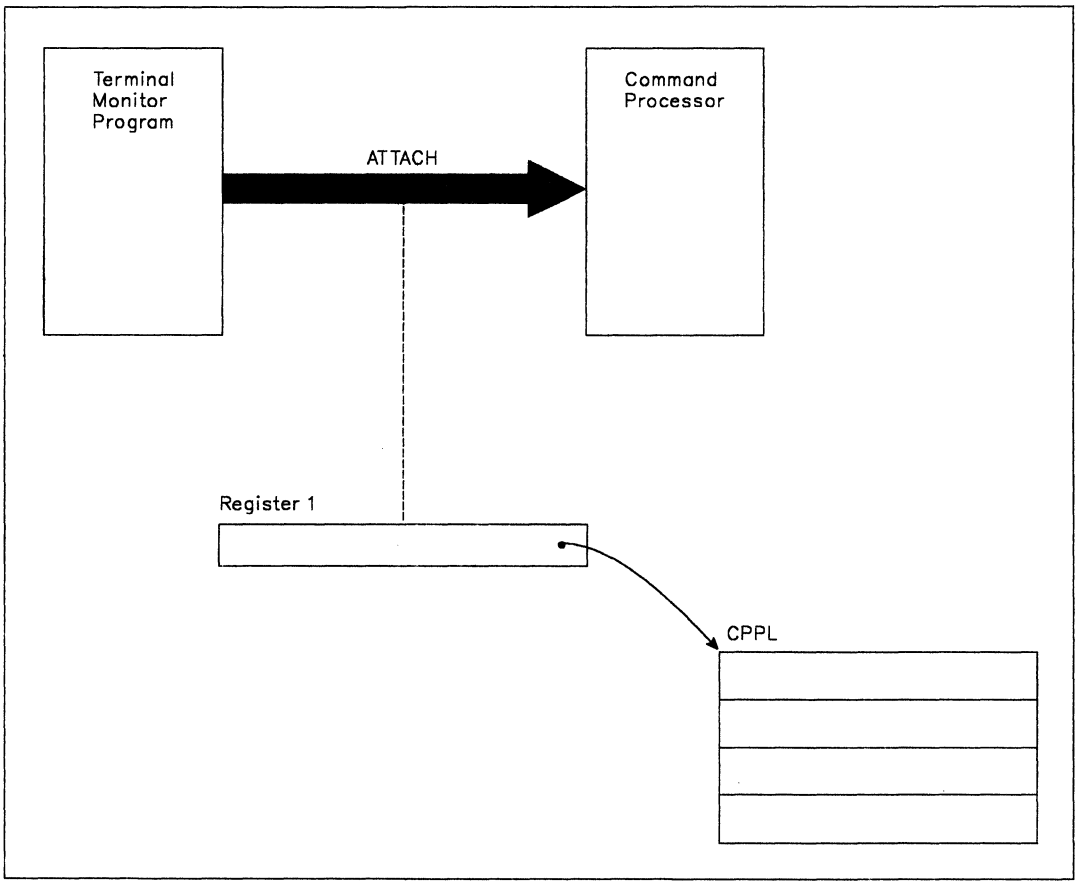


Figure 2. Control Block Interface between the TMP and a Command Processor

GENERAL-USE PROGRAMMING INTERFACE

When the terminal monitor program attaches a command processor, register 1 contains a pointer to a command processor parameter list (CPPL) containing addresses required by the command processor. Figure 3 describes the contents of the CPPL.

Figure 3. The Command Processor Parameter List (CPPL)

Number of Bytes	Field	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.

The first word of the CPPL contains the address of the command buffer for the currently attached command processor. As the TMP receives a line of input from the input stream, the input is placed into the command buffer. After determining that the input is a command name, the TMP attaches the appropriate command processor. Figure 4 shows the format of the command buffer.

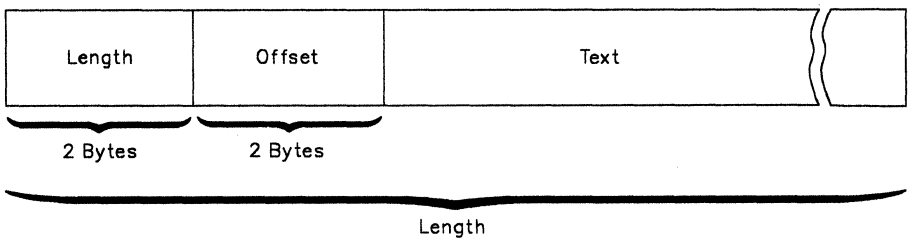


Figure 4. Format of the Command Buffer

When your command processor receives control, the fields in the command buffer appear as follows:

- The two-byte length field contains the length of the command buffer, including the four-byte header.
- If the user specified operands, the offset field contains the number of text bytes preceding the first operand. Otherwise, the offset field contains the length of the text portion of the buffer.
- The text field contains the command name, in uppercase characters, followed by any operands the user specified.

End of GENERAL-USE PROGRAMMING INTERFACE

Command Syntax

A command consists of a command name, optionally followed by one or more *operands*. Operands provide the specific information required for the command processor to perform the requested operation.

There are two types of operands that can follow a command name: *Positional* operands and *keyword* operands. Positional operands immediately follow the command name and must be in a specific order. Keyword operands are specific names or symbols that have a particular meaning to the command processor. A TSO user can enter keyword operands anywhere in the command line as long as they follow all positional operands. A keyword operand can have a *subfield* associated with it. A subfield consists of a parenthesized list of positional or keyword operands directly following the keyword.

A TSO user can enter comments in the command line anywhere a blank might appear by enclosing the text within the delimiters /* and */.

What is a Subcommand Processor?

If your command processor must perform a large number of complex functions, you can divide this work into individual operations. Each operation can be defined and performed by a *subcommand processor*. A TSO user requests one of the operations by first specifying the name of the command, and then specifying a subcommand to indicate which individual operation should be performed.

Subcommands are similar to commands in many ways, including syntax and the way they are given control. A subcommand processor is attached by the command processor and is passed a pointer to the CPPL in register 1.

Chapter 3. What You Need to Do to Write a Command Processor

This chapter describes the steps to follow when writing, installing and executing a command processor. Further details are contained in subsequent chapters.

1. **Write the assembler language program.**

   - Access the command processor parameter list (CPPL).

     When a command processor receives control from the TMP, register 1 contains the address of the CPPL. Use the IKJCPPL DSECT, provided in SYS1.MACLIB, to map the fields in the CPPL. Your command processor can then access the symbolic field names within the IKJCPPL DSECT by using the address contained in register 1 as the starting address for the DSECT. The use of the DSECT is recommended since it protects the command processor from any changes to the CPPL.

   - Validate any operands entered with the command.

     Your command processor must verify that the operands the user specified on the command are valid. Use the parse service routine (IKJPARS) to scan and verify the operands. See Chapter 4, "Validating Command Operands" on page 13 for a description of the functions provided by the parse service routine.

   - Communicate with the user through the job stream.

     Your command processor may need to obtain data from the input stream, and write messages or data to the output data set. For information on I/O, see Chapter 5, "Communicating with the User through the Job Stream" on page 25.

   - Perform the function of the command according to any operands the user specified.

     The operands that the user specified on the command indicate which functions your command processor should perform. You can use system services and the services provided by TSO to perform many functions.

   - Recognize and pass control to any subcommands.

     If you have chosen to implement subcommands, your command processor must be able to recognize a subcommand name specified in the input stream and pass control to the requested subcommand processor. For a description of the steps involved, see Chapter 6, "Passing Control to Subcommand Processors" on page 29.

   - Intercept and process abnormal terminations.

     Your command processor must be able to intercept abnormal terminations and perform the processing needed to prevent abnormal termination of the job step. For information on writing error handling routines, see Chapter 7, "Processing Abnormal Terminations" on page 33.

GENERAL-USE PROGRAMMING INTERFACE

- Set the return code in register 15 and return control to the TMP.

  When returning control to the TMP, your command processor must follow standard linkage conventions and set a return code in register 15. Your command processor should set one of the following return codes in register 15:

Return Code Dec(Hex)	Meaning
0(0)	The command processor has executed normally.
12(C)	An error encountered during execution has caused the command processor to terminate.

End of GENERAL-USE PROGRAMMING INTERFACE

2. **Assemble the command processor.**

   After you code your command processor, you must assemble the source into object code and place it in an object module.

3. **Install the command processor.**

   For a description of the methods that you can use to add your new command processor to TSO, see Chapter 8, "Installing a Command Processor" on page 37.

Chapter 4. Validating Command Operands

When your command processor receives control, it must verify that operands entered with the command are valid and that required operands are specified. This chapter introduces the parse service routine and describes how it can be used to determine the validity of command operands. For a complete description of the parse service routine, see Chapter 13, "Verifying Command and Subcommand Operands with the Parse Service Routine" on page 63.

Using the Parse Service Routine

When you write a command processor to run under TSO, you need a method to determine whether the command operands specified by the user are syntactically correct. The parse service routine (IKJPARS) performs this function by searching the command buffer for valid operands.

Parse recognizes positional 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.

Although parse recognizes comments present in the command buffer, it processes them by simply skipping over them. Comments, which are indicated by the delimiters /* and */, are not removed from the command buffer.

Before invoking the parse service routine, your command processor must use the parse macro instructions to create a parameter control list (PCL), which describes the permissible operands. You then invoke the parse service routine to compare 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).

Parse 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), indicate which operands are present in the command buffer. These operands indicate to your command processor the functions the user is requesting.

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, PDL and command buffer are included in the parse parameter list.

When the parse service routine finishes processing, it passes a return code in register 15 to your command processor. Your command processor should issue meaningful error messages for all non-zero return codes. The GNRLFAIL routine, which is discussed in Chapter 18, "Analyzing Error Conditions with the GNRLFAIL/VSAMFAIL Routine (IKJEFF19)" on page 235, can be used for this purpose.

Figure 5 on page 14 shows the interaction between a command processor and the parse service routine.

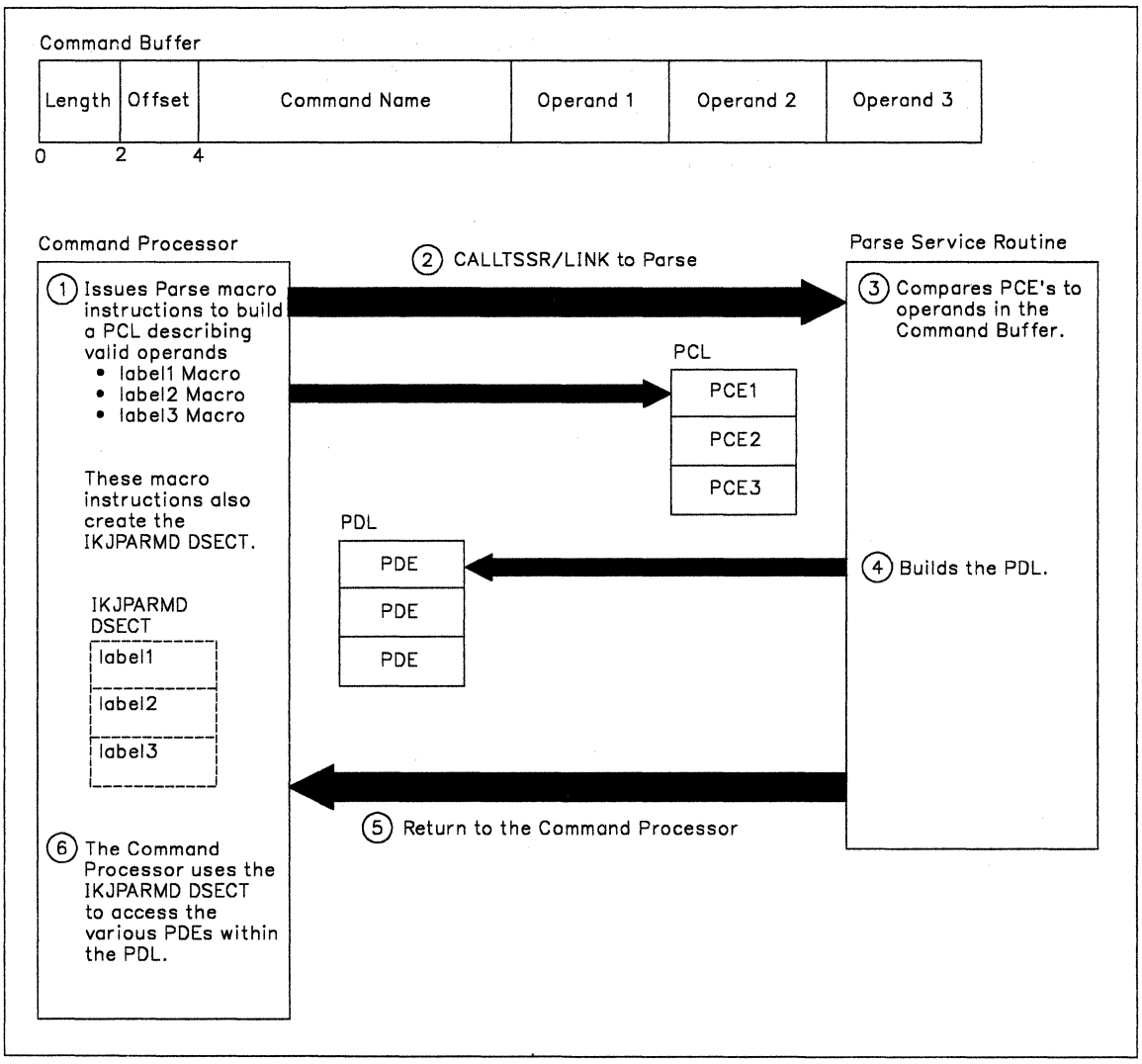


Figure 5. A Command Processor Using the Parse Service Routine

Checking Positional Operands for Logical Errors

Because the parse service routine checks the command operands only for syntax errors, you must write validity checking routines when it is also necessary to check positional operands for logical errors. Each positional operand can have a unique validity checking routine.

To indicate that a validity checking routine is to receive control, code the entry point address of the routine on the parse macro instruction that describes the operand. The validity checking routine you provide for a positional operand receives control after the parse service routine determines that the operand is specified and is syntactically valid.

When parse passes control to a validity checking routine, it passes a validity check parameter list, which contains the address of the PDE parse built to describe the positional operand. Your validity checking routine can use the information in the PDE to perform additional checking on the operand.

When processing is complete, the validity checking routine must pass a return code in general register 15 to the parse service routine. The return code informs parse of the results of the validity check and determines the action that parse takes.

A Sample Command Processor

The sample command processor in Figure 6 demonstrates the use of the parse service routine. A validity checking routine is also provided. The syntax for the sample command is:

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

where dsname is a positional operand and ACTION/NOACTION are keyword operands. NOACTION is the default if neither ACTION nor NOACTION are specified.

```
PROCESS  TITLE 'SAMPLE TSO COMMAND PROCESSOR
PROCESS  CSECT ,
PROCESS  AMODE 24                    COMMAND'S ADDRESSING MODE
PROCESS  RMODE 24                    COMMAND'S RESIDENCY  MODE
*********************************************************************
*                                                                   *
* TITLE - PROCESS                                                   *
*                                                                   *
* DESCRIPTION - SAMPLE TSO COMMAND PROCESSOR                        *
*                                                                   *
* FUNCTION -  THIS SIMPLE COMMAND PROCESSOR DEMONSTRATES THE USE    *
*             OF THE PARSE SERVICE ROUTINE TO SYNTAX CHECK THE      *
*             COMMAND OPERANDS.                                     *
*                                                                   *
* OPERATION - PROCESS IS A REENTRANT COMMAND PROCESSOR THAT PERFORMS *
*             THE FOLLOWING PROCESSING:                             *
*                                                                   *
*    1 - ESTABLISHES ADDRESSABILITY AND SAVES THE CALLER'S REGISTERS *
*    2 - ISSUES A GETMAIN FOR DYNAMIC STORAGE                       *
*    3 - USES THE PARSE SERVICE ROUTINE (IKJPARS) TO DETERMINE THE  *
*        VALIDITY OF THE COMMAND OPERANDS                           *
*    4 - PROVIDES A VALIDITY CHECKING ROUTINE TO PERFORM ADDITIONAL *
*        CHECKING OF THE POSITIONAL OPERAND                         *
*    5 - ISSUES A FREEMAIN TO RELEASE THE DYNAMIC STORAGE           *
*    6 - RESTORES THE CALLER'S REGISTERS BEFORE RETURNING           *
*    7 - RETURNS TO THE TMP WITH A RETURN CODE IN REGISTER 15       *
*                                                                   *
*********************************************************************
*
PROCESS  CSECT
         STM   R14,R12,12(R13)       SAVE CALLER'S REGISTERS
         LR    R11,R15               ESTABLISH ADDRESSABILITY WITHIN
         USING PROCESS,R11           THIS CSECT
         LR    R2,R1                 SAVE THE POINTER TO THE CPPL
*                                     AROUND THE GETMAIN
         GETMAIN RU,LV=L_SAVE_AREA   OBTAIN A DYNAMIC WORK AREA
         USING SAVEAREA,R1           AND ESTABLISH ADDRESSABILITY
         ST    R1,8(R13)             PUT THE ADDRESS OF PROCESS'S SAVE
*                                     AREA INTO THE CALLER'S SAVE AREA
         ST    R13,B_PTR             PUT THE ADDRESS OF PROCESS'S SAVE
*                                     AREA INTO ITS OWN SAVE AREA
         LR    R13,R1                LOAD GETMAINED AREA ADDRESS
         USING SAVE_AREA,R13         POINT TO THE DYNAMIC AREA
         DROP  R1                    DON'T USE R1 ANY MORE
```

Figure 6 (Part 1 of 9). A Sample Command Processor

```
         GETMAIN RU,LV=L_WORK_AREA   OBTAIN A DYNAMIC WORK AREA
         USING WORKA,R1              AND ESTABLISH ADDRESSABILITY TO
*                                    THE DYNAMIC WORK AREA
         STM   R0,R1,WORK_AREA_GM_LENGTH  SAVE LENGTH AND ADDR OF
*                                    DYNAMIC AREA
         LR    R10,R1                GET READY TO USE R10 AS THE
         USING WORKA,R10             DATA AREA SEGMENT BASE REGISTER
         DROP  R1
         ST    R2,CPPL_PTR           SAVE THE POINTER TO THE CPPL
*********************************************************************
*                                                                   *
*    MAINLINE PROCESSING                                            *
*                                                                   *
*********************************************************************
*
         XC    RETCODE,RETCODE       INITIALIZE THE RETURN CODE
         GETMAIN RU,LV=L_PPL         OBTAIN A DYNAMIC PPL WORK AREA
         STM   R0,R1,PPL_LENGTH      SAVE LENGTH AND ADDR OF DYNAMIC PPL
         GETMAIN RU,LV=L_ANSWER      OBTAIN A DYNAMIC PPL ANSWER AREA
         STM   R0,R1,ANSWER_LENGTH   SAVE LENGTH AND ADDR OF DYNAMIC PPL
*                                    ANSWER AREA
         L     R2,PPL_PTR            GET THE ADDRESS OF THE PPL
         USING PPL,R2                AND ESTABLISH ADDRESSABILITY
         L     R1,CPPL_PTR           GET ADDRESS OF CPPL
         USING CPPL,R1               AND ESTABLISH ADDRESSABILITY
         MVC   PPLUPT,CPPLUPT        PUT IN THE UPT ADDRESS FROM CPPL
         MVC   PPLECT,CPPLECT        PUT IN THE ECT ADDRESS FROM CPPL
         MVC   PPLCBUF,CPPLCBUF      PUT IN THE COMMAND BUFFER ADDRESS
*                                    FROM THE CPPL
         L     R1,WORK_AREA_GM_PTR   GET THE ADDRESS OF THE COMMAND
*                                    PROCESSOR'S DYNAMIC WORK AREA TO
         ST    R1,PPLUWA             BE PASSED TO THE VALIDITY CHECK
*                                    ROUTINE
         DROP R1
         L     R1,ANSWER_PTR         GET THE ADDRESS OF THE PARSE
*                                    ANSWER AREA AND
         ST    R1,PPLANS             STORE IT IN THE PPL
         XC    ECB,ECB               CLEAR COMMAND PROCESSOR'S
*                                    EVENT CONTROL BLOCK (ECB)
         LA    R1,ECB                GET THE ADDRESS OF THE COMMAND
*                                    PROCESSOR'S ECB AND
         ST    R1,PPLECB             PUT IT IN THE PPL
         L     R1,PCLADCON           GET THE ADDRESS OF THE PCL AND
         ST    R1,PPLPCL             PUT IT IN THE PPL FOR PARSE
         CALLTSSR EP=IKJPARS,MF=(E,PPL)   INVOKE PARSE
         DROP R2
         LTR   R15,R15               IF PARSE RETURN CODE IS ZERO
         BZ    PROCESS               PERFORM PROCESSING FOR THE COMMAND
         MVC   RETCODE(4),ERROR      SET CP RETURN CODE TO 12
         B     CLEANUP               PREPARE TO RETURN TO THE TMP
```

Figure 6 (Part 2 of 9). A Sample Command Processor

```
*
PROCESS  DS 0H
*         .
*         .
*         .
*
*  CODE TO PERFORM THE FUNCTION OF THE COMMAND PROCESSOR GOES HERE.
*  AFTER CALLING THE PARSE SERVICE ROUTINE TO VALIDATE THE COMMAND
*  OPERANDS, USE THE PDL RETURNED BY PARSE TO DETERMINE WHICH
*  OPERANDS THE USER ENTERED.  THEN PERFORM THE FUNCTION REQUESTED
*  BY THE USER.
*
*         .
*         .
*         .
*
*
*********************************************************************
*                                                                   *
*  CLEANUP AND TERMINATION PROCESSING                               *
*                                                                   *
*********************************************************************
*
CLEANUP  DS   0H
         L    R1,PPL_PTR             POINT TO PPL IN DYNAMIC WORK AREA
         FREEMAIN RU,LV=L_PPL,A=(1)  FREE THE STORAGE FOR THE PPL
         L    R1,ANSWER_PTR          POINT TO THE ANSWER PLACE
         L    R1,0(0,R1)             POINT TO THE PDL
         IKJRLSA (R1)                FREE STORAGE THAT PARSE ALLOCATED
*                                    FOR THE PDL
         L    R1,ANSWER_PTR          POINT TO THE ANSWER PLACE
         FREEMAIN RU,LV=L_ANSWER,A=(1) FREE THE STORAGE FOR THE
*                                    ANSWER WORD
         L     R5,RETCODE            SAVE RETURN CODE AROUND FREEMAIN
         L     R1,WORK_AREA_GM_PTR   POINT TO MODULE WORK AREA
         FREEMAIN RU,LV=L_WORK_AREA,A=(1)
*                                    FREE THE MODULE WORKAREA
         LR    R1,R13                LOAD PROCESS'S SAVE AREA ADDRESS
         L     R13,B_PTR             CHAIN TO PREVIOUS SAVE AREA
         DROP  R13
  FREEMAIN RU,LV=L_SAVE_AREA,A=(1)   FREE THE MODULE SAVEAREA
         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,R14                 RETURN TO the TMP
```

Figure 6 (Part 3 of 9). A Sample Command Processor

```
*********************************************************************
* POSITCHK - IKJPOSIT VALIDITY CHECKING ROUTINE                     *
*                                                                   *
* IF THE DATA SET NAME HAS A PREFIX OF SYS1 THEN THE VALIDITY       *
* CHECKING ROUTINE RETURNS A CODE OF 4 TO PARSE.  THIS RETURN       *
* CODE INDICATES TO PARSE THAT IT SHOULD ISSUE A MESSAGE TO THE     *
* JOB OUTPUT STREAM.                                                *
*                                                                   *
* IF THE DATA SET PREFIX IS ANYTHING OTHER THAN SYS1, THEN          *
* THIS ROUTINE RETURNS A CODE OF 0 TO PARSE.                        *
*                                                                   *
*********************************************************************
         DROP  R10                 WE WILL REUSE REGISTER 10
POSITCHK DS    0D
         STM   R14,R12,12(R13)     SAVE PARSE'S REGISTERS
         LR    R9,R15
         USING POSITCHK,R9         ESTABLISH ADDRESSABILITY
         LR    R2,R1               SAVE THE VALIDITY CHECK PARAMETER
*                                  LIST PARSE PASSED TO US
         GETMAIN RU,LV=L_SAVE_AREA OBTAIN A DYNAMIC SAVE AREA FOR
*                                  THE POSITCHK ROUTINE
         USING SAVEAREA,R1         AND ESTABLISH ADDRESSABILITY
         ST    R1,8(R13)           PUT THE ADDRESS OF THIS ROUTINE'S
*                                  SAVE AREA INTO PARSE'S SAVE AREA
         ST    R13,B_PTR           PUT THE ADDRESS OF THIS ROUTINE'S
*                                  SAVE AREA INTO ITS OWN SAVE AREA
*                                  FOR CALLING
         LR    R13,R1              LOAD ADDRESS OF GETMAINED AREA
         USING SAVEAREA,R13        AND ESTABLISH ADDRESSABILITY
         L     R10,4(R2)           POINT TO THE COMMAND PROCESSOR'S
*                                  ORIGINAL DYNAMIC WORK AREA
         USING WORKA,R10           DATA AREA SEGMENT BASE REGISTER
         ST    R2,VALCHK_PARAMETER_LIST_PTR
*                                  SAVE THE ADDRESS OF THE VALIDITY
*                                  CHECK PARAMETER LIST
         LM    R1,R3,0(R2)         GET THE ADDRESS OF THE PDE
         STM   R1,R3,VALIDITY_CHECK_PARAMETER_LIST
*                                  SAVE CONTENTS OF PARAMETER LIST
         XC    POSITCHK_RETCODE,POSITCHK_RETCODE
*                                  MAKE SURE WE START WITH A ZERO
*                                  RETURN CODE
```

Figure 6 (Part 4 of 9). A Sample Command Processor

```
         L     R2,PDEADR            GET THE ADDRESS OF THE PDE
         USING DSNAME_PTR,R2        AND ESTABLISH ADDRESSABILITY TO
*                                   OUR MAPPING OF THE PDE
         TM    DSNAME_FLAGS1,QUOTE  IS THE DATA SET NAME IN QUOTES?
         BNO   DSNOK                NO - DATA SET NAME IS OK
         L     R4,DSNAME_PTR        POINT TO THE DSN
         CLC   0(L'SYS1,R4),SYS1    IS HIGH LEVEL DESCRIPTOR SYS1?
         BNE   DSNOK                NO
         L     R5,FOUR              SYS1 IS INVALID.  SET RC=4
         ST    R5,POSITCHK_RETCODE  SAVE THE RETURN CODE
DSNOK    LR    R1,R13               LOAD ROUTINE'S SAVE AREA ADDRESS
         L     R13,B_PTR            CHAIN TO PREVIOUS SAVE AREA
         L     R5,POSITCHK_RETCODE  LOAD THE RETURN CODE
         FREEMAIN RU,LV=L_SAVE_AREA,A=(1)
*                                   FREE THE MODULE WORKAREA
         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,R14                RETURN TO PARSE
         DROP  R9
         DROP  R10
         DROP  R13
*
*********************************************************************
*                                                                   *
* DECLARES FOR CONSTANTS                                            *
*                                                                   *
*********************************************************************
*
PCLADCON      DC   A(PCLDEFS)        ADDRESS OF PCL
FOUR          DC   F'4'              USED TO SET/TEST RETURN CODE
EIGHT         DC   F'8'              USED TO SET/TEST RETURN CODE
TWELVE        DC   F'12'             USED TO SET/TEST RETURN CODE
ERROR         DC   F'12'             USED TO SET/TEST RETURN CODE
SYS1          DC   C'SYS1.'          HIGH-LEVEL DESCRIPTOR
```

Figure 6 (Part 5 of 9). A Sample Command Processor

```
*********************************************************************
*                                                                   *
* PARSE MACROS USED TO DESCRIBE THE COMMAND OPERANDS                *
*                                                                   *
*********************************************************************
*
PCLSTART DS       0H
PCLDEFS  IKJPARM  DSECT=PRDSECT
DSNPCE   IKJPOSIT DSNAME,                                          +
               PROMPT='DATA SET NAME TO BE PROCESSED',             +
               VALIDCK=POSITCHK
ACTPCE   IKJKEYWD DEFAULT='NOACTION'
         IKJNAME  'ACTION'
         IKJNAME  'NOACTION'
         IKJENDP
LPCL     EQU   *-PCLSTART           LENGTH OF THE PCL
*
*********************************************************************
*                                                                   *
* DECLARES FOR DYNAMIC VARIABLES                                    *
*                                                                   *
*********************************************************************
*
WORK_AREA               DSECT
WORKA                   DS  0F      START OF DYNAMIC WORK AREA
WORK_AREA_GM_LENGTH     DS  F       LENGTH OF WORKAREA
WORK_AREA_GM_PTR        DS  F       ADDRESS OF WORKAREA
PPL_LENGTH              DS  F       LENGTH OF PPL
PPL_PTR                 DS  F       ADDRESS OF PPL
ANSWER_LENGTH           DS  F       LENGTH OF PPL ANSWER AREA
ANSWER_PTR              DS  F       ADDRESS OF PPL ANSWER AREA
CPPL_PTR                DS  F       ADDRESS OF THE CPPL FROM TMP
RETCODE                 DS  F       THE RETURN CODE
PARSE_RETCODE           DS  F       THE RETURN CODE FROM PARSE
POSITCHK_RETCODE        DS  F       THE RETURN CODE FROM THE POSITCHK
*                                   VALIDATION EXIT
ECB                     DS  F       CP'S EVENT CONTROL BLOCK
VALCHK_PARAMETER_LIST_PTR DS F      POINTER TO THE VALIDITY CHECK
*                                   PARAMETER LIST
```

Figure 6 (Part 6 of 9). A Sample Command Processor

```
*********************************************************************
*                                                                   *
* MAPPING OF THE THREE WORD VALIDITY CHECK PARAMETER LIST.          *
*                                                                   *
* PARSE PASSES THIS PARAMETER LIST TO THE VALIDITY CHECK ROUTINE,   *
* POSITCHK.  IT CONTAINS THE FOLLOWING INFORMATION:                 *
*       1)  PDEADR - THE ADDRESS OF THE PDE FOR THE DATA SET NAME   *
*       2)  USERWORD - THE ADDRESS OF THE USER WORK AREA THAT THE   *
*                      COMMAND PROCESSOR SUPPLIED TO PARSE IN THE PPL.*
*       3)  VALMSG - THE ADDRESS OF A SECOND LEVEL MESSAGE.  PARSE  *
*                    INITIALIZES THIS FIELD TO X'00'.               *
*                                                                   *
*********************************************************************
*
VALIDITY_CHECK_PARAMETER_LIST DS 0F THE VALIDITY CHECK PARAMETER
*                                   LIST
PDEADR                  DS  F       ADDRESS OF THE PDE FROM PARSE
USERWORD                DS  F       ADDRESS OF THE WORK AREA WE GAVE
*                                   TO PARSE
VALMSG                  DS  F       ADDRESS OF A SECOND LEVEL MESSAGE
*                                   WE CAN GIVE BACK TO PARSE
L_WORK_AREA             EQU *-WORK_AREA
*                                   LENGTH OF DYNAMIC WORK AREA
*
*********************************************************************
*                                                                   *
* DECLARES FOR THE SAVE AREA                                        *
*                                                                   *
*********************************************************************
*
SAVE_AREA               DSECT
SAVEAREA                DS  0CL72   STANDARD SAVE AREA
PLI_LINK                DS  F       UNUSED
B_PTR                   DS  F       BACKWARD SAVE AREA POINTER
F_PTR                   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
L_SAVE_AREA             EQU *-SAVE_AREA
*                                   LENGTH OF SAVE AREA
```

Figure 6 (Part 7 of 9). A Sample Command Processor

```
*********************************************************************
*                                                                   *
*  MAPPING OF THE PDE BUILT BY PARSE TO DESCRIBE A DSNAME OR DSTHING  *
*  OPERAND.                                                         *
*                                                                   *
*********************************************************************
*
DSNAME_DSTHING DSECT                PDE MAPPING FOR THE FOR DSNAME
*                                   OR DSTHING
DSNAME_PTR              DS  F       POINTER TO THE DSNAME
DSNAME_LENGTH_1         DS  H       LENGTH OF THE DATA SET NAME
*                                   EXCLUDING QUOTES
DSNAME_FLAGS1           DS  CL1     FLAGS BYTE
*
*          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
*
                        DS  CL1     RESERVED
DSNAME_MEMBER_PTR       DS  F       POINTER TO THE MEMBER NAME
DSNAME_LENGTH_2         DS  H       LENGTH OF THE MEMBER NAME
*                                   EXCLUDING PARENTHESES
DSNAME_FLAGS2           DS  CL1     FLAGS BYTE
*
*          0... .... THE MEMBER NAME IS NOT PRESENT
*          1... .... THE MEMBER NAME IS PRESENT
*
                        DS  CL1     RESERVED
DSNAME_PASSWORD_PTR     DS  F       POINTER TO THE DATA SET PASSWORD
DSNAME_LENGTH_3         DS  H       LENGTH OF THE PASSWORD
DSNAME_FLAGS3           DS  CL1     FLAGS BYTE
*
*          0... .... THE DATA SET PASSWORD IS NOT PRESENT
*          1... .... THE DATA SET PASSWORD IS PRESENT
*
                        DS  CL1     RESERVED
L_DSNAME_PDE            EQU *-DSNAME_PTR
*
*********************************************************************
*                                                                   *
*  MAPPING OF THE PDE BUILT BY PARSE TO DESCRIBE THE KEYWORD OPERAND  *
*                                                                   *
*********************************************************************
*
KEYWD_PDE               DSECT
KEYWD_NUM               DS  H       CONTAINS THE NUMBER OF THE IKJNAME
*                                   MACRO INSTRUCTION THAT CORRESPONDS
*                                   TO THE OPERAND ENTERED/DEFAULTED
*
L_KEYWD_PDE             EQU *-KEYWD_PDE
```

Figure 6 (Part 8 of 9). A Sample Command Processor

```
*
         IKJPPL                PARSE PARAMETER LIST
L_PPL    EQU   *-PPL
*
         IKJCPPL               COMMAND PROCESSOR PARAMETER LIST
L_CPPL   EQU  *-CPPL
*
ANSWER   DSECT
         DS    F               PARSE ANSWER PLACE.  PARSE PLACES A
*                               POINTER TO THE PDL HERE
L_ANSWER EQU *-ANSWER
*
         CVT   DSECT=YES       CVT MAPPING NEEDED FOR CALLTSSR MACRO
*
*********************************************************************
*                                                                   *
*   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                    BASE REGISTER
R12      EQU   12
R13      EQU   13                    DATA REGISTER
R14      EQU   14                    RETURN ADDRESS
R15      EQU   15                    RETURN CODE
QUOTE    EQU   X'40'                 FULLY-QUALIFIED DATA SET NAME
         END   PROCESS
```

Figure 6 (Part 9 of 9). A Sample Command Processor

Chapter 5. Communicating with the User through the Job Stream

Your command processor may need to obtain data from the input stream or write messages or data to the output data set.

This chapter provides an overview of how to issue messages, perform I/O and change the source of input. For additional information on the macros and services discussed in this chapter, see "Part II: TSO Programming Services" on page 41.

Issuing Messages

TSO supports three classes of messages:

- Prompting messages
- Mode messages
- Informational messages.

Prompting messages begin with "MISSING" and indicate that a required operand is missing. For example, the parse service routine issues prompting messages when the user has specified an incorrect operand or when a required operand is missing.

Mode messages are issued to the output data set to indicate whether the TMP or a command processor is in control. When a mode message is issued, a new command or subcommand is obtained from the input stream. For example, the *READY* message issued by the TMP is a mode message.

If you have chosen to implement subcommands, your command processor should issue a mode message to indicate that the command is in control, and to obtain a subcommand from the input stream.

Informational messages are issued to the output data set to notify the user of the status of the command being executed. For example, informational messages should be issued if your command processor encounters an error and must terminate.

Message Levels

Messages that are issued to the output data set should usually have *second level messages* associated with them. Second level messages provide additional explanation of the initial message and follow the initial message in the output data set.

Prompting messages can have any number of second level messages. However, informational messages can have only one second level message associated with them. Mode messages cannot have second level messages.

Using the I/O Service Routines to Handle Messages

Your command processor can use the I/O service routines provided by TSO to obtain input and issue messages.

Use the PUTLINE service routine, which writes a line of data to the output data set, to issue prompting and informational messages. Use the PUTGET service routine, which writes a line of data to the output data set and obtains a line of input, to issue mode messages.

When issuing prompting or informational messages, you can also use PUTLINE to place inserts into message text and chain second level messages.

When PUTGET returns a line of data from the input stream, this data is placed in a buffer that resides in subpool 1 and is owned by your command processor. Although the buffers returned by PUTGET are automatically freed when your code relinquishes control, you can use the FREEMAIN macro instruction to free these buffers.

Using the TSO Message Issuer Routine (IKJEFF02)

If your command processor issues messages with numerous inserts, you should use the TSO message issuer service routine (IKJEFF02) instead of PUTLINE and PUTGET. Using IKJEFF02 has several advantages:

- It simplifies the issuing of messages with inserts because the same parameter list can be used to issue any message.
- This service makes it convenient to place all messages for a command in a single CSECT. This is important when you have to modify message texts.
- It provides support for second level messages that are associated with informational messages.

Using Generalized Routines for Issuing Messages

If your command processor invokes TSO services or system services, you should issue informational messages to notify the user if error conditions occur.

You can use DAIRFAIL to analyze return codes from dynamic allocation (SVC 99) and the TSO dynamic allocation interface routine (DAIR), and to issue error messages when appropriate. Use the GNRLFAIL/VSAMFAIL routine to issue error messages for VSAM macro failures, subsystem request failures, parse service routine failures, PUTLINE failures, and ABEND codes.

Performing I/O

Your command processor may need to write lines of data to the output data set or obtain data from the input stream. This topic discusses how to perform I/O for data other than messages and subcommand requests.

There are several methods that you can use to perform I/O.

- **The BSAM or QSAM macro instructions** provide I/O support for programs that run under TSO. For example, you can use the PUT or WRITE macro instructions to write data to the output data set and you can use the GET or READ macro instructions to obtain data from the input stream.

  The major benefit of using BSAM or QSAM to process I/O is that these access methods are not TSO dependent. Therefore, you can incorporate code from existing routines that use BSAM or QSAM into your command processor without having to modify the macro instructions.

- **The GETLINE and PUTLINE service routines** provide the ability to obtain data from the input stream and write data to the output data set, respectively. Use the GETLINE and PUTLINE macro instructions to invoke these I/O service routines.

  When GETLINE returns a line of input, this data is placed in a buffer that resides in subpool 1 and is owned by your command processor. Although the buffers returned by GETLINE are automatically freed when your code relinquishes control, you can use the FREEMAIN macro instruction to free these buffers.

  Use the PUTLINE macro instruction with the DATA operand to write one or more lines of data to the output data set.

Changing Your Command Processor's Source of Input

TSO maintains an internal pushdown list that determines the source of input. This pushdown list, or *stack*, is used and maintained by the TSO I/O service routines (STACK, GETLINE, PUTLINE and PUTGET).

The top element of the stack indicates the currently active input source. The TMP initializes this stack by creating the first element, which indicates that the input stream is the current source of input. Therefore, when your command processor receives control, the current source of input is the input stream. When you use the GETLINE, PUTLINE or PUTGET macro instructions, all input is read from the input steam and all output is written to the output data set.

You may want to obtain input from a source other than the input stream, such as another data set containing records to be processed. TSO allows an *in-storage list* to be used as the source of input. An in-storage list can be either a command procedure (list of commands) or a source data set. Use the STACK service routine in your command processor to change the source of input by either adding or removing an element from the input stack. However, your command processor cannot change or delete the first element.

Chapter 6. Passing Control to Subcommand Processors

If you have chosen to implement subcommands, your command processor must be able to recognize a subcommand name specified in the input stream and pass control to the requested subcommand processor. This chapter outlines the steps you must follow to support subcommands.

Command scan, the PUTGET service routine and the parse service routine are discussed in this chapter; refer to "Part II: TSO Programming Services" on page 41 for more information on these services.

To recognize a subcommand name and pass control to the subcommand processor, follow these steps:

1. Use the PUTGET service routine to issue a mode message and retrieve a line of input that may contain a subcommand.
2. Use the command scan service routine to determine if the user has entered a valid subcommand name.
3. Use the ATTACH macro instruction to pass control to the subcommand processor.
4. Use the DETACH macro instruction to release the subcommand processor when it has completed.

Step 1. Issuing a Mode Message and Retrieving an Input Line

Use the PUTGET service routine to issue a mode message to indicate which command is in control, and to return a line of input.

GENERAL-USE PROGRAMMING INTERFACE

When PUTGET returns a line of data from the input stream, it places this data in an input buffer that is owned by your command processor. Figure 7 shows the format of the input buffer returned by the PUTGET service routine.

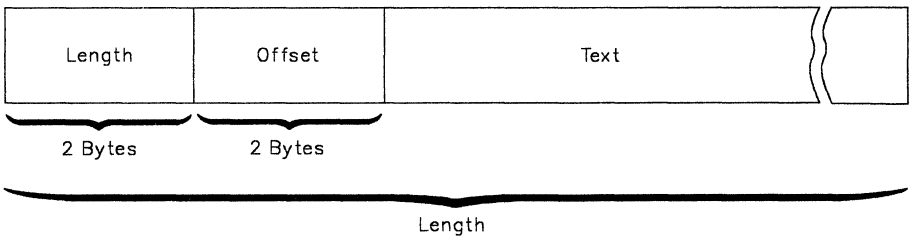


Figure 7. Format of the Input Buffer

The two-byte length field contains the length of the returned input line plus the length of the four-byte header. The two-byte offset field is always set to zero on return from the PUTGET service routine.

End of GENERAL-USE PROGRAMMING INTERFACE

Step 2. Validating the Subcommand Name

Use the command scan service routine to determine whether a syntactically valid subcommand name is present in the input buffer (command buffer). Command scan searches the input buffer for a subcommand name, checks the syntax of the name, and updates the offset field in the input buffer. If a valid subcommand name is found, command scan resets the offset field in the input buffer to the number of text bytes preceding the first subcommand operand, if any are present. For example, if the user enters

```
SUBCMD OPERAND1 OPERAND2
```

the offset field would be set to 7, the number of bytes that precede OPERAND1 in the input buffer.

Although command scan recognizes comments present in the input buffer, it skips over them without processing them. Comments, which are indicated by the delimiters /* and */, are not removed from the input buffer.

When your command processor passes control to command scan, it must pass a parameter list that contains pointers to control blocks and data areas that are needed by command scan. Addresses needed to access the input buffer and the output area filled in by command scan are included in this parameter list.

When command scan returns control to your command processor, check the return code in register 15. If the return code is zero, check the flag field in the output area to determine whether a syntactically valid subcommand name is present. Use the pointer to the subcommand name and the length of the name returned in the output area when you pass control to the appropriate subcommand processor.

Step 3. Passing Control to the Subcommand Processor

After determining that the user has specified a valid subcommand name, use the ATTACH macro instruction to pass control to the requested subcommand processor. Depending upon the function and complexity of the command processor and the subcommand processor, you may need to specify the ESTAI operand on the ATTACH macro to provide an error handling routine that receives control if the subcommand processor abnormally terminates. For information on error handling, see Chapter 7, "Processing Abnormal Terminations" on page 33. For information on the ATTACH macro instruction, see *Application Development Macro Reference*.

Subcommand processors are similar to command processors in many ways, including syntax and the way they receive control. When your command processor attaches the subcommand processor, pass a pointer to a command processor parameter list (CPPL) in register 1. The CPPLBUF field in the CPPL must contain the address of the input buffer containing the subcommand. (The CPPL is described in Figure 13 on page 51.)

Writing a Subcommand Processor

When you write a subcommand processor, follow steps that are similar to the steps you followed to write your command processor. This procedure is listed below:

1. Access the command processor parameter list (CPPL).
2. Validate any operands specified with the subcommand using the parse service routine.
3. Communicate with the user through the job stream.
4. Perform the function of the subcommand according to any operands the user specified.
5. Intercept and process abnormal terminations.
6. Set the return code in register 15 and return to the command processor.

These steps are discussed in more detail in Chapter 3, "What You Need to Do to Write a Command Processor" on page 11.

Step 4. Releasing the Subcommand Processor

When the subcommand processor has completed processing and returned control to your command processor, use the DETACH macro instruction to release it. For information on the DETACH macro instruction, see *Application Development Macro Reference.*

Chapter 7. Processing Abnormal Terminations

Depending upon the function and complexity of your command processor, you may need to provide error handling routines to process abnormal terminations (ABENDS). This chapter describes the criteria you should consider to determine whether special processing is needed for error recovery. It also provides guidelines for writing error handling routines.

Error Handling Routines

When an abnormal termination occurs, your command processor must be able to provide sufficient recovery to insure that the error condition does not cause the abnormal termination of the job step. Error handling routines give your command processor the ability to intercept an ABEND and allow it to clean up, bypass the problem, and if possible, attempt to retry execution.

A command processor must be able to recognize and respond to two types of abnormal terminations:

1. The command processor or a program at the same task level, such as command scan or the parse service routine, is terminating abnormally.
2. An attached subtask, such as a subcommand processor, is terminating abnormally.

ESTAE and ESTAI Exit Routines

Two types of error handling routines are used in writing command processors: *ESTAE exits* and *ESTAI exits*. An *ESTAE exit* is established by issuing the ESTAE macro instruction. The function of an ESTAE exit is to intercept abnormal terminations that occur at the current task level. The FESTAE macro instruction can be used to establish an ESTAE exit for authorized command processors.

An *ESTAI exit* processes abnormal terminations that occur at the daughter task level. ESTAI exits are established by using the ATTACH macro with the ESTAI operand.

See *SPL: Application Development Macro Reference* for information on the ESTAE and FESTAE macro instructions. See *Application Development Macro Reference* for a discussion of the ESTAI operand of the ATTACH macro instruction and for information on ESTAE and ESTAI exit routines.

When are Error Handling Routines Needed?

Not all command processors require special error handling. In many cases, the error handling routine provided by the TMP is sufficient. However, if your command processor falls into one of the following categories, you should provide an ESTAE exit routine to handle abnormal terminations at the command processor's task level:

- Command processors that process subcommands
- Command processors that request system resources that are not freed by ABEND or DETACH
- Command processors that process lists. Recovery processing is necessary to allow processing of other elements in the list if a failure occurs while processing one element.
- Command processors that use the STACK service routine to change the source of input. The error handling routine should issue the STACK macro instruction to clear the input stack before returning to the TMP.

In addition, if your command processor attaches subcommands, it should also provide an ESTAI exit to intercept abnormal terminations at the subcommand processor's task level. ESTAE and ESTAI exit routines should be used in such a way that the command processor gets control if a subcommand abnormally terminates.

Simple command or subcommand processors should not issue an ESTAE or an ESTAI if the ESTAI exit provided by the terminal monitor program (TMP) or the calling command processor, respectively, provides adequate processing.

Figure 8 on page 35 shows the relationship between the command processor, subcommand processor, and the error handling routines.

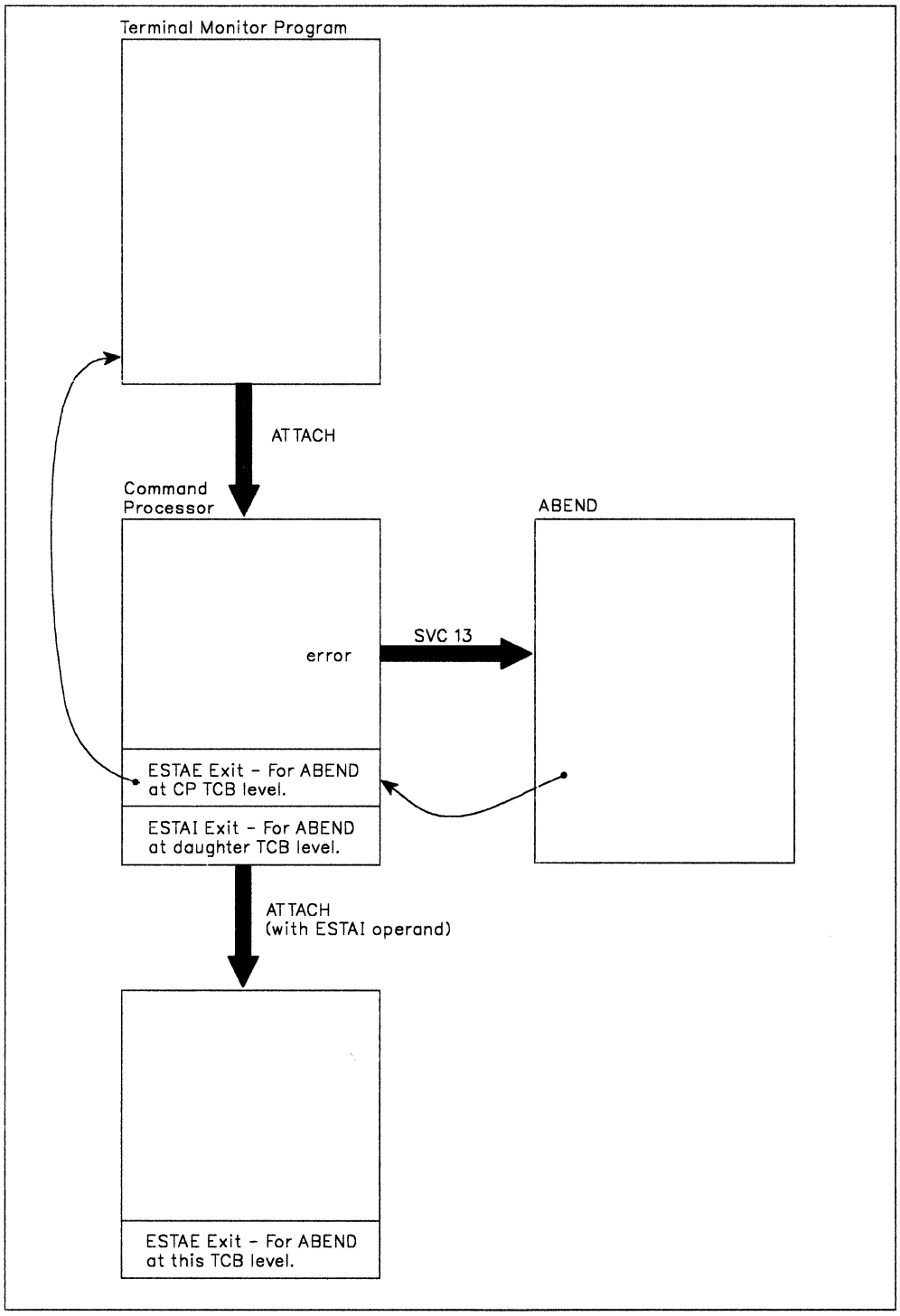


Figure 8. ABEND, ESTAI, ESTAE Relationship

Guidelines for Writing ESTAE and ESTAI Exit Routines

GENERAL-USE PROGRAMMING INTERFACE

When you write ESTAE and ESTAI exit routines, observe the following guidelines:

1. Issue an ESTAE macro instruction as early in your command processor as possible.
2. The error handling exit routine should issue a diagnostic error message of the form:

   ```
   1st level  { command-name    }  ENDED DUE TO ERROR+
              { subcommand-name }

   2nd level    COMPLETION CODE IS xxxx
   ```

   Obtain the name supplied in the first level message from the environment control table (ECT). The code supplied in the second level message is the completion code passed to the ESTAE or ESTAI exit from ABEND. You can use the GNRLFAIL service routine to issue the diagnostic error message, although it requires additional storage space (see guideline number 5).

   The error handling routine should issue these messages so that the original cause of abnormal termination is recorded, in case the error handling routine itself terminates abnormally before diagnosing the error.

   When an ABEND is intercepted, the command processor ESTAE exit routine must determine whether retry is to be attempted. If so, the exit routine must issue the diagnostic message and return, indicating by a return code that an ESTAE retry routine is available. If a retry is not to be attempted, the exit routine must return, and indicate with a return code that no retry is to be attempted. The TMP, which receives control after the command processor's ESTAE exit routine, issues the diagnostic message. For a description of the return codes from ESTAE exit routines and their meanings, see *Application Development Macro Reference*.
3. The ESTAE or ESTAI routine that receives control from ABEND must perform all necessary steps to provide system cleanup.
4. The error handling exit routine should attempt to retry program execution when possible. If the command processor can circumvent or correct the condition that caused the error, the error handling routine should attempt to retry execution. In other cases, however, RETRY has no function and the command processor ESTAE exit should not specify the RETRY option.
5. Storage might not be available when the ESTAE or ESTAI routine receives control. Any storage the routine requires should be acquired before the routine receives control, and be passed to it.

End of GENERAL-USE PROGRAMMING INTERFACE

Chapter 8. Installing a Command Processor

After you have completed writing your command processor, you must install it in a way that makes the command available for you, and possibly other users, to execute. This chapter describes the methods that you can use to add your new command processor to TSO.

As part of the installation process, use the linkage editor to convert the object modules that result from assembling your command processor into a load module that is suitable for execution. The particular data set that contains the load module is determined by the method that you choose to install your command processor. These methods are described in the topics that follow.

Using a Private Step Library

If you are an unauthorized user, you can define a private step library using the STEPLIB DD statement in the JCL you use to execute the command processor. This step library is a partitioned data set that contains the command. Use the linkage editor to enter your command processor as a member of the partitioned data set.

If you are an authorized user and you intend to make your command available to a large number of TSO users, this method is not recommended because of the TSO performance degradation that results from the additional search time required for each command. However, using a STEPLIB is advantageous if you want to make your command available to only selected TSO users. It is also a useful method to temporarily install your command processor while you are testing and refining your code.

Placing Your Command Processor in SYS1.CMDLIB

If you are an authorized user, you can use the linkage editor to enter your command processor as a member of the partitioned data set SYS1.CMDLIB. Placing your command processor in SYS1.CMDLIB makes it available to all TSO users.

Creating Your Own Command Library

If you are an authorized user, you can create your own command library and concatenate it to the SYS1.CMDLIB data set. To do this, create new statements in the link list (LNKLST00 or LNKLSTxx) in SYS1.PARMLIB. Use the linkage editor to enter your command processor as a member of the command library. This method makes your command available to all TSO users.

Chapter 9. Executing a Command Processor

After you have installed your command processor, you are ready to execute it. This chapter describes the JCL statements you must submit to the operating system to execute a command processor. For additional information on writing JCL, refer to *JCL Reference*.

Writing JCL for Command Execution

To execute a command processor, write JCL statements that execute the terminal monitor program (TMP). The TMP provides an interface between the user, command processors, and the TSO control program. It obtains commands, gives control to command processors, and monitors their execution. The TMP is attached as APF-authorized and executes in either supervisor state or problem program mode.

Figure 9 illustrates the JCL statements needed to execute the TMP.

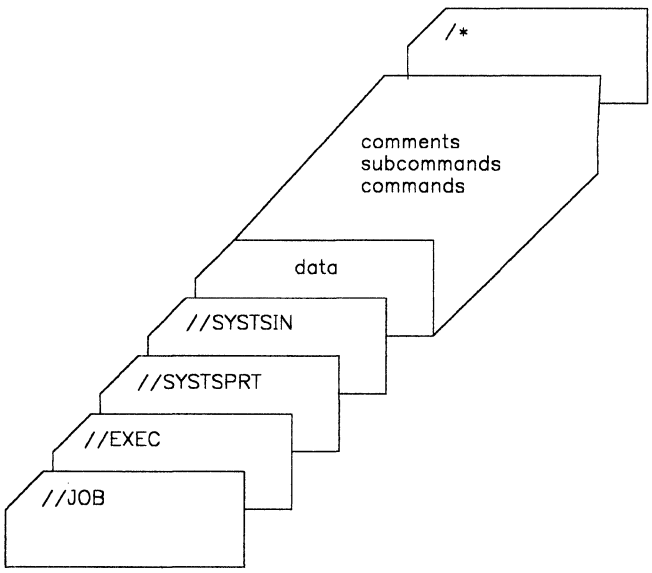


Figure 9. JCL Needed to Process Commands

The JCL required to execute the TMP includes the following:

1. A JOB statement, including a jobname and operands that specify the processing options.
2. An EXEC statement that specifies IKJEFT01 (the TMP) as the program to be executed. The format is:

   ```
   //stepname EXEC PGM=IKJEFT01,DYNAMNBR=nn,PARM='command'
   ```

If you are executing commands that dynamically allocate data sets, specify the DYNAMNBR parameter. This parameter indicates the number of allocations of data sets or ddnames that can be used at one time per job step. The limit for the DYNAMNBR parameter is system-dependent. Refer to *JCL Reference* for more information.

You may use the PARM parameter to specify the first (or only) command to be executed. This parameter is used most often when you execute one command in the step.

3. A SYSTSPRT DD statement that controls output from your job. This DD statement can refer to a system printer or to a sequential or partitioned data set. If the data set is partitioned, you must specify the member name on the DD statement as DSN=pdsname(membername).

   Messages issued by programs using the TSO I/O service routines are written to the data set indicated by the SYSTSPRT DD statement.

4. A SYSTSIN DD statement that controls input to your job. Use this statement to indicate which commands and subcommands are to be executed.

   You can specify the input data directly following the SYSTSIN DD statement, or you can refer to a sequential or partitioned data set. If the data set is partitioned, you must specify the member name on the DD statement as DSN=pdsname(membername). You cannot refer to concatenated data sets on the SYSTSIN DD statement.

   For each command to be executed, specify the name of the command followed by the operands that are needed for the function you want performed. Each command or subcommand must begin on a separate input line.

   Programs that use the TSO I/O service routines to obtain input receive their input from the data set indicated by the SYSTSIN DD statement.

Handling Error Conditions

The return code from a job step that executes TSO commands is the return code of the last command executed.

An ABEND code is issued when either the TMP or a command processor terminate abnormally. In this situation, TSO processing stops and the remainder of the commands in SYSTSIN are ignored. To obtain a dump, specify a SYSUDUMP or SYSABEND DD statement in your JCL. For information on specific ABEND codes, refer to *Message Library: System Codes*.

Part II: TSO Programming Services

TSO provides services that perform a wide range of programming functions. You can use these services in system or application programs to perform the following tasks:

- **Invoking TSO Service Routines**

  To pass control to certain TSO service routines, use the CALLTSSR macro instruction. See Chapter 11, "Invoking TSO Service Routines with the CALLTSSR Macro Instruction" on page 53.

- **Checking the Syntax of Subcommand Names**

  Use the command scan service routine in your command processors to validate a subcommand name. See Chapter 12, "Verifying Subcommand Names with the Command Scan Service Routine" on page 55.

- **Checking the Syntax of Command and Subcommand Operands**

  Use the parse service routine to validate command or subcommand operands. See Chapter 13, "Verifying Command and Subcommand Operands with the Parse Service Routine" on page 63.

- **Communicating with the User through the Job Stream**

  TSO provides several services to aid you in processing I/O and issuing messages.

  - You can use the TSO 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. The I/O service routines can be used to issue messages to the output data set. See Chapter 14, "Using the TSO I/O Service Routines" on page 153.
  - Your command processors can use the TSO message issuer routine (IKJEFF02) to issue messages to the output data set. See Chapter 15, "Using the TSO Message Handling Routine (IKJEFF02)" on page 201.

- **Processing Data Sets**

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

  - **Allocating, Concatenating and Freeing Data Sets:** TSO 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, using SVC 99.* For a complete discussion of dynamic allocation, see *SPL: Application Development Guide*. DAIR is discussed in Chapter 16, "Using the Dynamic Allocation Interface Routine (DAIR)" on page 207.
  - **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 ID. See Chapter 19, "Using IKJEHCIR to Retrieve System Catalog Information" on page 239.

- **Analyzing Return Codes**

  Use the DAIRFAIL routine (IKJEFF18) to analyze return codes from dynamic allocation (SVC 99) or DAIR and issue appropriate error messages. See Chapter 17, "Using the DAIRFAIL Routine (IKJEFF18)" on page 231.

  Use the GNRLFAIL/VSAMFAIL routine (IKJEFF19) to analyze VSAM macro instruction failures, subsystem request (SSREQ) failures, parse service routine or PUTLINE failures, and ABEND codes, and issue an appropriate error message. See Chapter 18, "Analyzing Error Conditions with the GNRLFAIL/VSAMFAIL Routine (IKJEFF19)" on page 235.

Coding the Macro Instructions

The following paragraphs describe the notation used to define the macro syntax in Part II of 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.

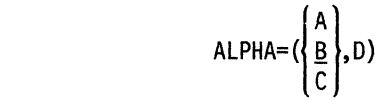
5. An underscore indicates a default option. If an underscored alternative is selected, it need not be written in the actual macro instruction.

   Example: The representation

   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.

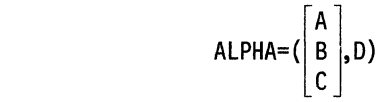
   Example: The representation

   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

   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 10. Considerations for Using TSO Services

This chapter discusses considerations for using the services documented in this manual. Topics include:

- Programming considerations for MVS/ESA
- Interfacing with the TSO service routines.

MVS/ESA Considerations

This topic discusses considerations for MVS/ESA that you should be aware of when writing a command processor or using the services documented in this book. You must be familiar with the publications that describe comprehensive programming considerations for MVS/ESA and with the publications that describe the system 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.

General Interface Considerations

The interfaces described in this topic reflect what is *possible* for programs to do on an MVS/ESA system. When determining the attributes and linkage conventions for a program, analyze the program's individual interfaces and its overall interactions with other programs. This topic provides general guidelines for making these determinations.

You must consider address space control modes, addressing modes and program residency when determining linkage conventions. See "Interface Considerations for the TSO Service Routines" on page 47 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
- Address space control (ASC) mode attributes
- AMODE and RMODE attributes

The following discussion provides a general description of ASC mode, AMODE and RMODE attributes. For a detailed description of ASC mode considerations, refer to *SPL: Application Development — Extended Addressability.* For a detailed discussion of 31-bit addressing, refer to *SPL: Application Development — 31-Bit Addressing.*

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.

All service routines supplied by TSO execute in primary mode.

AMODE = 24, RMODE = 24

Programs with these attributes must receive control in 24-bit addressing mode, and are loaded below 16 megabytes 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 TSO-supplied command processors have these attributes and are loaded below 16 megabytes 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. Although a program with the AMODE = ANY attribute might have to switch addressing modes for certain processing, the program must switch back to the addressing mode in which it received control before returning to its 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 megabytes in virtual storage; the particular interfaces should be analyzed to determine where input can reside. However, a program should meet certain criteria in order to be assigned the AMODE = ANY attribute. Refer to *SPL: Application Development — 31-Bit Addressing* for a description of the criteria.

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 megabytes 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).

Refer to *SPL: Application Development — 31-Bit Addressing* for more information on the AMODE = 31 attribute.

Interface Considerations for the TSO Service Routines

All TSO service routines documented in this book must receive control in primary address space control mode. These service routines execute and 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.

The command processor parameter list (CPPL), which contains certain addresses required as input to the TSO service routines, resides below 16 megabytes in virtual storage. Refer to "Interfacing with the TSO Service Routines" on page 50 for more information on the CPPL.

Figure 10 shows the interface considerations for the TSO service routines.

Figure 10. Interface Considerations for TSO Service Routines

Service Routine	Entry Point Name	Interface Considerations
Catalog information routine	IKJEHCIR	This routine can be invoked in either 24- or 31-bit addressing mode, but all input passed to this routine must reside below 16 megabytes in virtual storage. This routine executes in 24-bit addressing mode and returns control in the same addressing mode in which it is invoked.
Dynamic allocation interface routine	IKJDAIR	This service routine can be invoked in either 24- or 31-bit addressing mode. When invoked in 31-bit addressing mode, this routine can be passed input that resides above 16 megabytes in virtual storage. This routine executes and returns control in the same addressing mode in which it is invoked.
TSO message issuer routine DAIRFAIL GNRLFAIL/VSAMFAIL GETLINE service routine Parse service routine PUTGET service routine PUTLINE service routine Command scan service routine STACK service routine	IKJEFF02 IKJEFF18 IKJEFF19 IKJGETL IKJPARS IKJPTGT IKJPUTL IKJSCAN IKJSTCK	These service routines must receive control in 24-bit addressing mode. All input passed to them must reside below 16 megabytes in virtual storage. These routines execute and return control in 24-bit addressing mode.

Invoking the TSO Service Routines

You can use either the LINK, LOAD or CALLTSSR macro instructions to pass control to the TSO service routines. The CALLTSSR macro is used for certain TSO routines only. It is described in Chapter 11, "Invoking TSO Service Routines with the CALLTSSR Macro Instruction" on page 53.

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.

For a program executing in 31-bit addressing mode, use the LINK macro instruction to invoke those service routine that must receive control in 24-bit addressing mode. In this case, the LINK macro switches to 24-bit addressing mode on behalf of the invoking program. A program that resides above 16 megabytes in virtual storage *must* use the LINK macro instruction to invoke those service routines that must receive control in 24-bit addressing mode.

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, the BASSM or BSM instruction can be used to set the appropriate addressing mode. If you use BASSM or BSM, you should ensure that the invoked program can return successfully.

Summary of Macro Interfaces

Figure 11 shows the interface rules for using the macros discussed in this manual.

In Figure 11, 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.

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	
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
IKJTSMSG	-	-	P	P
PUTGET	X	X	I,P	
PUTLINE	X	X	I,P	
STACK	X	X	I,P	

Figure 11. Interface Rules for Using Macro Instructions

CALLTSSR

The CALLTSSR macro instruction can be issued in either 24-bit or 31-bit addressing mode. See Chapter 11, "Invoking TSO Service Routines with the CALLTSSR Macro Instruction" on page 53 for more information on issuing the CALLTSSR macro.

GETLINE, PUTGET, PUTLINE, STACK

The GETLINE, PUTGET, PUTLINE, and STACK macros must be issued in 24-bit addressing mode. Input passed to these routines must reside below 16 megabytes in virtual storage.

IKJTSMSG

The IKJTSMSG macro must be issued by a program loaded below 16 megabytes in virtual storage. Refer to Chapter 15, "Using the TSO Message Handling Routine (IKJEFF02)" on page 201 for a description of the of the input parameter list for IKJEFF02.

Parse Macros (IKJENDP through IKJTERM)

The parameter list passed to the parse service routine must reside below 16 megabytes in virtual storage. As a result, the parse macro instructions that generate input to parse must be issued by a program loaded below 16 megabytes in virtual storage. See Figure 11 for a list of the parse macros and their linkage requirements. The IKJRLSA parse macro must be issued in 24-bit addressing mode mode.

End of GENERAL-USE PROGRAMMING INTERFACE

Interfacing with the TSO Service Routines

When you invoke the TSO service routines from a command processor, you must pass certain addresses contained in the command processor parameter list (CPPL).

GENERAL-USE PROGRAMMING INTERFACE

The Command Processor Parameter List

When the terminal monitor program attaches a command processor, register 1 contains a pointer to a command processor parameter list (CPPL) containing addresses required by the command processor. The CPPL is a four-word parameter list that is located in subpool 1.

End of GENERAL-USE PROGRAMMING INTERFACE

The control block interface between the TMP and an attached command processor is shown in Figure 12.

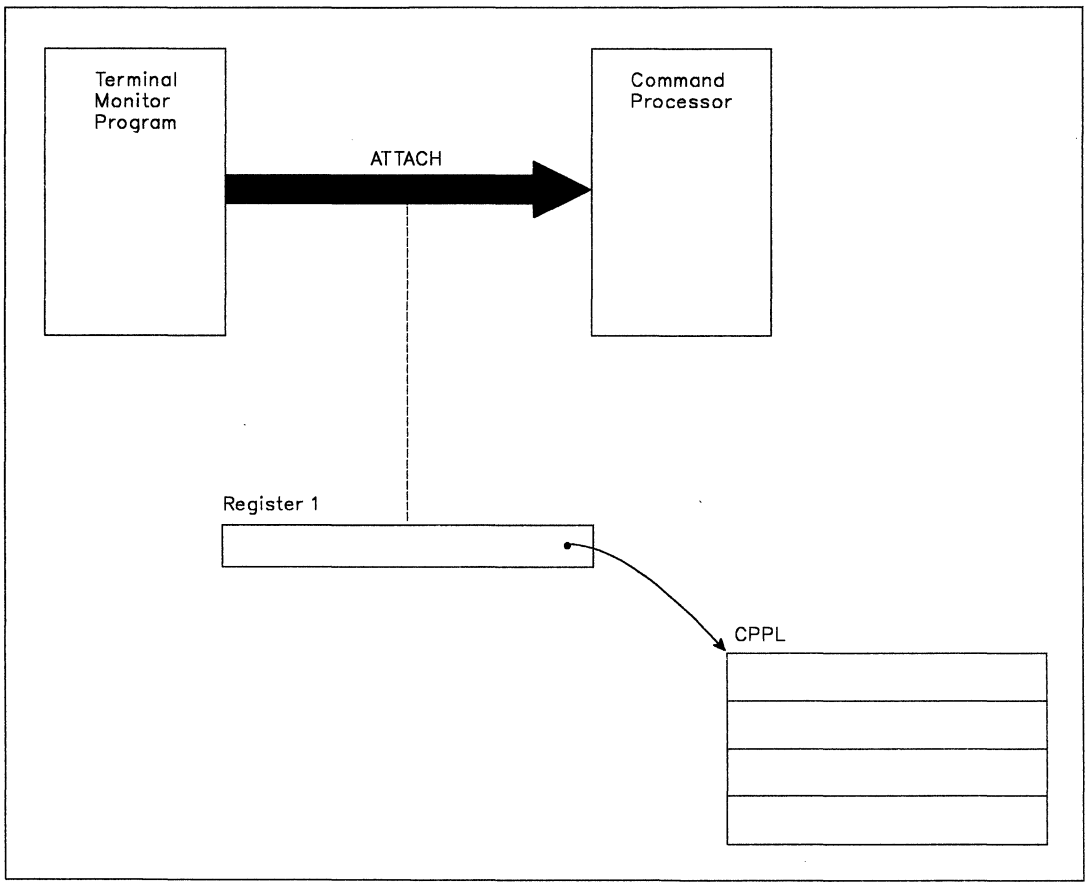


Figure 12. Control Block Interface between the TMP and a Command Processor

GENERAL-USE PROGRAMMING INTERFACE

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. The use of the DSECT is recommended because it protects the command processor from any changes to the CPPL. Figure 13 describes the contents of the CPPL.

Figure 13. The Command Processor Parameter List (CPPL)

Number of Bytes	Field	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). The ECT is built by the TMP during its initialization process; it is used by the TSO service routines and by some TSO commands. Use the IKJECT mapping macro, which is provided in SYS1.MACLIB, to map the fields in the ECT.

End of GENERAL-USE PROGRAMMING INTERFACE

Services that Access Data in the CPPL

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

IKJDAIR Dynamic allocation interface routine
IKJEFF02 TSO message issuer routine
IKJEFF18 DAIRFAIL
IKJEFF19 GNRLFAIL/VSAMFAIL
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

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

Chapter 11. Invoking TSO Service Routines with the CALLTSSR Macro Instruction

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

GENERAL-USE PROGRAMMING INTERFACE

When to Use the CALLTSSR Macro Instruction

You can use the CALLTSSR macro instruction to generate a branch to certain TSO service routines residing in the link pack area. If the routine does not reside in the link pack area, CALLTSSR generates a LINK macro instruction. 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 service routines only:

IKJDAIR	Dynamic allocation interface routine
IKJEFF02	TSO message issuer routine
IKJEHCIR	Catalog information routine
IKJPARS	Parse service routine
IKJSCAN	Command scan service routine

Notes:

1. A module that uses the CALLTSSR macro instruction must include the CVT mapping macro.
2. IKJEFF02, IKJPARS and IKJSCAN must receive control in 24-bit addressing mode. Therefore, if these routines reside in the link pack area, your program must invoke the CALLTSSR macro instruction in 24-bit addressing mode.

Syntax and Operands

Figure 14 shows the execute form of the CALLTSSR macro instruction. There is no list form. Each operand is explained following the figure.

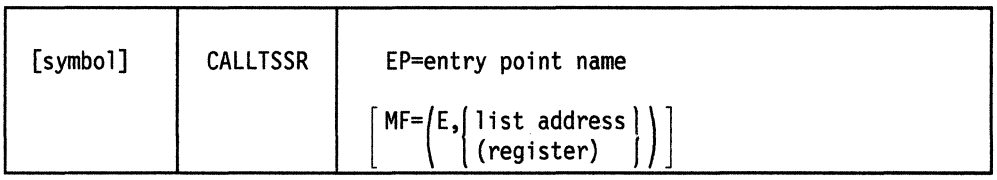


Figure 14. The CALLTSSR Macro Instruction

EP = entry point name
Specifies one of the following names: IKJDAIR, IKJEFF02, IKJEHCIR, IKJPARS, or IKJSCAN.

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

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

End of GENERAL-USE PROGRAMMING INTERFACE

Example

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 12. Verifying Subcommand Names with the Command Scan Service Routine

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.

GENERAL-USE PROGRAMMING INTERFACE

Figure 15 shows the format of the command buffer.

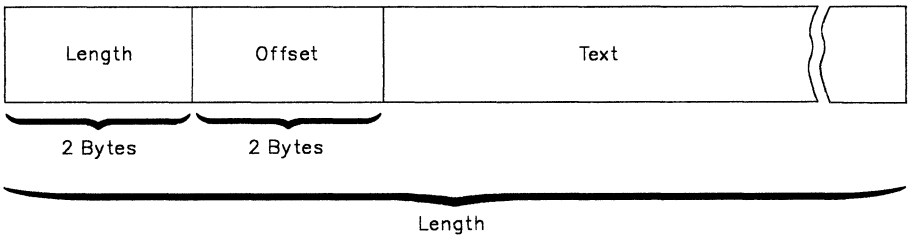


Figure 15. Format of the Command Buffer

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.

End of GENERAL-USE PROGRAMMING INTERFACE

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 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 an alphabetic or a national character.
- The remaining characters must be alphameric.
- The length of the subcommand name must not exceed eight characters.
- The command delimiter must be a separator character.

It is recommended that the name include one or more numerals. Since no IBM-supplied command names include numerals, this insures that your subcommand name will be unique.

Figure 16 shows the various character types recognized by the command scan service routine. Unless otherwise indicated, alphameric characters are:

Alphabetic (A-Z)
Numeric (0-9)
National ($, #, @)

Character		Character Type						
		Separator	National	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	@		X					
	a-z			X				
	A-Z			X				
	0-9				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

Figure 16. Character Types Recognized by Command Scan

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 must receive control in 24-bit addressing mode. If your program uses the CALLTSSR macro instruction to invoke IKJSCAN, and IKJSCAN resides in the link pack area, your program must issue the CALLTSSR macro instruction in 24-bit addressing mode. However, if IKJSCAN does *not* reside in the link pack area, your program can issue the CALLTSSR macro instruction in either 24- or 31- bit addressing mode.

All input passed to IKJSCAN must reside below 16 megabytes in virtual storage.

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 17 shows the parameter list structure that your command processor must create as input to the command scan service routine.

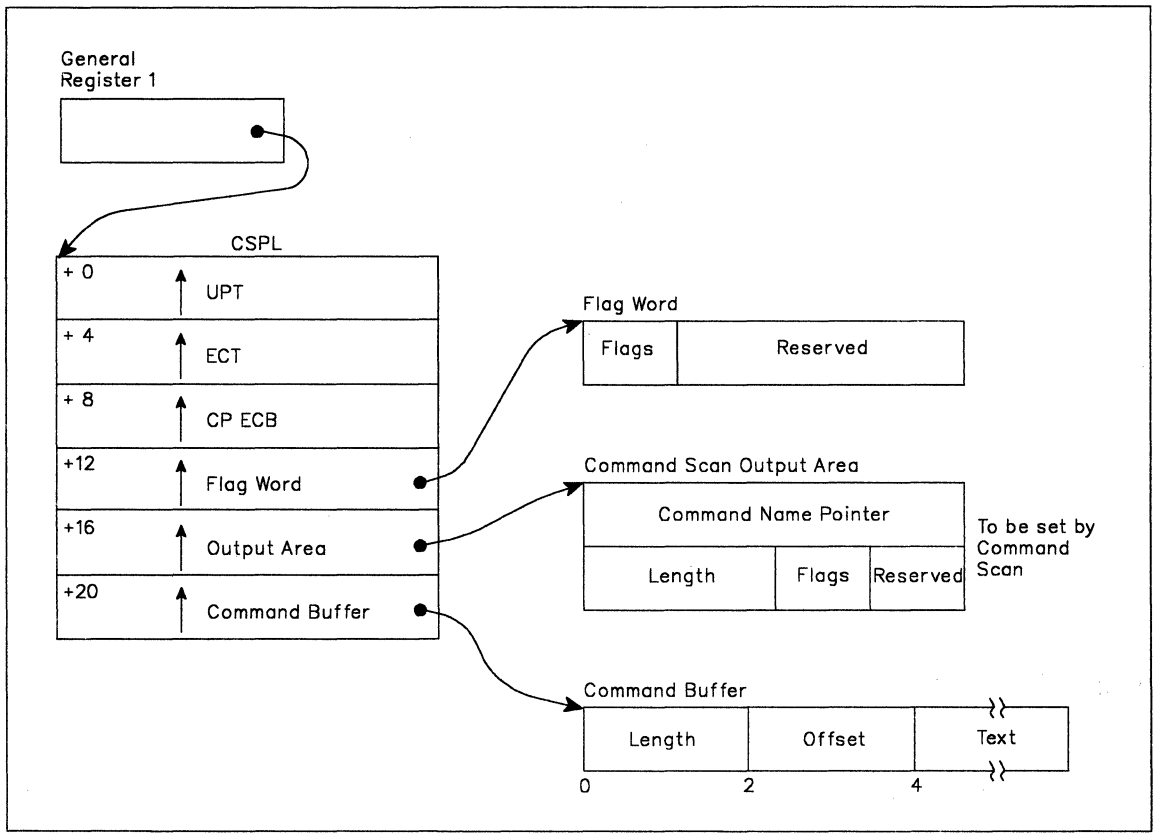


Figure 17. The Parameter List Structure Passed to Command Scan

Use the IKJCSPL DSECT, which is provided in SYS1.MACLIB, to map the fields in the CSPL. Figure 18 shows the format of the command scan parameter list.

Figure 18. The Command Scan Parameter List

Number of Bytes	Field	Contents or Meaning
4	CSPLUPT	The address of the user profile table. This address is passed to a command processor by the TMP in the CPPL.
4	CSPLECT	The address of the environment control table. This address is passed to a command processor by the TMP in the CPPL.
4	CSPLECB	The address of the command processor's event control block. (Required if command scan is called by a command processor to scan a subcommand; zeros if command scan is called by the TMP.)
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; the last three bytes are reserved.
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 17.)
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 only uses the first byte of the field; the remaining three bytes are reserved.

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. Figure 19 shows the format of the command scan output area.

Figure 19. The Command Scan Output Area

Number of Bytes	Field	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 Figure 20.
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.

End of GENERAL-USE PROGRAMMING INTERFACE

Operation of the Command Scan Service Routine

If you set the flags field in the flag word to X'80' (to indicate that the command name is not to be syntax checked) the command scan service routine determines if the input buffer contains a subcommand. The subcommand name is considered to begin at the first non-separator character found, and end at the first command delimiter character found. See Figure 16 on page 57 for a list of the separator characters and command delimiters.

Command scan translates any lowercase letters in the subcommand name to uppercase, fills the command scan output area, updates the command buffer offset field, and returns to the calling program.

If you have requested syntax checking (X'00' in the flag field of the flag word), the command name must meet the syntax requirements described in "Syntax Requirements for Command and Subcommand Names" on page 56.

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. Figure 20 shows the possible CSOA settings and command buffer offset settings upon return from the command scan service routine.

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'10'	The buffer is empty or contains only separators.	Zero	The end of the buffer.
X'08'	The command name is syntactically invalid.	Zero	Unchanged.

Figure 20. Return from Command Scan - CSOA and Command Buffer Settings

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 invalid parameters.

Chapter 13. Verifying Command and Subcommand Operands with the Parse Service Routine

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)," "Character Types Accepted by the Parse Service Routine" on page 66 and "Services Provided by the Parse Service Routine" on page 67, 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, 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 specified in any order, as long as they follow all of the positional 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 specified 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.

The following describes the parse macro instructions and their functions:

IKJPARM	Begins the parameter control list and establishes a symbolic reference for the parameter descriptor list.
IKJPOSIT	Builds a parameter control entry. This PCE describes a positional operand that contains delimiters recognized by the parse service routine, but not including the positional operands described by the IKJTERM, IKJOPER, IKJIDENT, or IKJRSVWD macro instructions.
IKJTERM	Builds a parameter control entry. This PCE describes a positional operand that can be a constant, statement number, or variable.
IKJOPER	Builds a parameter control entry that describes an expression. An expression consists of three parts; two operands and an operator in the form: `(operand1 operator operand2)`
IKJRSVWD	Builds a parameter control entry. This PCE can be used with the IKJTERM macro instruction to describe a reserved word constant, with the IKJOPER macro instruction to describe the operator of an expression, or by itself to describe a reserved word operand.
IKJIDENT	Also builds a parameter control entry; however, this PCE describes a positional operand that does not depend upon a particular delimiter.
IKJKEYWD	Builds a parameter control entry that describes a keyword operand.
IKJNAME	Builds a parameter control entry that describes the possible names that can be specified for a keyword or a reserved word operand.
IKJSUBF	Indicates the beginning of a keyword subfield description. A subfield consists of a parenthesized list of positional or keyword operands directly following the keyword.
IKJENDP	Indicates the end of the PCL.
IKJRLSA	Releases any virtual storage allocated by the parse service routine that remains after the parse service routine has returned control to the command processor.

Figure 21 shows the interaction between a command processor and the parse service routine.

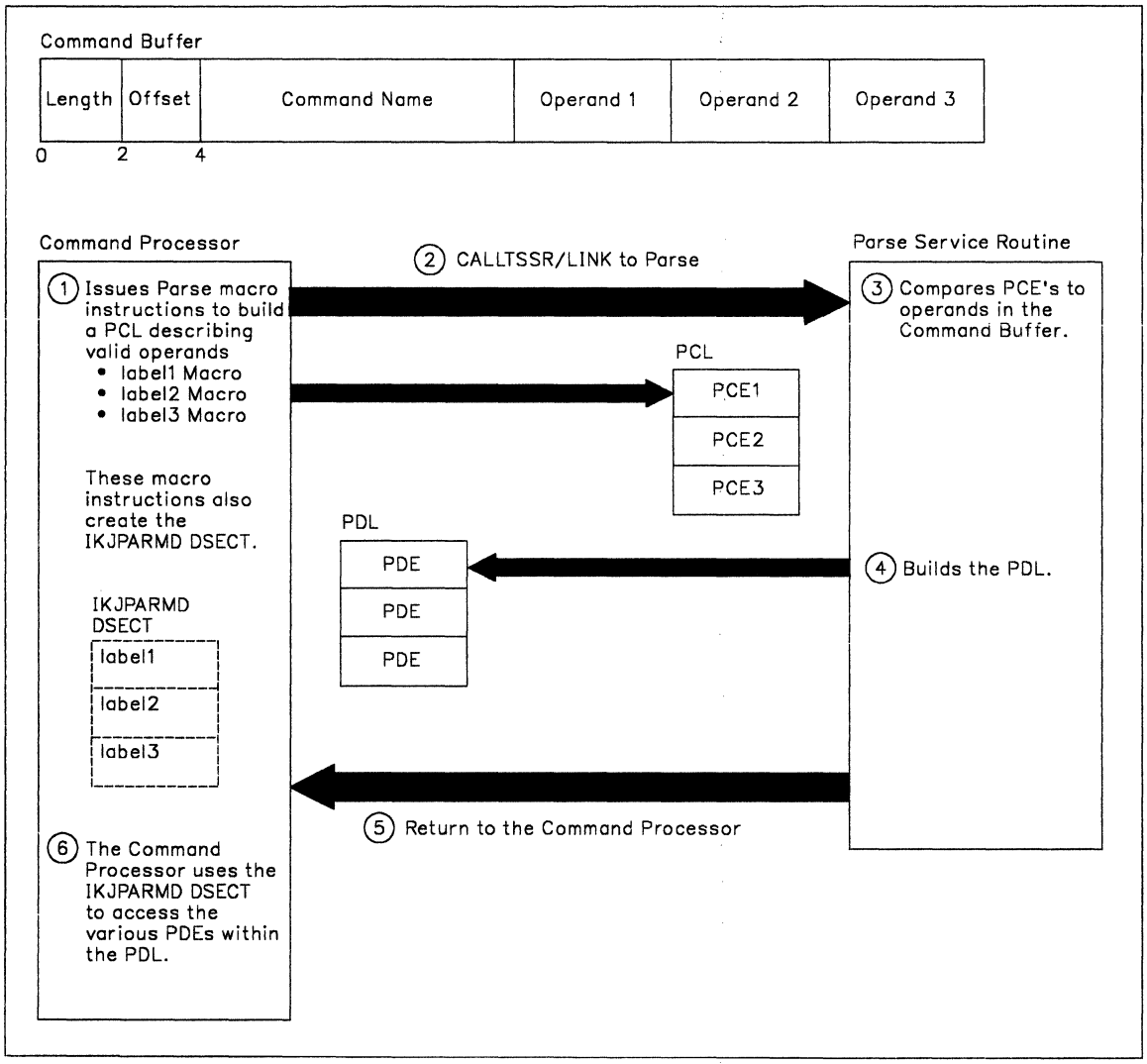


Figure 21. A Command Processor Using the Parse Service Routine

Character Types Accepted by the Parse Service Routine

Figure 22 shows the various character types that are recognized by the parse service routine. Throughout this chapter, the alphameric characters are as follows, unless otherwise indicated.

Alphabetic A - Z
Numeric 0 - 9
National $, #, @

Character		Character Type						
		Separator	National	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	@		X					
	a-z			X				
	A-Z			X				
	0-9				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

Figure 22. Character Types Recognized by the Parse Service Routine

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. If command operands are incorrect or if required operands are missing, parse writes an error message to the output data set.

In addition, the parse service routine provides the following services that can be selected by the calling routine:

- 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 translates the command operands to uppercase.
- It substitutes default values for missing operands.
- It inserts implied keywords.

Notifying the User about Missing or Required Operands

The parse service routine notifies the TSO user if the command operands found are incorrect or if required operands are missing. The parse service routine writes error messages to the output data set in the following situations:

- A dsname was specified with a slash but without a password.
- An operand is syntactically invalid.
- A keyword is ambiguous, that is, it is not clear to the parse service routine which keyword of several similar ones is being specified.
- 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 invalid.

In these cases, the parse service routine issues an error message and returns a code to the calling routine indicating that the TSO user specified an incorrect command. Parse appends any second level messages to the error message for the missing or invalid operand.

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.

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 to the output data set. 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.

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 115 describes what you must do to provide a validity checking routine.

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.

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.

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 specified. 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 specified on the command.

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 69.

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 Instructions to Define Command Syntax" on page 81.
   - Use the IKJPARM macro instruction to begin the 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 PCL for the command or subcommand.
3. Write validity checking routines to do additional checking on positional operands. See "Using Validity Checking Routines" on page 115 for a discussion of this topic.
4. Pass control to the parse service routine. See "Passing Control to the Parse Service Routine" on page 117.
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 119.
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 121 for a description of the PDEs returned by parse.

GENERAL-USE PROGRAMMING INTERFACE

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 specified 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 Figure 22 on page 66). The command operands end either at the end of a logical line, or at a semicolon. If the command operands end with a semicolon, and other characters are specified 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 specified first in the parameter string, and they must be entered in a specific order.

Keyword operands This type can be specified anywhere in the command as long as they follow all positional operands. Discussion of keyword operands begins on page 80.

Positional Operands

Positional operands must be specified 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 issues a message if the operand 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 - operands that include delimiters as part of their definition. Delimiter-dependent operands are discussed below.

Non-delimiter-dependent operands - operands that do not include delimiters as part of their definition. Non-delimiter-dependent operands are discussed on page 78.

Delimiter-Dependent Operands

Those operands that include delimiters as part of their definition are called delimiter-dependent operands. Figure 23 shows the delimiter-dependent syntaxes that the parse service routine recognizes and the macro instruction that is used to specify each type.

Figure 23. Delimiter-Dependent Operands

Operand	Macro Instruction Used to Describe Operand
DELIMITER STRING VALUE ADDRESS PSTRING 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, or digit. A self-defining delimiter character is represented in this discussion by the symbol #. The delimiter operand is used only in conjunction 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 specified, 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 specified as two contiguous delimiters. Note that a string received from a default must not include the delimiters.

VALUE

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

X'string'

The character must be an alphabetic or national character. 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 an alphabetic or national character, or if the second character is not an apostrophe.

ADDRESS

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

Absolute address

An 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 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 83.)

Relative address

A 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

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

Floating-point register address

A 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 specified in either uppercase or lowercase.

Symbolic address

A symbolic address consists of any combination, up to 32 characters in length, of the alphameric characters and the break character. The first character must be either an alphabetic or a national character.

Qualified address

A qualified address has one of the following formats:

```
1. modulename.entryname.relative-address
2. modulename.entryname
3. modulename.entryname.symbolic-address
4. .entryname.symbolic-address
5. .entryname.relative-address
6. .entryname
```

- modulename - any combination of one to eight alphameric characters, where the first is an alphabetic or national character
- entryname - same syntax as a modulename, 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 1-3. However, if the address applies to the currently active module, it is not necessary to specify *modulename*, as in formats 4-6.

Indirect address

An indirect address is an absolute, relative, symbolic, or general register address followed by from one to 255 indirection symbols (percent signs), such as:

```
+A%
```

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

End of GENERAL-USE PROGRAMMING INTERFACE

Figure 24 shows an example of an indirect address that is made up of a relative address with one level of indirect addressing.

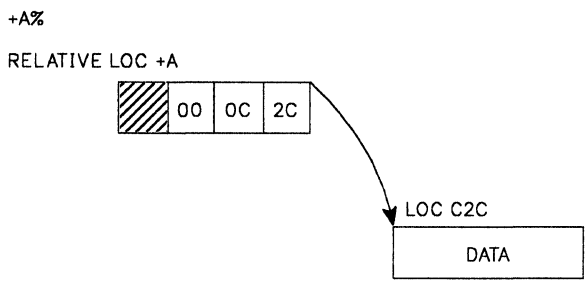


Figure 24. Example of Indirect Addressing

The number of indirection symbols following the address indicates the number of levels of indirect addressing. In Figure 24, the data is at the location pointed to by bits 0-24 of relative address +A.

GENERAL-USE PROGRAMMING INTERFACE

Address expression

An address expression has the following format:

```
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
  - When you specify the EXTENDED keyword of IKJPOSIT to indicate extended mode, the user can specify 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 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 indirect addressing, is shown in Figure 25. 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.

End of GENERAL-USE PROGRAMMING INTERFACE

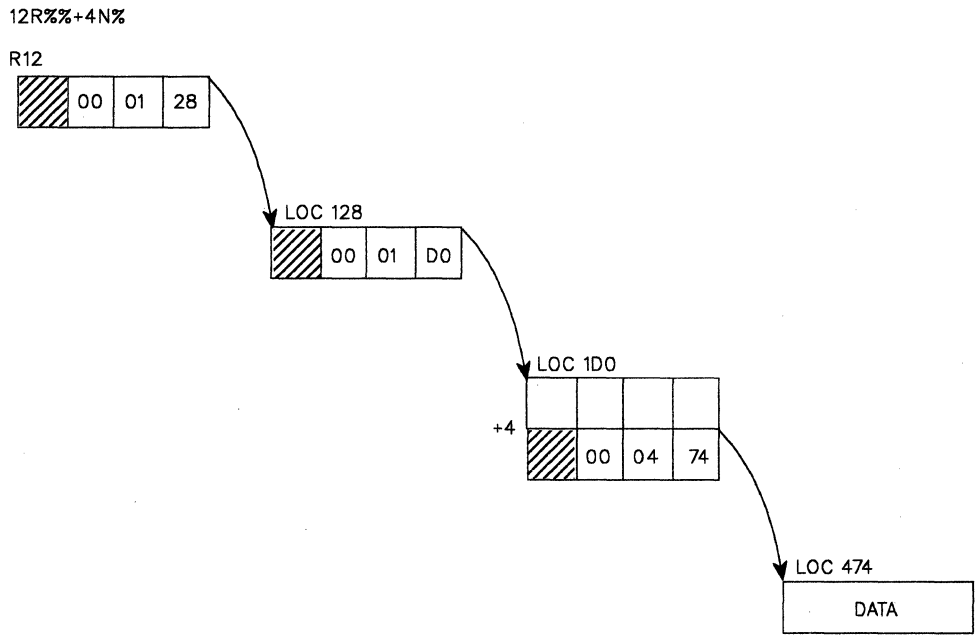


Figure 25. An Address Expression with Indirect Addressing

GENERAL-USE PROGRAMMING INTERFACE

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.

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 alphameric characters up to eight characters in length, the first character of which must be an alphabetic or national character.

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 alphameric characters, the first of which must be an alphabetic or a national character.

The parse service routine considers the entire dsname operand missing if the first character scanned is not an apostrophe, an alphabetic character, a national 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 the slash and the password are not specified, the parse service routine does not issue a message for a missing password.

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.

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

'string'

The string can consist of combination of characters, of any length, with one restriction: if the user wants to specify apostrophes within the string, two successive apostrophes must be specified for each single apostrophe desired. 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.

SPACE
Space is a special purpose operand; it allows a string operand that directly follows a command name to be specified 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 alphameric characters of which the first is alphabetic or national ($, #, @). 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, specified 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.

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 specified for a data-name is limited to 255.

Subscript
Consists of a data-name with subscripts enclosed in parentheses following the data-name specified 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 specified for a data-name is limited to 3, specified 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

where:

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 alphameric (A through Z or 0 through 9).

Line number
Consists of a string of digits (0 through 9) and cannot exceed a length of 6 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 specified 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 nonquoted character string where a separator or delimiter indicates the end.

CHAR
A character string is any data in the form of a quoted or nonquoted string.

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 or national character.

NUMERIC
Indicates a number (0-9).

ALPHANUM
Indicates an alphabetic or national character 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; national characters ($, #, @) are not accepted.

NONATNUM
Indicates numbers and alphabetic characters are accepted; national characters ($, #, @) are not accepted.

An asterisk can be specified 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 specified 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 specified enclosed in parentheses as follows:

```
(positional-operand positional-operand...)
```

If one or more of the items contained in the list are to be specified 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
DSNAME
DSTHING
JOBNAME
CONSTANT
STATEMENT NUMBER
VARIABLE
HEX
CHAR
INTEG
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
Any positional operand that is not dependent upon delimiters

If the user specifies an operand that begins with a left parentheses, and you have specified in either the IKJPOSIT or IKJIDENT macro instruction that the operand can be specified 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 specified as a list, and the TSO user wants to specify a dsname of the form:

```
(membername)/password
```

The user must specify it as:

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

Keyword Operands

Keyword operands can be specified anywhere in the command as long as they follow all positional operands. They can consist of any combination of alphameric 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, 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 specifies a keyword with a subfield in which there is a required operand, but does not specify the subfield, the parse service routine issues a message.

If a subfield has a positional operand that can be specified 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 specified 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 Instructions 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 specified.

The command processor builds the PCL and the PCEs within it by using the parse macro instructions. These macro instructions generate the PCL and establish symbolic references for the parameter descriptor list (PDL). The PDL is returned to the command processor by the parse service routine 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.

Figure 26 describes the functions of each of the parse macro instructions.

Figure 26. 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 specified for a keyword or reserved word operand.
IKJSUBF	Builds a PCE that indicates the beginning of a keyword subfield description.
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:

- When complete, all of the parse macros, except for IKJRLSA, return to the user's CSECT. If a DSECT appears between the CSECT statement and the parse macro(s), an assembly error occurs. To prevent this error, place the DSECT after the macro(s).
- The IKJPOSIT, IKJTERM, IKJOPER, IKJRSVWD, IKJIDENT, IKJKEYWD, IKJNAME, and IKJSUBF macro instructions describe the positional and keyword operands valid for the command processor. The label fields of these macro instructions are used by your command processor to reference fields within the DSECT that maps the PDL returned by the parse service routine.

The macros that generate input to parse must be issued by a program that is loaded below 16 megabytes in virtual storage so that parse can access the PCL. The IKJRLSA macro instruction must be issued in 24-bit addressing mode.

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 27 shows the format of the IKJPARM macro instruction. Each of the operands is explained following the figure.

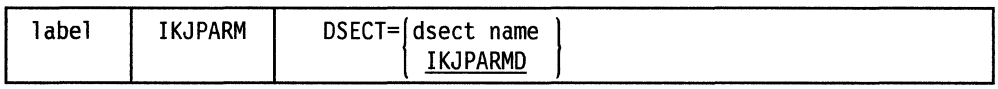


Figure 27. The IKJPARM Macro Instruction

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

DSECT=
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 Figure 28. This PCE begins the parameter control list.

Figure 28. The Parameter Control Entry Built by IKJPARM

Number of Bytes	Field	Contents or Meaning
2		Length of the parameter control list. This field contains a hexadecimal number representing the number of bytes in this PCL.
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
DELIMITER
STRING
VALUE
ADDRESS
PSTRING
DSNAME
DSTHING
QSTRING
JOBNAME

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 29 shows the format of the IKJPOSIT macro instruction. Each of the operands is explained following the figure.

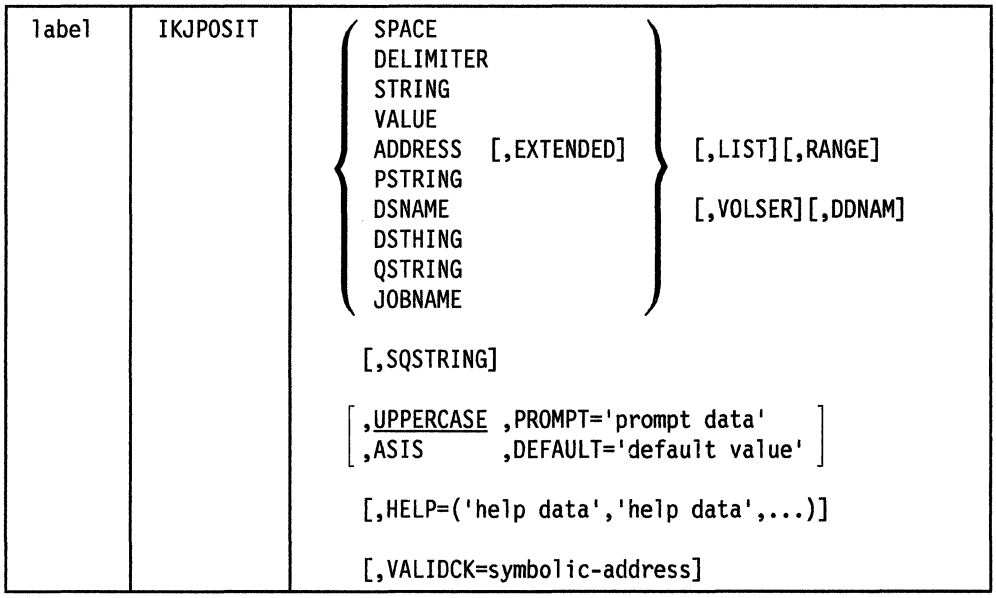


Figure 29. The IKJPOSIT Macro Instruction

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 70.

Positional Operand Type	Where Described
SPACE	Page 75
DELIMITER	Page 70
STRING	Page 71
VALUE	Page 71
ADDRESS	Page 71
PSTRING	Page 74
DSNAME	Page 74
DSTHING	Page 75
QSTRING	Page 75
JOBNAME	Page 75

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 TSO user 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. If you omit the EXTENDED operand, the parse service routine processes all addresses as 24-bit addresses. For more information, refer to the description of the address operand on page 71.

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

```
commandname (operand,operand, ...)
```

This list option can be used with the following delimiter-dependent positional operands:

DSNAME, DSTHING, ADDRESS, VALUE, JOBNAME, and PSTRING (within a subfield only).

RANGE
The command operands can be specified by the 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 invalid.

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 specified by the 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 specified by the user. If the operand is not specified, the parse service routine supplies a message ID and adds the word MISSING to the beginning of this message before writing it to the output data set.

DEFAULT = 'default value'
The operand described by this IKJPOSIT macro instruction is required, but the user need not specify 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 written to the output data set following the prompting message. Parse adds a message ID and the word MISSING to the beginning of each message before writing it to the output data set.

These messages are not issued when the missing operand is a password on a dsname operand.

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 Figure 30.

Figure 30 (Page 1 of 2). The Parameter Control Entry Built by IKJPOSIT

Number of Bytes	Field	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...	SQSTRING
	.... .0..	Reserved
	.... ..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.
1		This field contains a hexadecimal number indicating the type of positional operand described by this PCE. These numbers have the following meaning: HEX 1 DELIMITER 2 STRING 3 VALUE 4 ADDRESS 5 PSTRING 6 Not Used 7 DSNAME 8 DSTHING 9 QSTRING A SPACE B JOBNAME C Not Used D EXTENDED ADDRESS E to FF Not Used.
1		Contains the length minus one of the default or prompting information supplied on the IKJPOSIT macro instruction. This field and the next 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.

Figure 30 (Page 2 of 2). The Parameter Control Entry Built by IKJPOSIT

Number of Bytes	Field	Contents or Meaning
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.
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 31 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 31. 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 specified. Blanks within the apostrophes are allowed.

LIST
The command operands can be specified by the 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 specified by the 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 specified by the user.

TYPE=

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 70 for a syntactical definition of these operands.

SBSCRIPT[=label-PCE]

Specifies one of two conditions:

1. If you specify SBSCRIPT 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 SBSCRIPT 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 specified by the user.

If the operand is not specified, the parse service routine adds a message ID and the word MISSING to the beginning of the message before writing it to the output data set.

DEFAULT='default value'

The operand described by this IKJTERM macro instruction is required, but the user need not specify it. If the operand is not specified, 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 written to the output data set following the prompting message.

Parse adds a message ID and the word MISSING to the beginning of each message before writing it to the output data set.

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 specified 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 Figure 32.

Figure 32 (Page 1 of 2). The Parameter Control Entry Built by IKJTERM

Number of Bytes	Field	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.
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	Contains the hexadecimal length of the parameter-type field.
	Byte 3-4	Contains 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.

Figure 32 (Page 2 of 2). The Parameter Control Entry Built by IKJTERM

Number of Bytes	Field	Contents or Meaning
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 specified 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 33 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 33. 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 specified by the 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 invalid, parse issues a message 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 specified by the user. If the operand is not specified, the parse service routine supplies a message ID and adds the word MISSING to the beginning of this message before writing it to the output data set.

DEFAULT = 'default value'
The operand described by this IKJOPER macro instruction is required, but the user need not specify it. If the operand is not specified, 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 written to the output data set following the prompting message.

Parse adds a message ID and the word MISSING to the beginning of each message before writing it to the output data set.

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 specified 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 specified 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 specified 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 Figure 34.

Figure 34 (Page 1 of 2). The Parameter Control Entry Built by IKJOPER

Number of Bytes	Field	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	Contains the hexadecimal length of the parameter-type field.
	Byte 3-4	Contains 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.

Figure 34 (Page 2 of 2). The Parameter Control Entry Built by IKJOPER

Number of Bytes	Field	Contents or Meaning
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.
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 35 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 35. 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 specified by the 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 specified by the user. If the operand is not specified, the parse service routine supplies a message ID and adds the word MISSING to the beginning of this message before writing it to the output data set.

DEFAULT = 'default value'
The operand described by this IKJRSVWD macro instruction is required, but the user need not specify it. If the operand is not specified, 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 written to the output data set following the prompting message.

Parse adds a message ID and the word MISSING to the beginning of each message before writing it to the output data set.

The Parameter Control Entry Built by IKJRSVWD

The IKJRSVWD macro instruction generates the variable parameter control entry (PCE) shown in Figure 36.

Figure 36. The Parameter Control Entry Built by IKJRSVWD

Number of Bytes	Field	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 Byte 3-4	Contains the hexadecimal length of the parameter-type field. Contains 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 78.

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 37 shows the format of the IKJIDENT macro instruction. Each of the operands is explained following the figure.

```
label    IKJIDENT   'parameter-type' [,LIST][,RANGE]

                    [,ASTERISK][,UPPERCASE ][,MAXLNTH=number]
                               [,ASIS      ]

                    [ ,FIRST= { ALPHA    } ] [ ,OTHER= { ALPHA    } ]
                              { NUMERIC  }               { NUMERIC  }
                              { ALPHANUM }               { 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 37. 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 specified by the user. Blanks within the apostrophes are allowed.

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

```
commandname (operand,operand,...)
```

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

```
commandname operand:operand
```

If you specify RANGE and OTHER=ANY, parse treats any colons 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.

ASTERISK
An asterisk can be substituted for this positional operand.

Note: ASTERISK and INTEG are mutually exclusive.

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. This 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.

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 or national character. ALPHA is the default value for both the FIRST and the OTHER operands.

NUMERIC
A digit, 0-9.

ALPHANUM
An alphabetic, numeric, or national character.

ANY
Any character other than a blank, comma, tab, or semicolon. Parentheses must be balanced.

NONATABC
An alphabetic character only. National characters and numerics are excluded.

NONATNUM
An alphabetic or numeric character. National 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 specified by the user. If the operand is not specified, the parse service routine supplies a message ID and adds the word MISSING to the beginning of this message before writing it to the output data set.

DEFAULT = 'default value'

The operand is required, but a default value can be used. If the operand is not specified by the user, the value specified as the default value is used.

Note: The operand is optional if neither PROMPT nor 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 specified in the form X'n...' must be valid hexadecimal digits (0-9, A-F). All input specified 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. (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 written to the output data set following the prompting message. These messages are not issued when the prompt is for a password on a dsname operand.

Parse adds a message ID and the word MISSING to the beginning of each message before writing it to the output data set.

The Parameter Control Entry Built by IKJIDENT

The IKJIDENT macro instruction generates the variable length parameter control entry (PCE) shown in Figure 38.

Figure 38 (Page 1 of 2). The Parameter Control Entry Built by IKJIDENT

Number of Bytes	Field	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
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	Integer
	.... 1...	Character
	.... .1..	Hexadecimal
	..0. ..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: 0 Any (except blank, comma, tab, semicolon) 1 Alphabetic or national 2 Numeric 3 Alphabetic, national, or numeric 4 Alphabetic 5 Alphabetic or numeric 6 to FF Not used

Figure 38 (Page 2 of 2). The Parameter Control Entry Built by IKJIDENT

Number of Bytes	Field	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 national 2 Numeric 3 Alphabetic, national, or numeric 4 Alphabetic 5 Alphabetic or numeric 6 to FF Not used
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 105 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 alphameric characters up to 31 characters in length, the first of which must be an alphabetic character.

Figure 39 shows the format of the IKJKEYWD macro instruction. Each of the operands is explained following the figure.

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

Figure 39. 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 Figure 40.

Figure 40 (Page 1 of 2). The Parameter Control Entry Built by IKJKEYWD

Number of Bytes	Field	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.

Figure 40 (Page 2 of 2). The Parameter Control Entry Built by IKJKEYWD		
Number of Bytes	**Field**	**Contents or Meaning**
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.
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 41 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 41. 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 107 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 specified 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.

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 42 shows the format of the IKJNAME macro instruction. Each of the operands is explained following the figure.

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

Figure 42. 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 96 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 Figure 43.

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

Figure 43 (Page 1 of 2). The Parameter Control Entry Built by IKJNAME

Number of Bytes	Field	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.

Figure 43 (Page 2 of 2). The Parameter Control Entry Built by IKJNAME

Number of Bytes	Field	Contents or Meaning
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.
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 minus one of first alias.
Variable		The first alias.
1		The length minus one 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, 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 44 shows the format of the IKJSUBF macro instruction.

```
label   IKJSUBF
```

Figure 44. 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 Figure 45.

Figure 45. The Parameter Control Entry Built by IKJSUBF

Number of Bytes	Field	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 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 46 shows the format of the IKJENDP macro instruction.

```
IKJENDP
```

Figure 46. The IKJENDP Macro Instruction

The Parameter Control Entry Built by IKJENDP

The IKJENDP macro instruction generates the parameter control entry (PCE) shown in Figure 47. It is merely an end-of-field indicator.

Figure 47. The Parameter Control Entry Built by IKKJENDP

Number of Bytes	Field	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.

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 48 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 48. 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 117 for a description of the parse parameter list.

End of GENERAL-USE PROGRAMMING INTERFACE

Examples Using the Parse Macro Instructions

Example 1

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 Chapter 4, "Validating Command Operands" on page 13.

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

PROCESS	dsname	[ACTION / NOACTION]

Figure 49 on page 110 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='DATA SET NAME TO BE PROCESSED',             X
               VALIDCK=POSITCHK
ACTPCE   IKJKEYWD DEFAULT='NOACTION'
         IKJNAME  'ACTION'
         IKJNAME  'NOACTION'
         IKJENDP
```

Figure 49. Example 1 - Using Parse Macros to Describe Command Operand Syntax

Example 2

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:

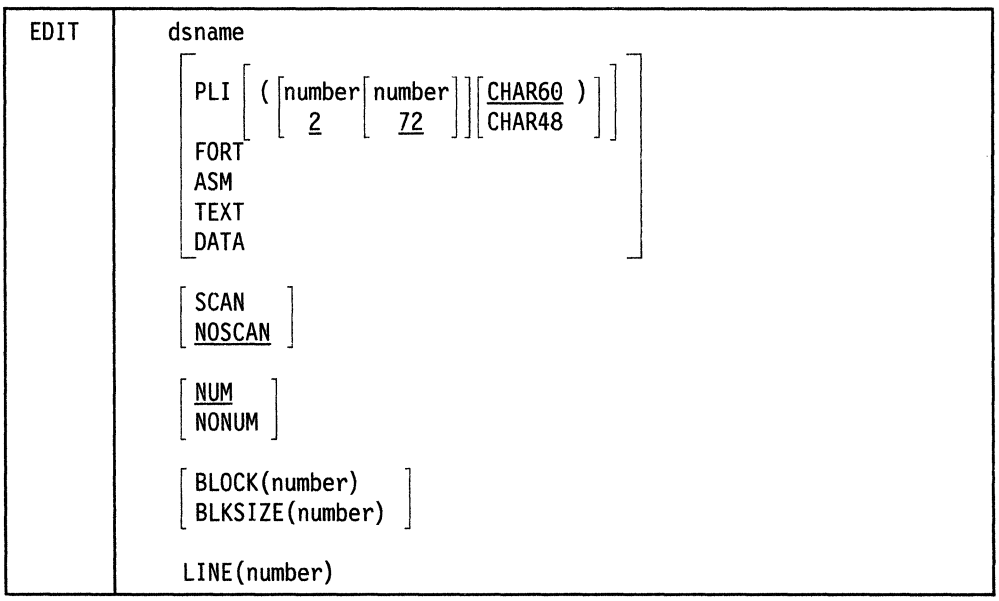


Figure 50 on page 111 shows the sequence of parse macro instructions that describe the syntax of this EDIT command to the parse service routine. The parse macro instructions used in this example perform the following functions:

- 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 50. Example 2 - Using Parse Macros to Describe Command Operand Syntax

Example 3

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:

COMMAND	OPERANDS
AT	{ stmt (stmt-1,stmt-2,...) stmt-3:stmt-4 } (cmd chain) COUNT(integer)

Figure 51 on page 112 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 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 51. Example 3 - Using Parse Macros to Describe Command Operand Syntax

Example 4

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:

COMMAND	OPERANDS
LIST	symbol PRINT(symbol)

Figure 52 on page 113 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 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 52. Example 4 - Using Parse Macros to Describe Command Operand Syntax

Example 5

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:

COMMAND	OPERANDS
WHEN	{addr / expression} (subcommand chain)

Figure 53 on page 114 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 specified 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 53. 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
- 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 operand is entered with a password, parse passes control to the validity checking routine after first parsing both the dsname and the password. If the user specifies a list, the validity checking routine is called as each element in the list is parsed. If a range is specified, the parse service routine calls the validity checking routine only after both items of the range are parsed.

Passing Control to Validity Checking Routines

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 Figure 54.

Figure 54. Format of the Validity Check Parameter List

Field Label	Byte Offset	Byte Length	Contents or Meaning
PDEADR	0(0)	4	The address of the parameter descriptor entry (PDE) built by parse for this syntactically correct operand.
USERWORD	4(4)	4	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.
VALMSG	8(8)	4	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. Figure 55 shows the return codes, their meaning, and the action taken by the parse service routine.

Figure 55. 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 invalid.	The parse service routine writes an error message to the output data set.
8(8)	The operand is invalid.	The validity checking routine has issued an error message to the output data set.
12(C)	The operand is invalid; 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.

Prior to issuing a return code of 12, your validity checking 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. The PPL is described in "The Parse Parameter List."

The parse service routine must receive control in 24-bit addressing mode. If your program uses the CALLTSSR macro instruction to invoke IKJPARS, and IKJPARS resides in the link pack area, your program must issue the CALLTSSR macro instruction in 24-bit addressing mode. However, if IKJPARS does *not* reside in the link pack area, your program can issue the CALLTSSR macro instruction in either 24- or 31- bit addressing mode.

Before you invoke the parse service routine, you must build a 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 Parse Parameter List

The parse parameter list (PPL) is a seven-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. Figure 56 shows the format of the parse parameter list.

Figure 56. The Parse Parameter List

Field Label	Byte Offset	Byte Length	Contents or Meaning
PPLUPT	0(0)	4	The address of the user profile table.
PPLECT	4(4)	4	The address of the environment control table.
PPLECB	8(8)	4	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.
PPLPCL	12(C)	4	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.
PPLANS	16(10)	4	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'.
PPLCBUF	20(14)	4	The address of the command buffer.
PPLUWA	24(18)	4	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.

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:

Figure 57. Return Codes from the Parse Service Routine

Return Code Dec(Hex)	Meaning
0(0)	Parse completed successfully.
4(4)	The command operands were incomplete.
12(C)	Parse did not complete; the parse parameter list contains invalid information.
16(10)	Parse did not complete; parse issued a GETMAIN and 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.

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 121 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 or 20. When the return code is 4, parse has already issued a message. When the return code is 20, the validity checking routine 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 18, "Analyzing Error Conditions with the GNRLFAIL/VSAMFAIL Routine (IKJEFF19)" on page 235.

End of GENERAL-USE PROGRAMMING INTERFACE

All input passed to IKJPARS must reside below 16 megabytes in virtual storage. Figure 58 shows this flow of control between a command processor and the parse service routine.

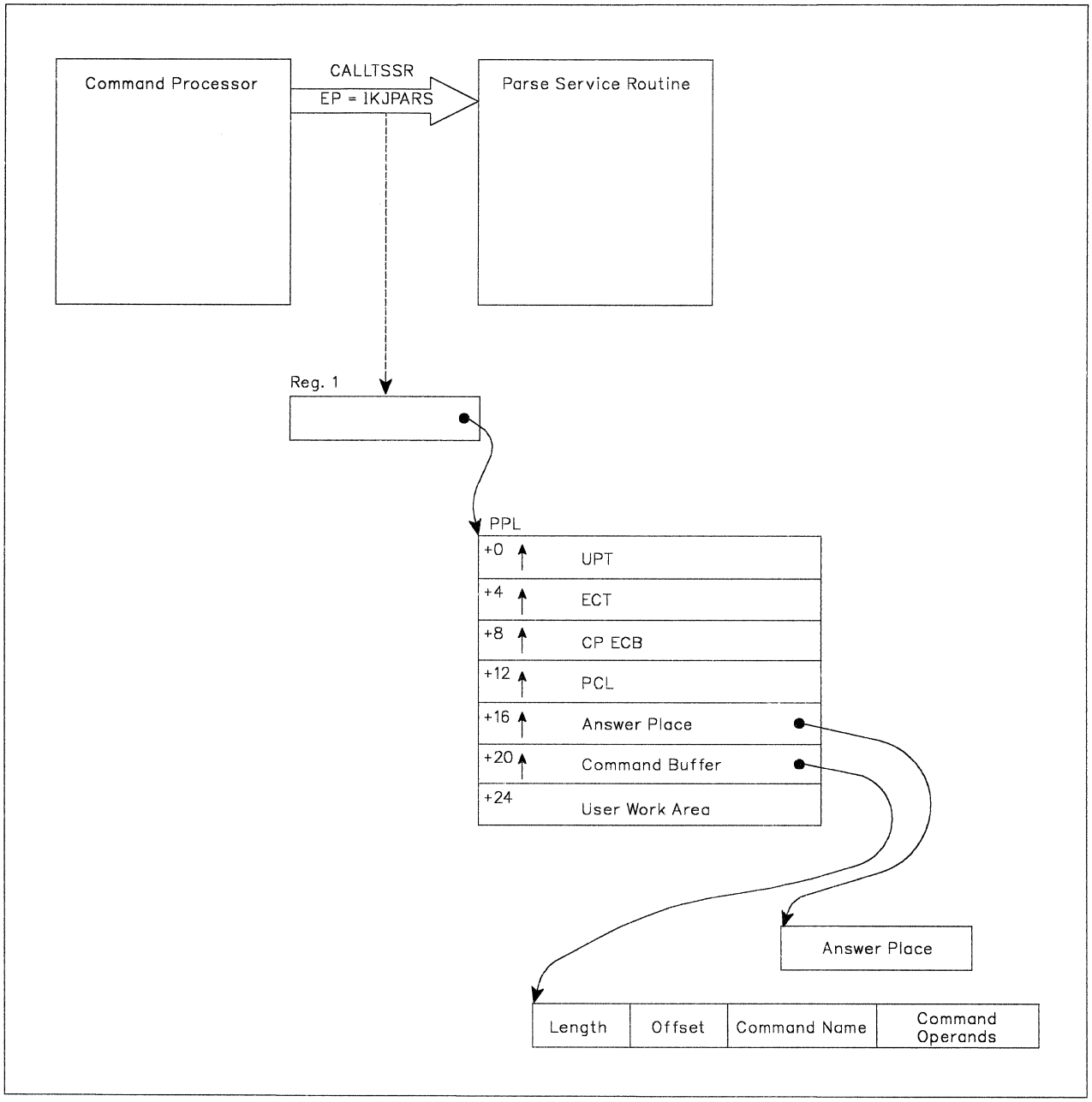


Figure 58. Control Flow between Command Processor and the Parse Service Routine

Examining the PDL Returned by the Parse Service Routine

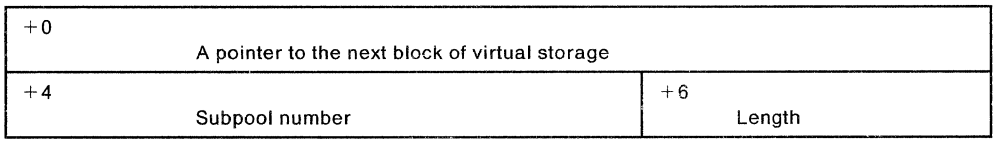
The parse service routine returns the results of the scan of the command buffer to the command processor in a parameter descriptor list (PDL). The PDL, built by parse, consists of the parameter descriptor entries (PDE), which contain pointers to the operands, indicators of the options specified, and 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. Then use the labels you used on the parse macro instructions to access the corresponding PDEs.

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 69. 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.



Pointer to the next block of virtual storage:
The parse service routine gets virtual storage for the PDL. 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. Virtual storage allocated by the parse service routine for the PDL 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

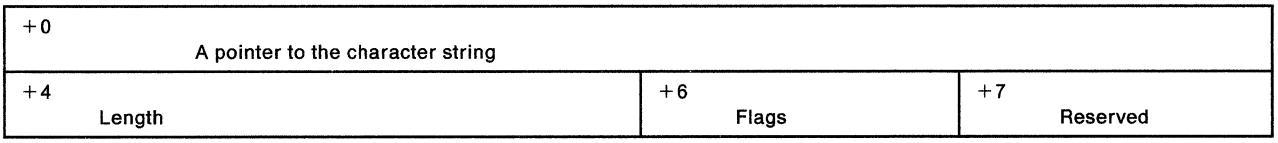
The labels you use to name the macro instructions provide access to the corresponding PDEs. The positional operands described by the IKJPOSIT macro instruction have the following PDE formats.

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 uses the IKJPOSIT macro to build a two-word PDE to describe a STRING, PSTRING, or a QSTRING operand; the PDE has the following format:



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.

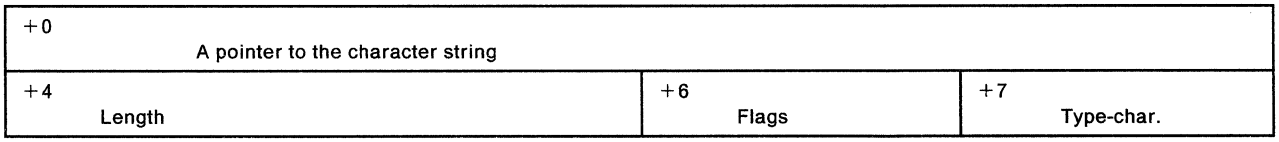
Flags:

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 uses the IKJPOSIT macro to build a two-word PDE to describe a VALUE operand; the PDE has the following format:



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 uses the IKJPOSIT macro instruction to build a six-word PDE to describe a DSNAME or a DSTHING operand. The PDE has the following format:

+0 A pointer to the dsname		
+4 Length1	+6 Flags1	+7 Reserved
+8 A pointer to the member name		
+12 Length2	+14 Flags2	+15 Reserved
+16 A pointer to the password		
+20 Length3	+22 Flags3	+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.

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.

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 uses the IKJPOSIT macro to build a four word PDE to describe a JOBNAME operand. The PDE has the following format:

+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 uses the IKJPOSIT macro to build a nine word PDE to describe an ADDRESS operand. The PDE has the following format:

<table>
<tr><td colspan="3">+0
A pointer to the load name</td></tr>
<tr><td>+4
Length1</td><td>+6
Flags1</td><td>+7
Reserved</td></tr>
<tr><td colspan="3">+8
A pointer to the entry name</td></tr>
<tr><td>+12
Length2</td><td>+14
Flags2</td><td>+15
Reserved</td></tr>
<tr><td colspan="3">+16
A pointer to the address string</td></tr>
<tr><td>+20
Length3</td><td>+22
Flags3</td><td>+23
Reserved</td></tr>
<tr><td>+24
Flags4</td><td>+25
Sign</td><td>+26
Indirect count</td></tr>
<tr><td colspan="3">+28
A pointer to the first expression value PDE</td></tr>
<tr><td colspan="3">+32
Reserved for use by user validity check routine</td></tr>
</table>

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.

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	Letters.
Absolute address	The period.

Flags3:
The bits set in this one-byte flag field indicate whether the address string is present.

Setting	Meaning
0...	The address string is not present.
1...	The address string is present.
.xxx xxxx	Reserved bits.

Offset 23:
This byte is reserved for use by a validity checking routine.

Flags4:
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).

Sign:
Contains the arithmetic sign character used before the expression value defined by the first expression value PDE. 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.

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 uses the IKJPOSIT macro to build a four-word PDE to describe an expression value; it has the following format:

<table>
<tr><td colspan="4">+0
A pointer to the address string</td></tr>
<tr><td colspan="2">+4
Length3</td><td>+6
Reserved</td><td>+7
Reserved</td></tr>
<tr><td>+8
Flags5</td><td>+9
Sign</td><td colspan="2">+10
Indirect count</td></tr>
<tr><td colspan="4">+12
A pointer to the next expression value PDE</td></tr>
</table>

Pointer to the address string:
Contains a pointer to the expression value address string.

Length3:
Contains the length of the expression value address string. The N is not included in this length value.

Flags5:
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 0100	04	This is a decimal expression value.
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 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.

PDEs Created for Positional Operands Described by IKJTERM

CONSTANT

The parse service routine uses the IKJTERM macro to build a five-word PDE to describe a CONSTANT operand. The PDE has the following format:

+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 specified 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 specified 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 specified 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.
..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 uses the IKJTERM macro to build a five-word PDE to describe a STATEMENT NUMBER operand. The PDE has the following format:

+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 specified. Contains zero if not present.

Pointer to the line number:
Contains a pointer to the line number, if specified. Contains zero if not present.

Pointer to the verb number:
Contains a pointer to the verb number, if specified. Contains zero if not present.

VARIABLE

The parse service routine builds a five-word PDE (when using the IKJTERM macro) to describe a VARIABLE operand. The PDE has the following format:

+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 specified.

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 specified. This field contains zero if the optional program-id name is not present.

Length2:
Contains the length of the program-id name, if specified. 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:

+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 uses the IKJOPER macro to build a two-word PDE to describe an EXPRESSION operand. The PDE has the following format:

+0 Reserved		
+4 Reserved	+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 uses the IKJRSVWD macro instruction to build a two-word PDE to describe a RESERVED WORD operand. The PDE has the following format:

+0 Reserved	+2 Reserved-word number	
+4 Reserved	+6	+7 Flags 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 uses the IKJIDENT macro instruction to build a two-word PDE to describe a non-delimiter-dependent positional operand; it has the following format:

+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.

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:

- DSNAME
- DSTHING
- ADDRESS
- VALUE
- CONSTANT
- VARIABLE
- STATEMENT NUMBER
- HEX
- INTEG
- CHAR
- 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 user cannot actually specify a list. If a list is not specified, this word is set to X'FF000000'. If a list is specified, 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. Since 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 59 shows the PDL returned by the parse service routine after two positional operands have been specified. In this case, the first operand, a STRING operand, had been defined as not accepting lists. The second operand, a VALUE operand, had the LIST option coded in the IKJPOSIT macro instruction that defined the operand syntax. The VALUE operand was specified as a two-element list.

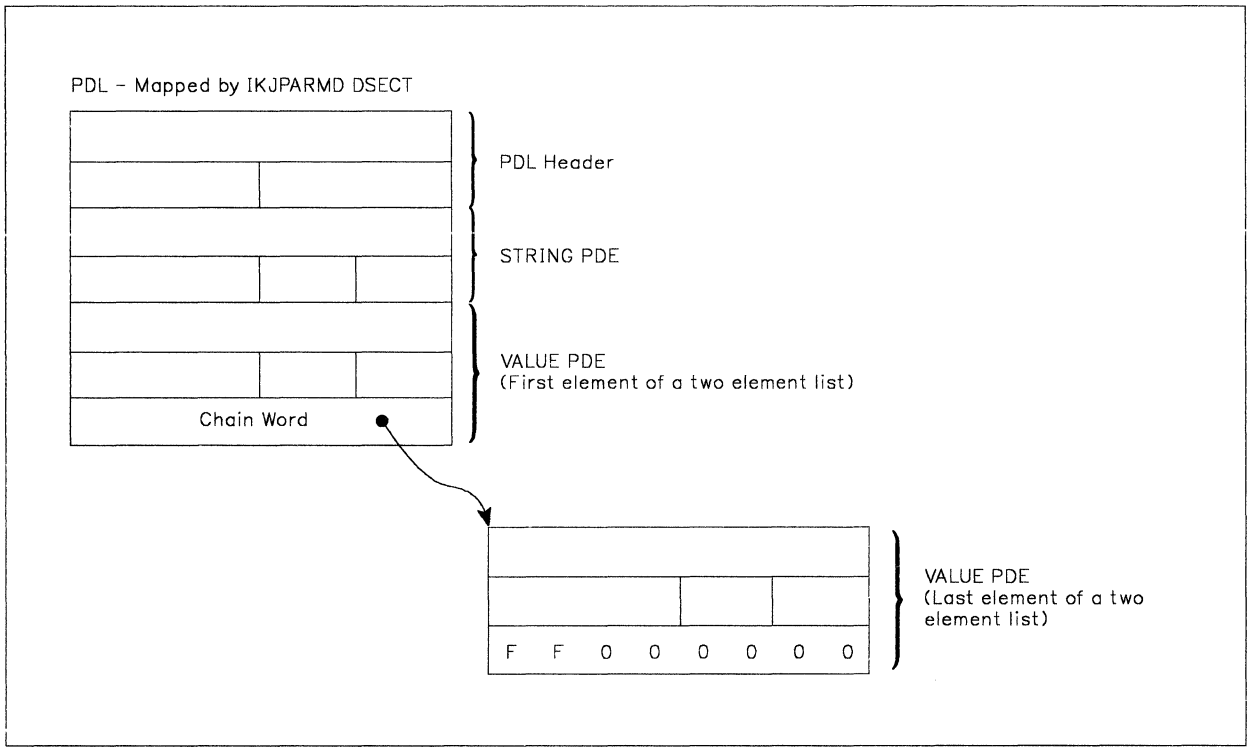


Figure 59. A PDL Showing PDEs that Describe a List

RANGE

The RANGE option can be used with the following positional operand types:

- HEX (X' ' only)
- ADDRESS
- VALUE
- CONSTANT
- VARIABLE
- STATEMENT NUMBER
- INTEG
- 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 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 60 shows the PDL returned by the parse service routine after two positional operands have been specified. In this case, the first operand is a STRING 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 specified as a range, and, consequently, parse sets the second PDE to zero.

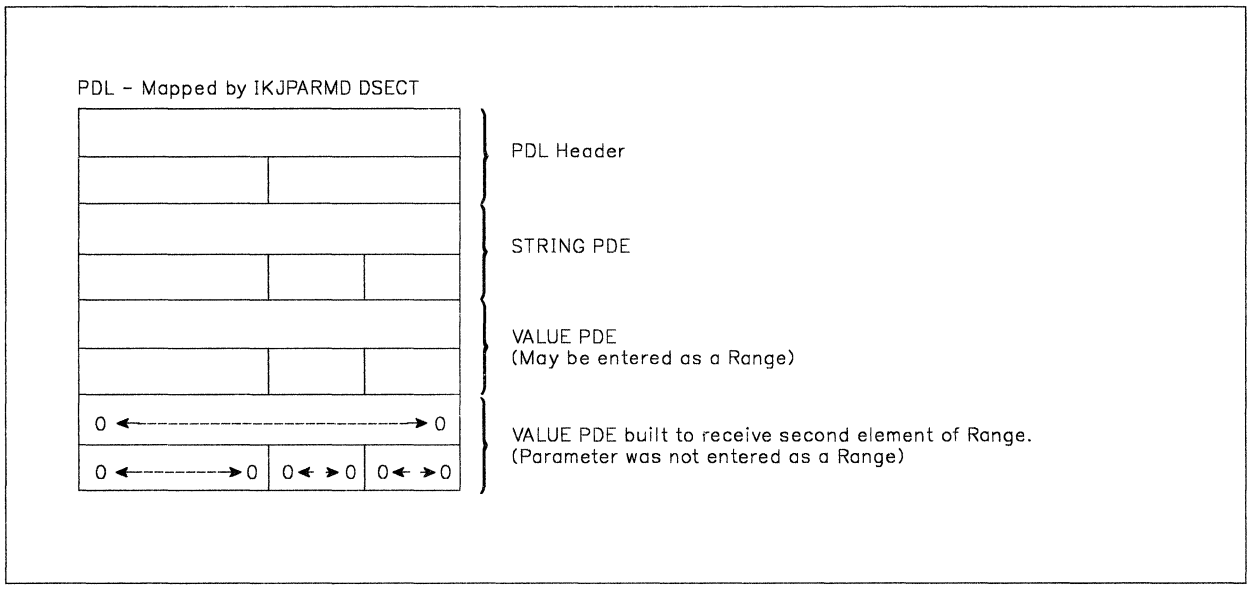


Figure 60. A PDL Showing PDEs Describing a Range

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 61 shows this general format.

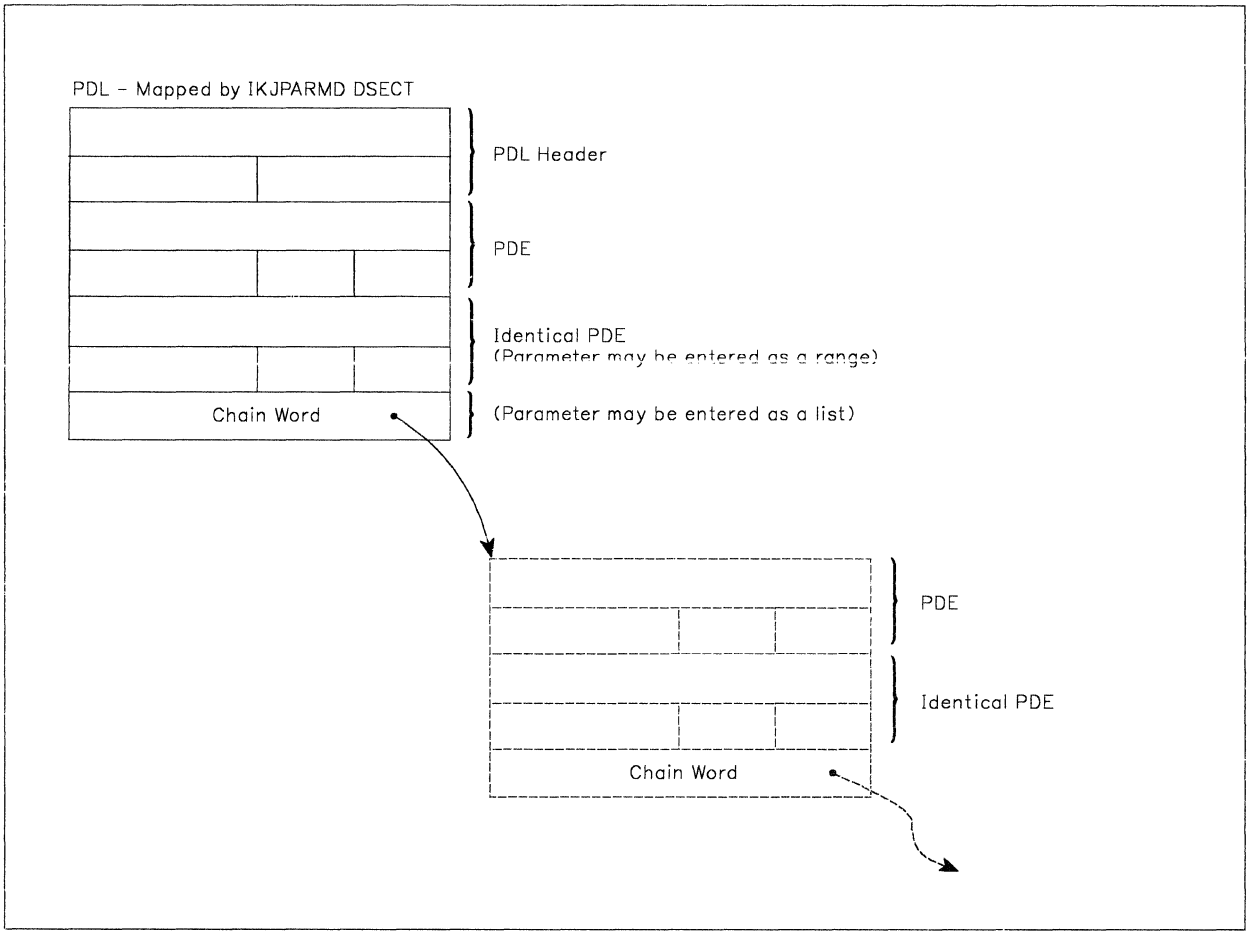


Figure 61. A PDL Showing PDEs that Describe LIST and RANGE Options

If you have specified both the LIST and the RANGE options in the parse macro instruction describing a positional operand, the TSO user 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 62 shows the PDL returned by the parse service routine if the user specifies a single operand.

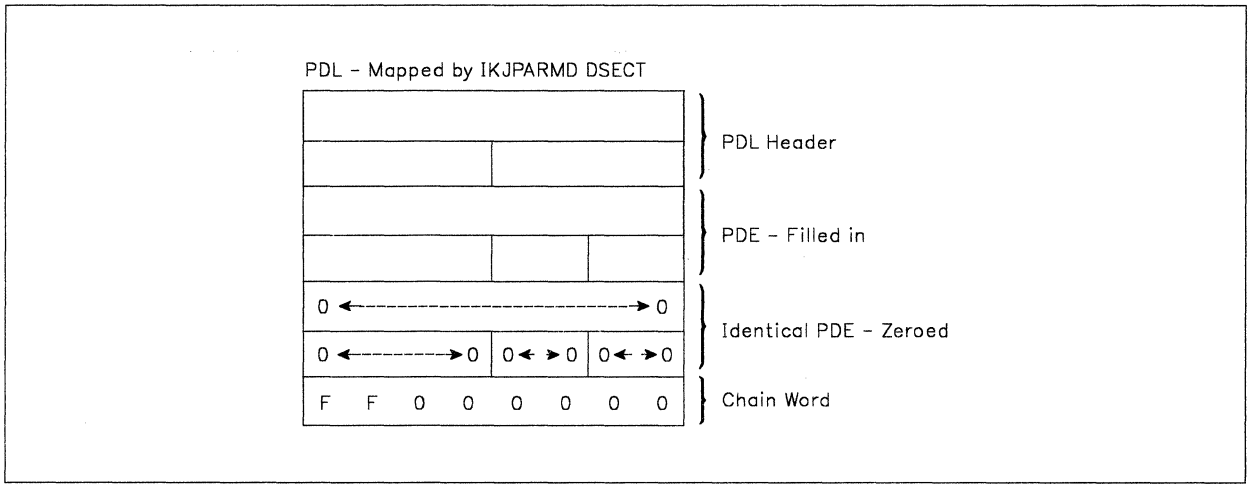


Figure 62. PDL - LIST and RANGE Acceptable, Single Operand Specified

As Figure 62 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 specified a single operand.

Figure 63 shows the PDL returned by the parse service routine if the user specifies a single range of the form:

```
operand:operand
```

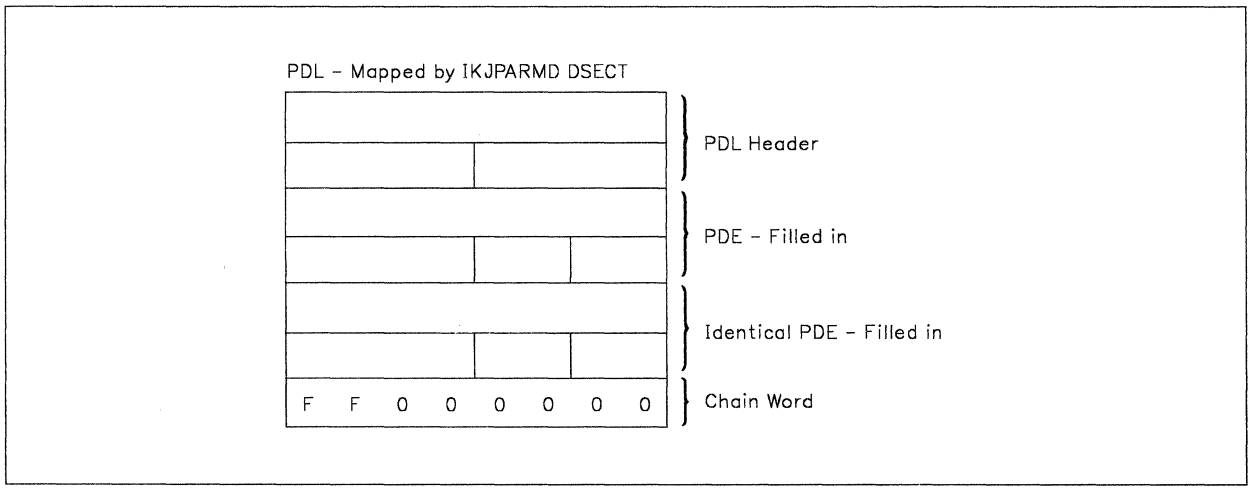


Figure 63. PDL - LIST and RANGE Acceptable, Single Range Specified

As Figure 63 shows, the parse service routine fills in both PDEs to describe the single RANGE operand specified 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 specified in the form of a list).

Figure 64 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,...)
```

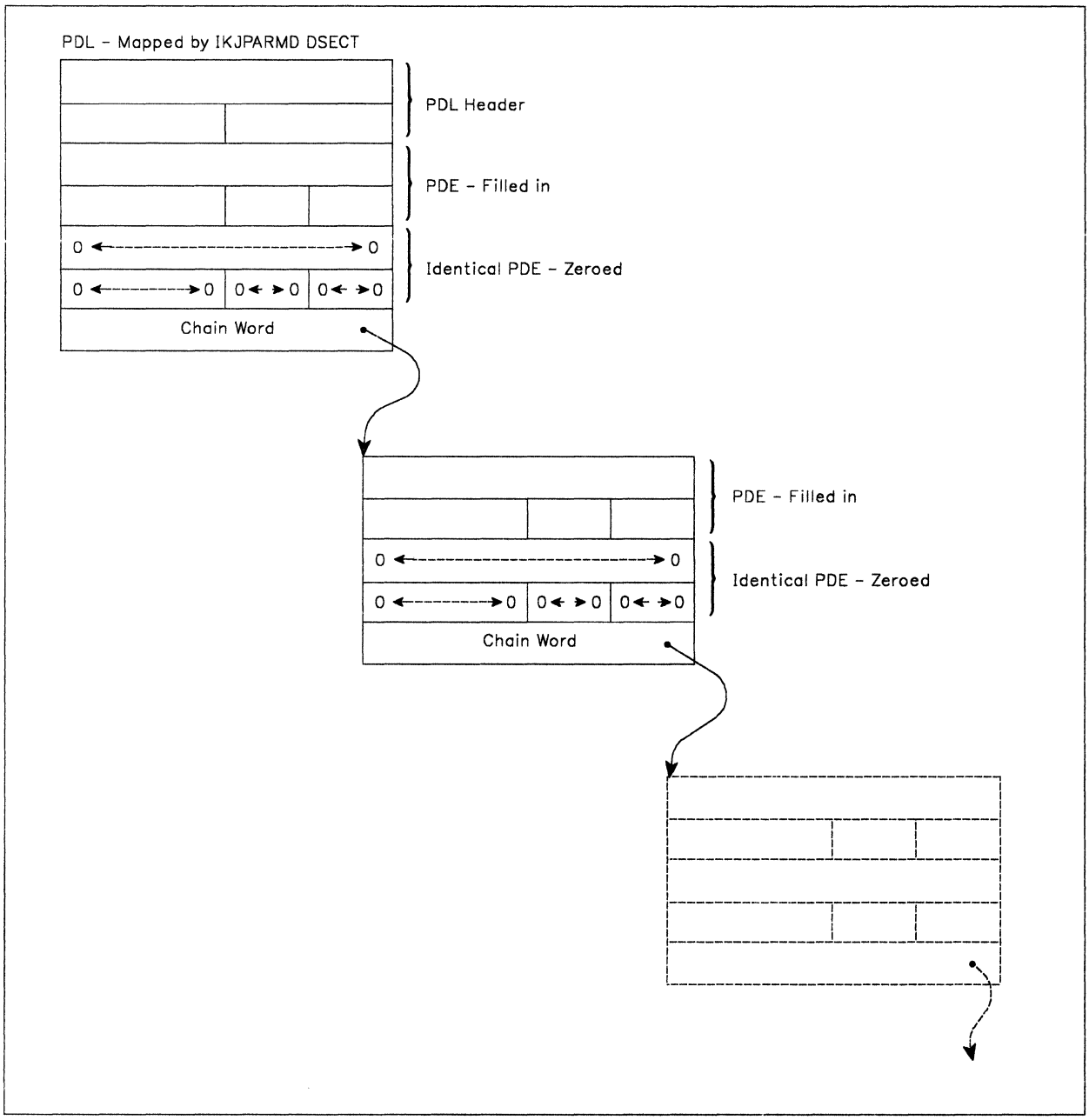


Figure 64. PDL - LIST and RANGE Acceptable, LIST Specified

As Figure 64 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 specified 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 specified.

Figure 65 shows the format of the PDL returned by the parse service routine if the user specifies a list of ranges in the form:

```
(operand:operand, operand:operand,...)
```

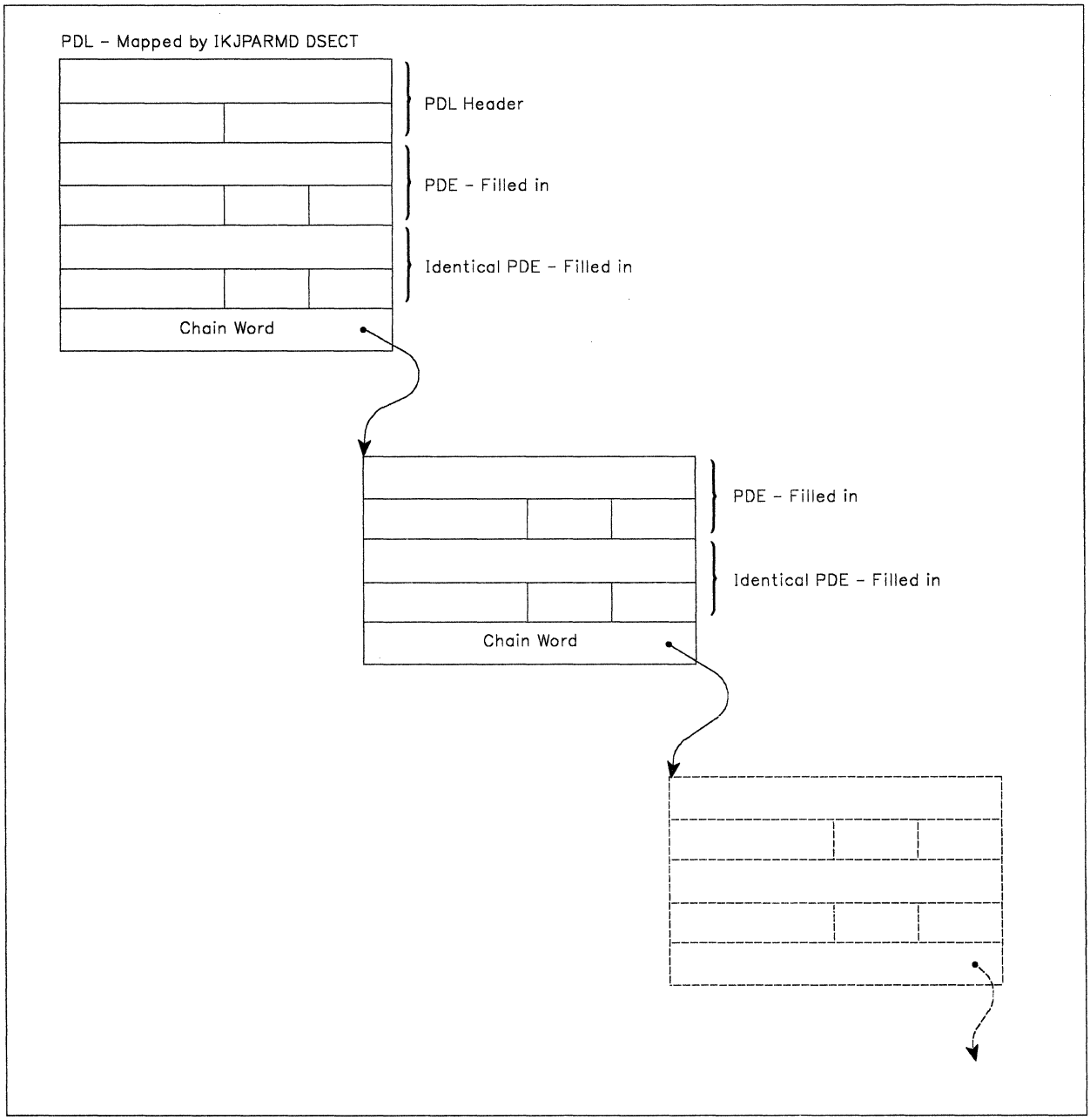


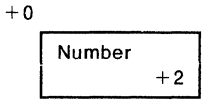
Figure 65. PDL - LIST and RANGE Acceptable, List of Ranges Specified

As Figure 65 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.

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:



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 the number of the IKJNAME macro instruction that corresponds to the keyword name specified.

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.

End of GENERAL-USE PROGRAMMING INTERFACE

Examples Using the Parse Service Routine

Example 1

This example expands upon "Example 1" on page 109. 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 Chapter 4, "Validating Command Operands" on page 13.

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

PROCESS	dsname	[ACTION NOACTION]

Figure 66 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='DATA SET NAME TO BE PROCESSED',             X
               VALIDCK=POSITCHK
ACTPCE   IKJKEYWD DEFAULT='NOACTION'
         IKJNAME  'ACTION'
         IKJNAME  'NOACTION'
         IKJENDP
```

Figure 66. Example 1 - Using Parse Macros to Describe Command Operand Syntax

Figure 67 shows the IKJPARMD DSECT created by the expansion of the parse macro instructions.

```
PRDSECT DSECT
         DS    2A
DSNPCE   DS    6A
ACTPCE   DS    H
```

Figure 67. Example 1 - The PRDSECT DSECT Created by Parse

If a TSO user specified the PROCESS command described in this example in the form:

```
PROCESS MYID.DATA NOACTION
```

the parse service routine would scan the command parameters, build a parameter descriptor list (PDL), place the address of the PDL into the fullword pointed to by the fifth word of the parse parameter list, and return to the calling program.

The calling routine uses the address of the PDL as a base address for the PRDSECT DSECT.

Figure 68 on page 143 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.

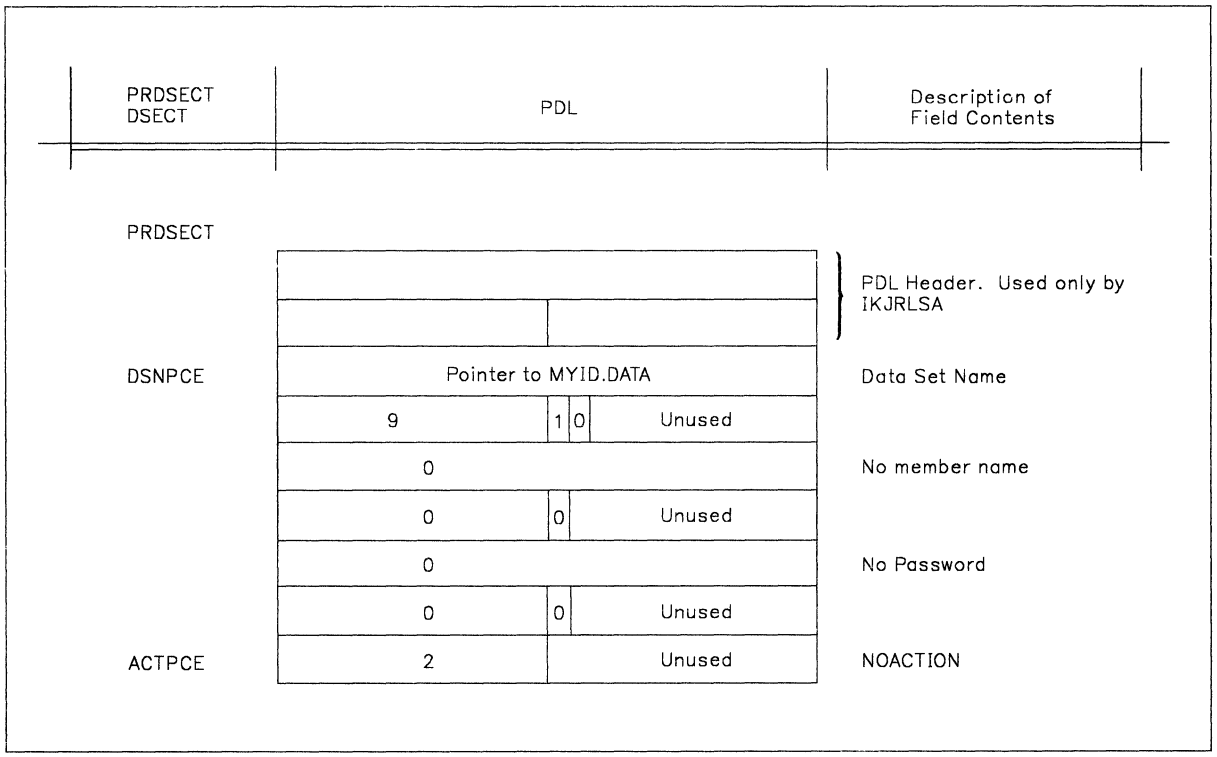


Figure 68. Example 1 - The PRDSECT DSECT and the PDL

Example 2

This example expands upon "Example 2" on page 110. 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:

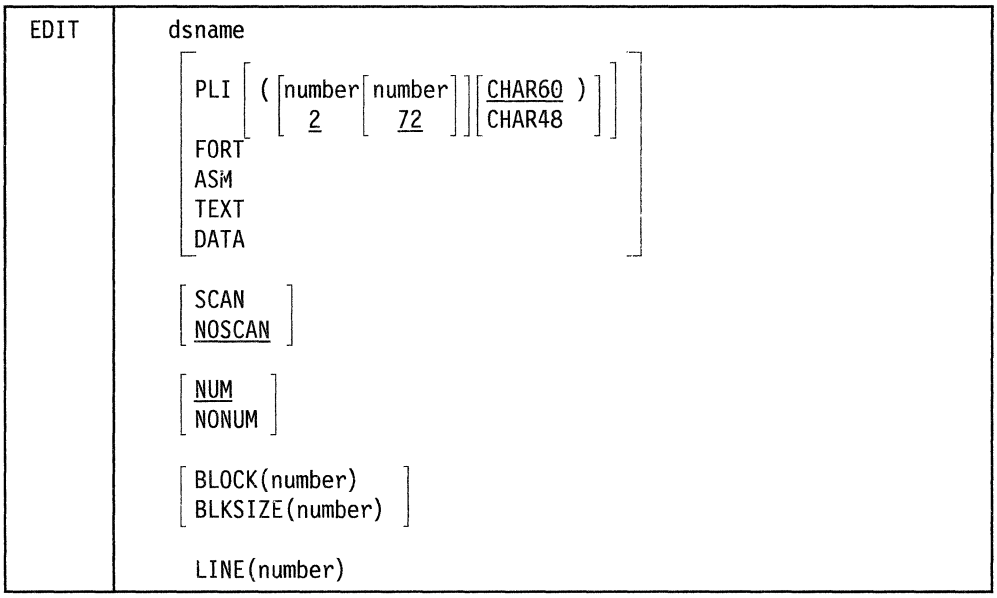


Figure 69 on page 145 shows the sequence of parse macro instructions that describe the syntax of this EDIT command to the parse service routine. The parse macro instructions used in this example build the parameter control list (PCL) describing the syntax of the EDIT command operands. The macro instructions 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.

```
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 69. Example 2 - Using Parse Macros to Describe Command Operand Syntax

Figure 70 shows the IKJPARMD DSECT created by the expansion of the parse macro instructions.

```
IKJPARMD DSECT
         DS    2A
DSNAM    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 70. Example 2 - The IKJPARMD DSECT Created by Parse

If a user specified the EDIT command described in this example in the form:

```
EDIT SYSFILE/X PL1(3) NONUM BLOCK(160)
```

the parse service routine would scan the command parameters, build a parameter descriptor list (PDL), place the address of the PDL into the fullword pointed to by the fifth word of the parse parameter list, and return to the calling program.

The calling routine uses the address of the PDL as a base address for the IKJPARMD DSECT.

Figure 71 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.

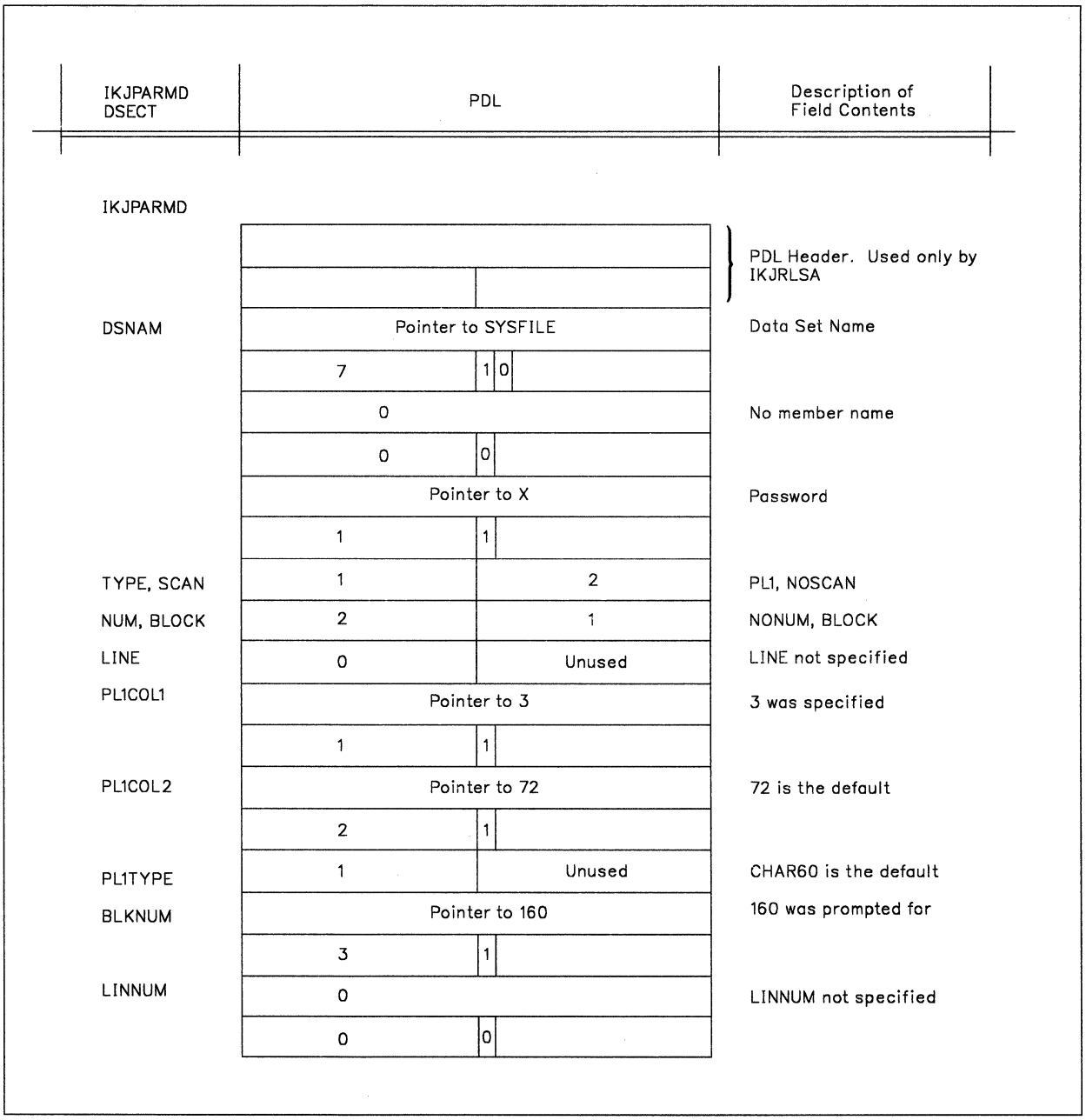


Figure 71. Example 2 - The IKJPARMD DSECT and the PDL

Example 3

This example expands upon "Example 3" on page 111. 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:

COMMAND	OPERANDS
AT	{ stmt (stmt-1,stmt-2,...) stmt-3:stmt-4 } (cmd chain) COUNT(integer)

Figure 72 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 use to map the parameter descriptor list returned by the parse service routine. In this example, the name of the DSECT is PARSEAT.

```
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 72. Example 3 - Using Parse Macros to Describe Command Operand Syntax

Figure 73 shows the PARSEAT DSECT created by the expansion of the parse macro instructions.

```
PARSEAT  DSECT
         DS    2A
STMTPCE  DS    11A
POSITPCE DS    2A
KEYPCE   DS    H
IDENTPCE DS    2A
```

Figure 73. Example 3 - The PARSEAT DSECT Created by Parse

In this example, if the user specified the AT command as:

```
AT 200.3 (LIST ALL) COUNT(3)
```

the parse service routine would build a parameter descriptor list (PDL) and place the address of the PDL into the fullword pointed to by the fifth word of the parse parameter list.

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 74 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.

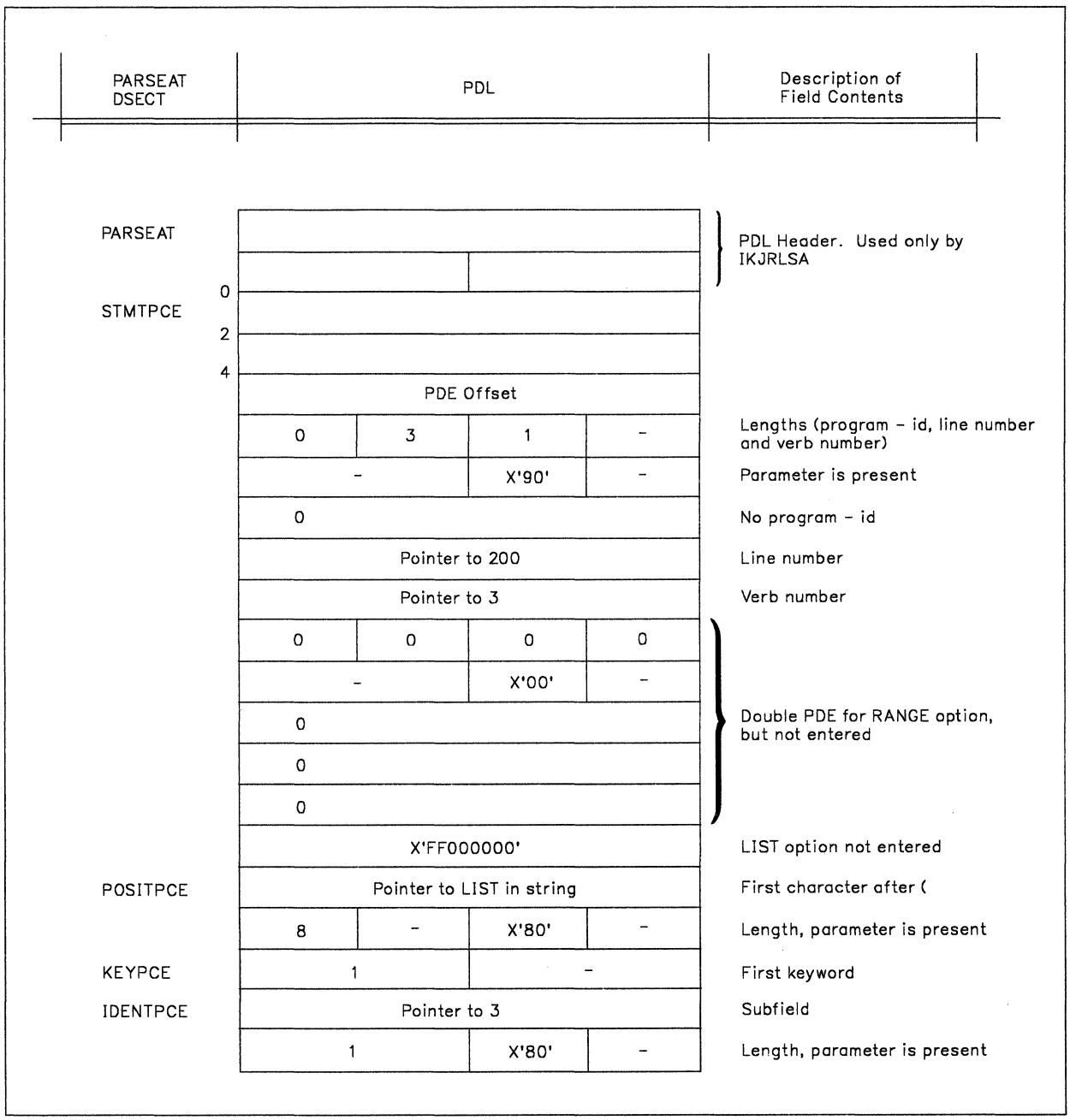


Figure 74. Example 3 - The PARSEAT DSECT and the PDL

Example 4

This example expands upon "Example 4" on page 112. 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:

COMMAND	OPERANDS
LIST	symbol PRINT(symbol)

Figure 75 on page 149 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.

```
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 75. Example 4 - Using Parse Macros to Describe Command Operand Syntax

Figure 76 shows the PARSELST DSECT created by the expansion of the parse macro instructions.

```
PARSELST DSECT
         DS    2A
VARPCE   DS    5A
SUBPCE   DS    15A
KEYPCE   DS    H
PRINTSUB DS    11A
```

Figure 76. Example 4 - The PARSELST DSECT

In this example, if the user specified the LIST command as:

```
list a of b in c(1) print(d)
```

the parse service routine would build a parameter descriptor list (PDL) and place the address of the PDL into the fullword pointed to by the fifth word of the parse parameter list.

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 77 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.

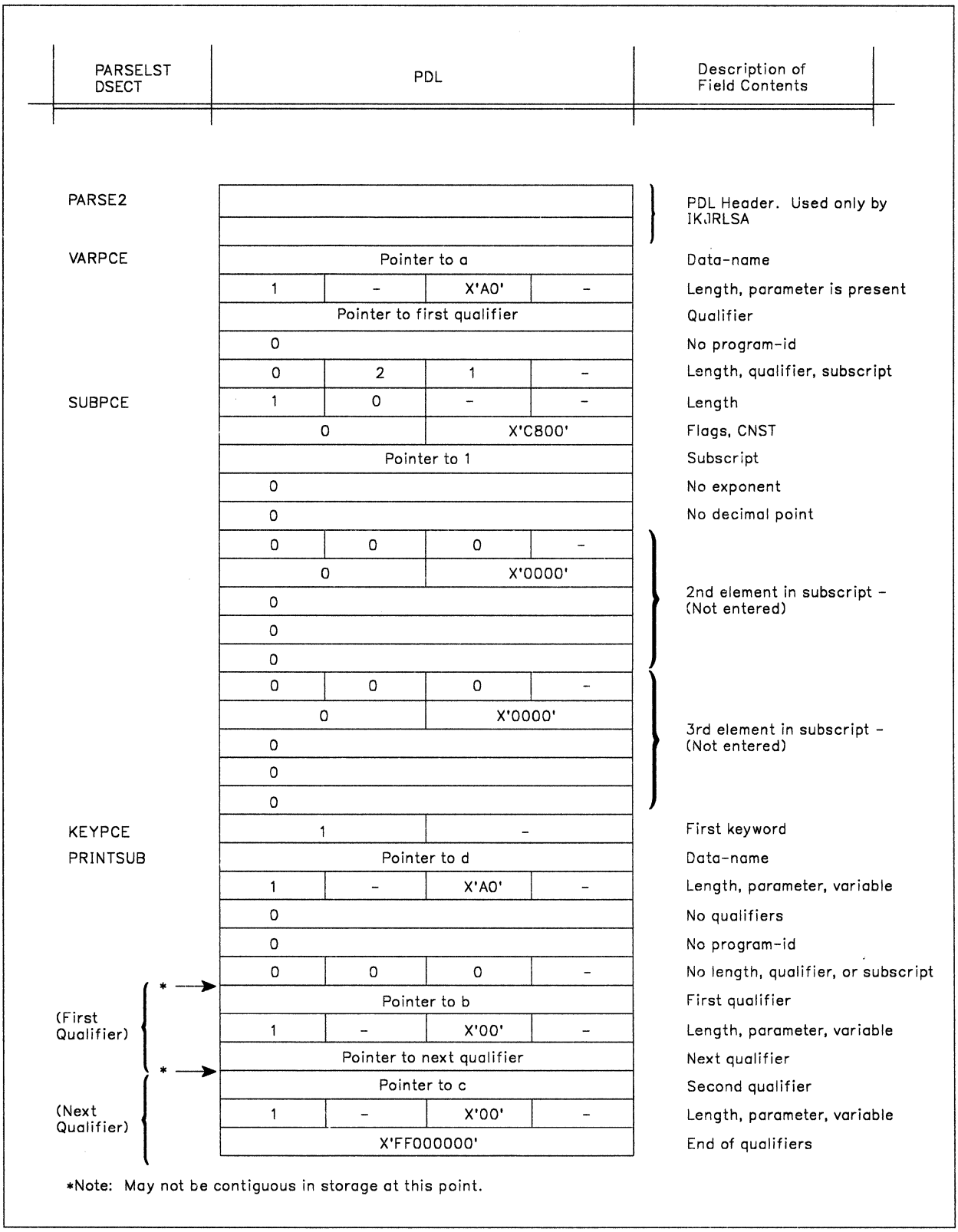


Figure 77. Example 4 - The PARSELST DSECT and the PDL

Example 5

This example expands upon "Example 5" on page 113. 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:

COMMAND	OPERANDS
WHEN	{addr / expression} (subcommand chain)

Figure 78 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.

```
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 78. Example 5 - Using Parse Macros to Describe Command Operand Syntax

Figure 79 shows the PARSELST DSECT created by the expansion of the parse macro instructions.

```
PARSEWHN DSECT
         DS    2A
OPER     DS    2A
SYMBOL1  DS    5A
OPERATOR DS    2A
SYMBOL2  DS    5A
ADDR1    DS    5A
LASTONE  DS    2A
```

Figure 79. Example 5 - The PARSEWHN DSECT

In this example, if the user specified the WHEN command as:

```
WHEN (A EQ B) (LIST B)
```

the parse service routine would build a parameter descriptor list (PDL) and place the address of the PDL into the fullword pointed to by the fifth word of the parse parameter list.

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.

Figure 80 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.

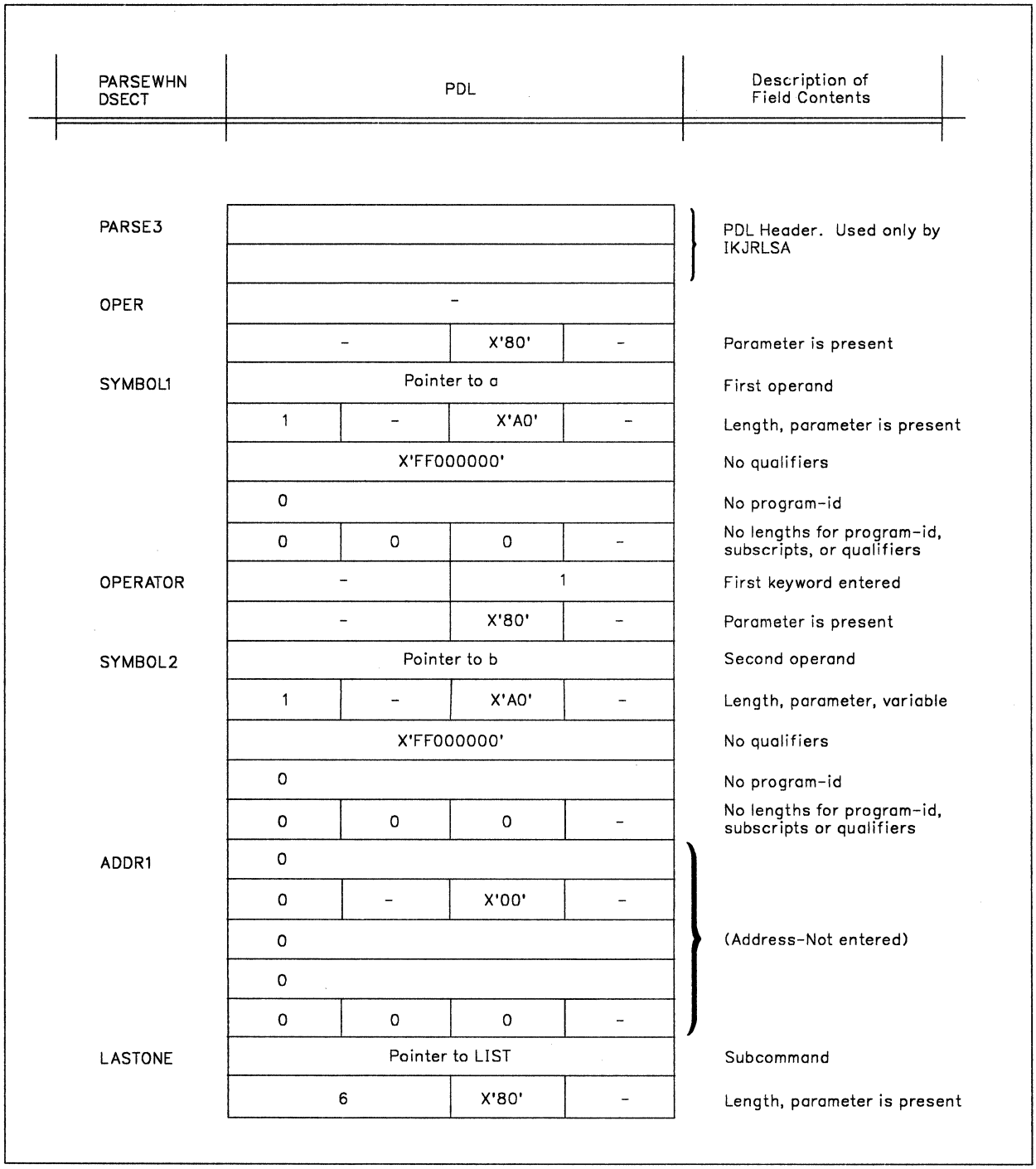


Figure 80. Example 5 - The PARSEWHN DSECT and PDL

Chapter 14. Using the TSO I/O Service Routines

This chapter describes how to use the TSO I/O service routines, STACK, GETLINE, PUTLINE, and PUTGET, to process I/O.

Functions of the I/O Service Routines

If you write your own command processors, use the the I/O service routines to process I/O. Figure 81 describes the function of each of the I/O service routines.

Figure 81. The TSO I/O Service Routines

Service Routine	Function
STACK	Establishes and changes the source of input.
GETLINE	Obtains a line of input, other than commands and subcommands.
PUTLINE	Writes a line to the output data set.
PUTGET	Writes a message to the output data set and obtains a line of input in response.

The I/O service routines, STACK, GETLINE, PUTLINE, and PUTGET, offer the following features:

- They allow your command processor to direct requests for input to an in-storage list or data set.
- They provide a message formatting facility that allows you to insert text segments into a basic message format.
- 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 by using the list and execute forms of 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 instruction. 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.

You can use the DELETE macro instruction to release the storage area occupied by the load module when you have finished with your I/O. The I/O service routines are contained in the IKJPTGTR load module.

Addressing Mode Considerations

Your command processor must invoke the I/O service routines in 24-bit addressing mode. These routines execute in 24-bit addressing mode, and are loaded below 16 megabytes in virtual storage. All input to the I/O service routines must reside below 16 megabytes in virtual storage.

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 Service Routines" on page 50. 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 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. Figure 82 describes the format of the IOPL.

Figure 82. The Input/Output Parameter List

Number of Bytes	Field	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.
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	DSECT Name	Parameter Block
STACK	IKJSTPB	The STACK parameter block
GETLINE	IKJGTPB	The GETLINE parameter block
PUTLINE	IKJPTPB	The PUTLINE parameter block
PUTGET	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 maintained by the I/O service routines. The first element of the input stack is initialized by the terminal monitor program (TMP), and afterward, cannot be changed or deleted. The IBM-supplied TMP initializes this first element to indicate that the input stream, controlled by the SYSTSIN DD statement in the TSO user's JCL, is the current input source. 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 build an alternate input stack by using the STACK macro instruction. To build an alternate input stack, do the following:

1. Preserve the current input stack by saving the original value of the ECTIOWA field. The ECTIOWA field is contained in the event control table (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.

Note that programs cannot build input stacks directly; they must invoke the STACK service routine to create a valid input stack.

In the sections that follow, the following topics are discussed:

- The list and execute forms of the STACK macro instruction
- The sources of input
- The STACK parameter block
- The list source descriptor
- Return codes from 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. The DATASET, STORAGE, and DELETE operands set bits in the STACK parameter block. These bit settings indicate to the STACK service routine which options 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 83 shows the list form of the STACK macro instruction; each of the operands is explained following the figure.

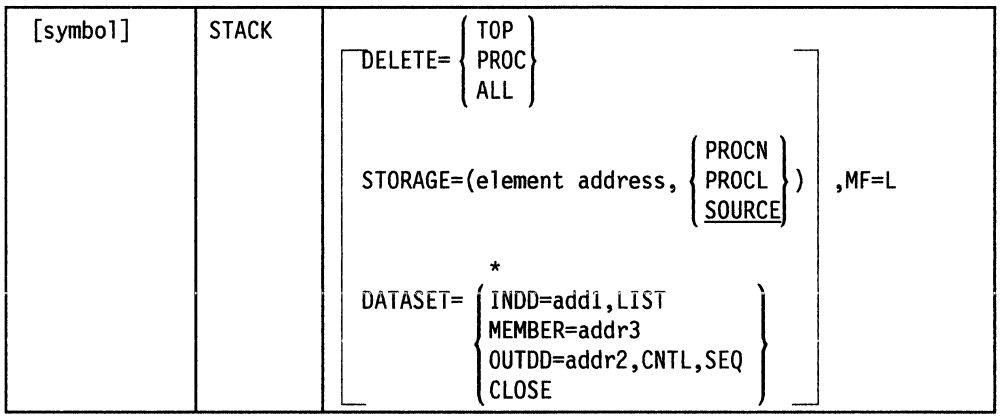


Figure 83. The List Form of the STACK Macro Instruction

DELETE =
Deletes an element or elements from the input stack. TOP, PROC, or ALL further defines the element to be deleted.

TOP
Deletes the topmost element (the element most recently added to the input stack).

PROC
Deletes the current procedure element from the input stack. If the top element is not a PROC element, deletes all elements down to, and including, the first PROC element.

ALL
Deletes all elements, except the bottom or first element, from 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 megabytes in virtual storage. See "Building the List Source Descriptor (LSD)" on page 163 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.

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 read from the command procedure is written to the output data set.

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
Supports dataset I/O for TSO commands to include reading from a SYSIN data set and writing to a SYSOUT dataset. To use the dataset function, the input and output files passed to the STACK service routine must be preallocated, either by a command processor that invokes dynamic allocation or a user-supplied DD statement.

* Specifies that STACK use the bottom element in the input stack for I/O operations.

INDD = addr1
Specifies the input file name.

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.

SEQ
Indicates to dataset I/O that sequence numbers should not be removed.

The Execute Form of the STACK Macro Instruction

Use the execute form of the STACK macro instruction to perform the following functions:

- To set up the input/output parameter list (IOPL).
- To 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.
- To pass control to the STACK service routine, which modifies the input stack.

The operands 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. The DATASET, STORAGE, and DELETE operands set bits in the STACK parameter block. These bit settings indicate to the STACK service routine which options you want.

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.

The other operands and their sublists are optional because they can be supplied by the list form of the macro instruction.

You are not required to specify the ENTRY operand on the macro instruction. If you do not specify it, a LINK macro instruction will be generated to invoke the STACK I/O service routine.

Figure 84 shows the execute form of the STACK macro instruction; each of the operands is explained following the figure.

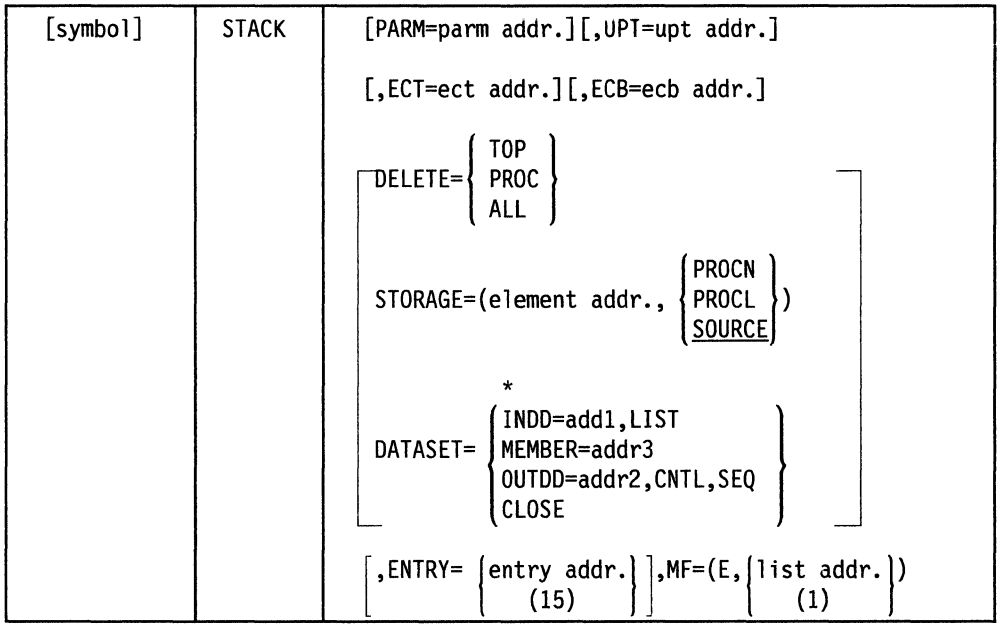


Figure 84. The Execute Form of the STACK Macro Instruction

PARM = parm addr

Specifies the address of the 5-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 will be 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 one 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 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 will be placed in the IOPL.

ECB = ecb addr
Specifies the address of an event control block (ECB). This address will be 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.

DELETE
Deletes one or more elements from the input stack. TOP, PROC, or ALL specifies which element(s).

TOP
Deletes the topmost element (the element most recently added to the input stack).

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.

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 163 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 read from the command procedure is written to the output data set.

DATASET
Supports dataset I/O for TSO commands to include reading from a SYSIN dataset and writing to a SYSOUT dataset. To use the dataset function, the input and output files passed to the STACK service routine must be preallocated, either by a command processor that invokes dynamic allocation or a user-supplied DD statement.

* Specifies that STACK use the bottom element on the input stack for I/O operations.

INDD = addr1
Specifies the input file name.

LIST
Lists the input from the input stream.

MEMBER = addr3
Specifies the 8-character member name for the input file.

OUTDD = addr2
Specifies the output file name.

CNTL
The output line has its own control character.

SEQ
Indicates to dataset I/O that sequence numbers should not be removed.

CLOSE
Closes the data control blocks (DCBs) of the bottom element of the input stack.

ENTRY = entry address or (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 will be 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 will be 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.

Sources of Input

There are two types of input sources, a data set and an in-storage list.

Data Set: The terminal monitor program (TMP) initializes the first element of the input stack as a data set element. This element indicates that the input stream, which is controlled by the SYSTSIN DD statement in the TSO user's JCL, is the source of input. All input and output requests through GETLINE, PUTLINE, and PUTGET are read from the input stream and written to the output data set. The output data set is controlled by the SYSTSPRT DD statement in the TSO user's JCL.

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.

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. MODE messages, which are normally sent to the output data set by a command or subcommand, are not sent; instead, a command is obtained from the in-storage list.

If the PROCL option is specified, the command is written to the output data set 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. 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. The IBM-supplied terminal monitor program (TMP) shares subpool 78 with command processors. 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 shared 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.

For an example showing how to use the STACK service routine to specify an in-storage list as the input source, see Figure 90 on page 168.

Building the STACK Parameter Block (STPB)

When the list form of the STACK macro instruction expands, it builds a five 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. Figure 85 describes the contents of the STPB.

Figure 85. The STACK Parameter Block

Number of Bytes	Field	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.
	 xxxx	Reserved bits.
1	none	Element code: A flag byte describing the element to be added to the input stack.
	x...	Reserved.
	.1..	An in-storage element.
	..1.	Input DD name present.
	...1	Output DD name present.
	.... xx..	Reserved.
	.... ..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.
1		Reserved
1	none	DATASET operation.
	xxxx x...	Reserved.
	.... .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 DELETE has been specified, this field will contain zeros.
4	STPBINDD	Pointer to input DD name.
4	STPBODDN	Pointer to output DD name.
4	STPBMBRN	Pointer to membername.

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. The LSD must reside below 16 megabytes in virtual storage.

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 the shared subpool designated by the terminal monitor program; the IBM-supplied TMP shares subpool 78 with the command processors. 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. Figure 86 describes the contents of the LSD.

Figure 86. The List Source Descriptor

Number of Bytes	Field	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.

Return Codes from STACK

When it returns to the program which invoked it, the STACK service routine will provide one of the following return codes in general register 15:

Figure 87. 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 invalid.
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).

End of GENERAL-USE PROGRAMMING INTERFACE

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 88 describes this condition.

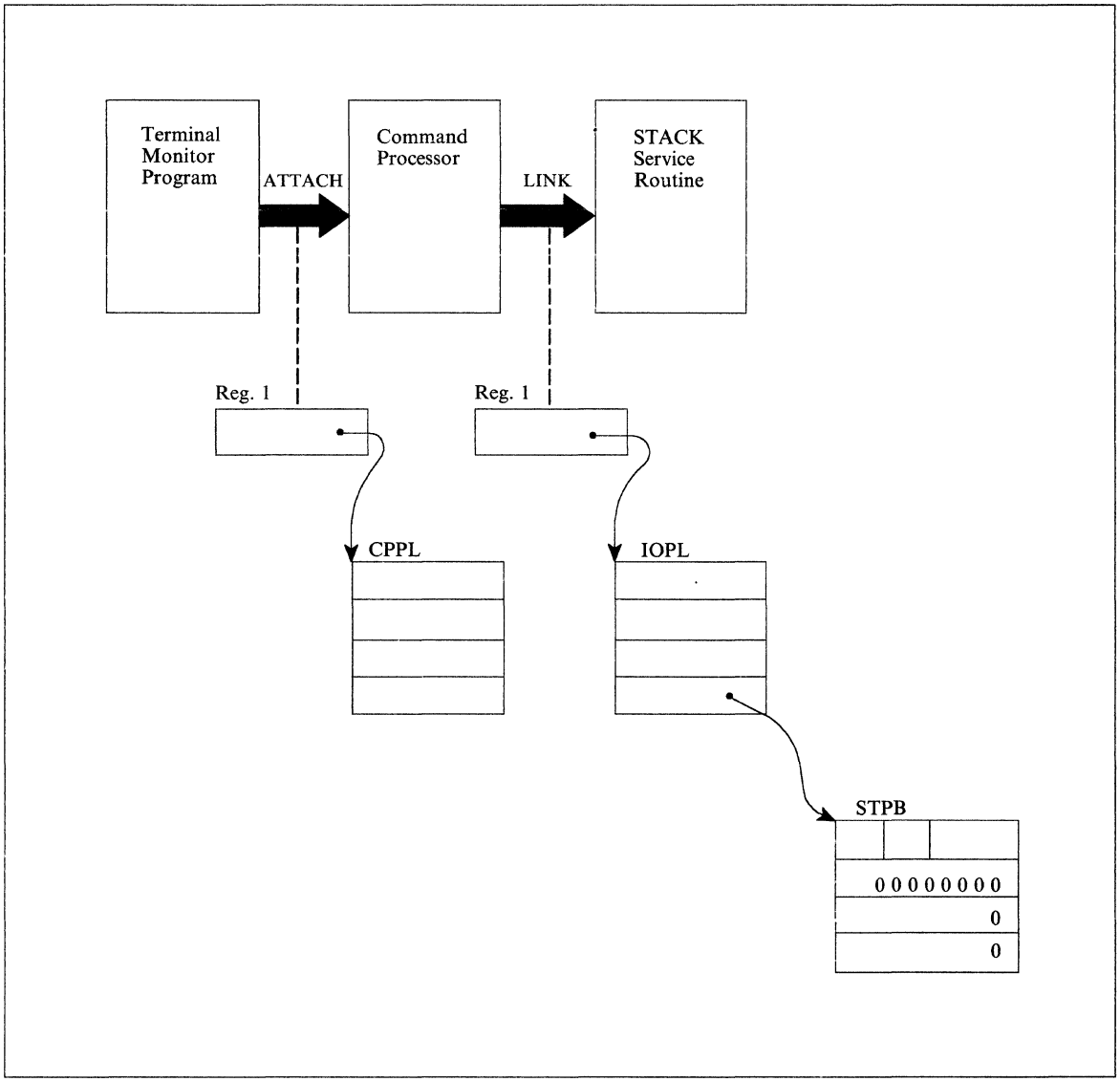


Figure 88. STACK Control Blocks: No In-Storage List

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).

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 89.

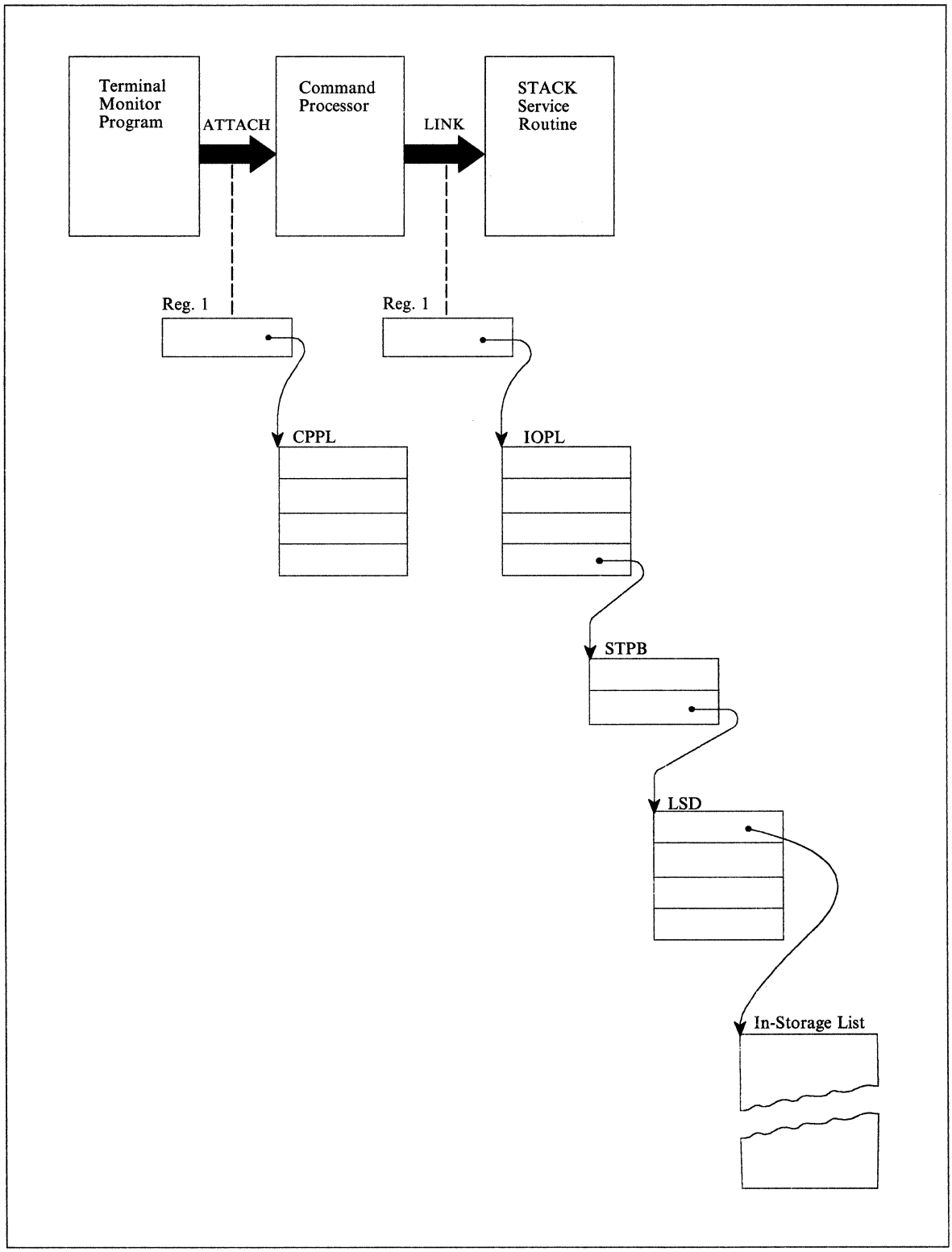


Figure 89. STACK Control Blocks: In-Storage List Specified

Example Using STACK

Figure 90 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 90 (Part 1 of 2). Example of STACK Specifying an In-storage List as the Input Source

```
*            .
*            .
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'CMD1 OPA OPB OPC'
         DC    X'00180000'
         DC    C'CMD2 OPTA OPTB OPTC '
         DC    X'00240000'
         DC    C'COMMAND     NOMSGID    NOPROMPT'
         DC    X'00100000'
         DC    C'CMD3 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
```

Figure 90 (Part 2 of 2). Example of STACK Specifying an In-storage List as the Input Source

Using GETLINE to Get a Line of Input

Use the GETLINE macro instruction to obtain all input lines other than commands and subcommands. Commands and subcommands should be obtained with the PUTGET macro instruction.

When a GETLINE macro instruction is executed, a line is obtained from the current source of input, which is either a data set or an in-storage list. 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 operand sets bits in the GETLINE parameter block to indicate to the GETLINE service routine which options you want performed.

Figure 91 shows the list form of the GETLINE macro instruction; each of the operands is explained following the figure.

[symbol]	GETLINE	[INPUT=(ISTACK [,LOGICAL])] ,MF=L [,PHYSICAL]

Figure 91. 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, LOGICAL and PHYSICAL. ISTACK and LOGICAL are the default values.

ISTACK

Obtain an input line from the currently active input source indicated by the input stack.

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.

MF=L

Indicates that this is the list form of the macro instruction.

The Execute Form of the GETLINE Macro Instruction

Use the execute form of the GETLINE macro instruction to perform the following functions:

- To set up the input/output parameter list (IOPL).
- To 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.
- To 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 operand sets bits in the GETLINE parameter block. These bit settings indicate to the GETLINE service routine which options you want performed.

Figure 92 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 / ,PHYSICAL})] [,ENTRY={entry address / (15)}],MF=(E,{list address / (1)})

Figure 92. 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, LOGICAL, and PHYSICAL. ISTACK and LOGICAL are the default values.

ISTACK
Obtains an input line from the currently active input source indicated by the input stack.

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.

ENTRY = entry address or (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.

Sources of Input

There are two sources of input provided; they are a data set, and an in-storage list.

Data Set: Input comes from a data set if you have specified the current element of the input stack by including the ISTACK operand in the GETLINE macro instruction, and the current element is a data set element.

If you specify a data set as the input source, you have the option of requesting the GETLINE service routine to process the input as a logical or physical line by including the LOGICAL or the PHYSICAL operand in the macro instruction. 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 input data set and to concatenate that line with the line previously obtained. The continuation mark is overlaid with the first character of the new line.

In-Storage List: If the top element of the input stack is an in-storage list, 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.

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. See "Return Codes from GETLINE" on page 175 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. Figure 93 describes the contents of the GTPB.

Figure 93. The GETLINE Parameter Block

Number of Bytes	Field	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.
	xx.. xxxx	Reserved bits.
	Byte 2	
	xxxx xxxx	Reserved.
2		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. The input buffer belongs to the command processor that issued the GETLINE macro instruction. The buffers returned by GETLINE are automatically freed when your command processor relinquishes control. You can free the input buffer with the FREEMAIN macro instruction after you have processed or copied the input line.

Regardless of the source of input, 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 94 shows the format of the input buffer returned by the GETLINE service routine.

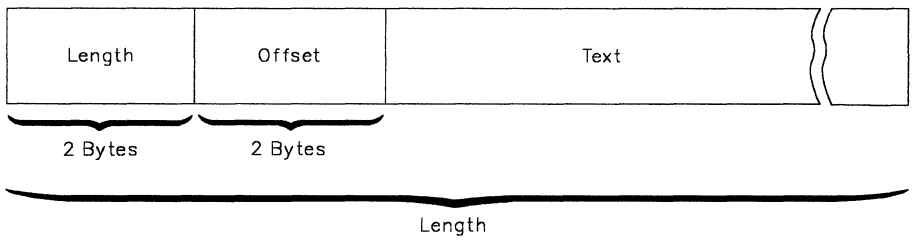


Figure 94. Format of the GETLINE Input Buffer

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.

Return Codes from GETLINE

When it returns to the program that invoked it, the GETLINE service routine returns one of the following codes in general register 15:

Return Code Dec(Hex)	Meaning
4(4)	GETLINE has completed successfully. The line was returned from an in-storage list.
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)	Invalid parameters were passed to the GETLINE service routine.
24(18)	A conditional GETMAIN was issued by GETLINE for input buffers and there was not sufficient space to satisfy the request.

Figure 95. Return Codes from the GETLINE Service Routine

End of GENERAL-USE PROGRAMMING INTERFACE

Figure 96 shows the GETLINE control block structure after the GETLINE service routine has returned an input line.

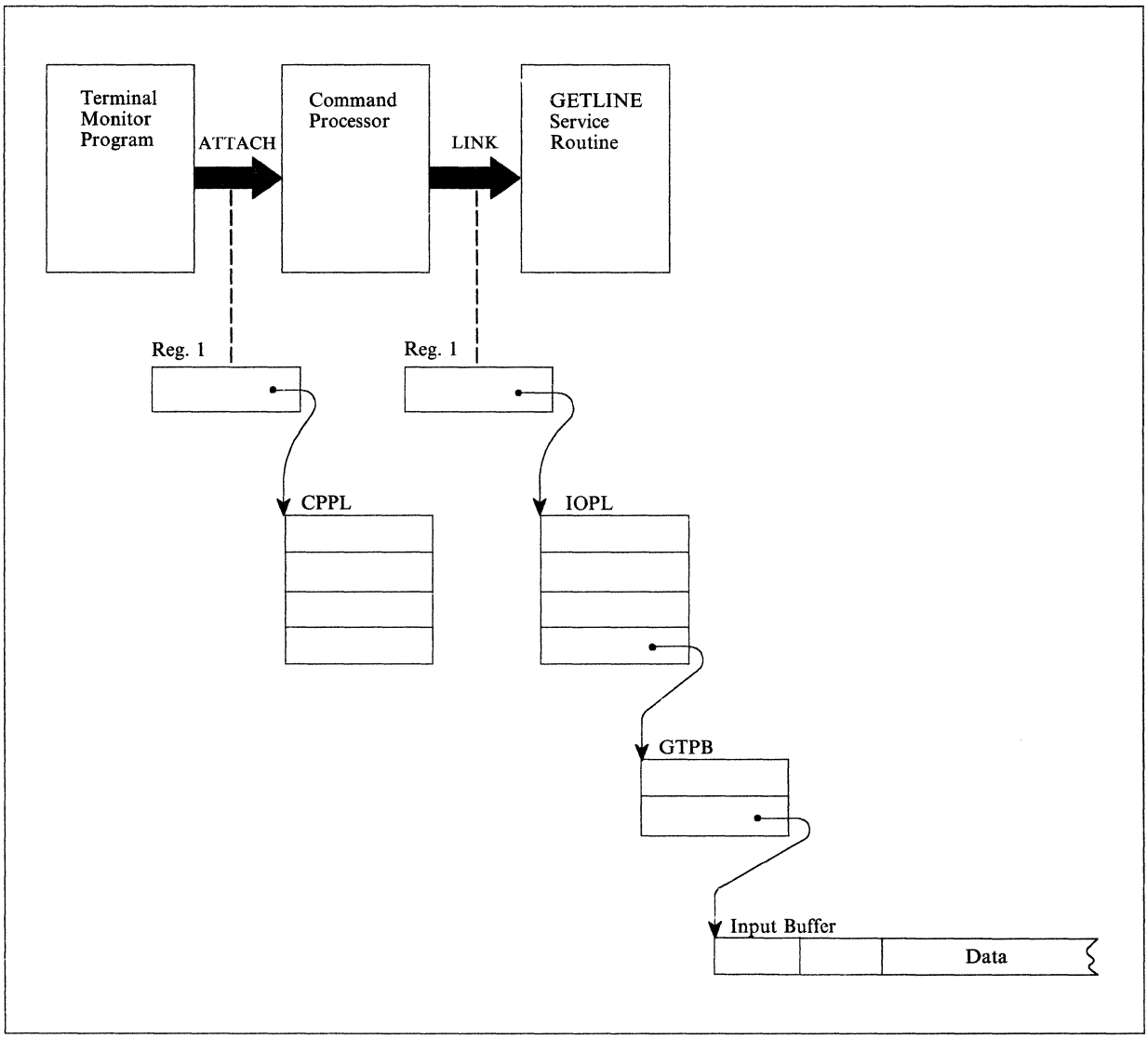


Figure 96. GETLINE Control Blocks - Input Line Returned

Using PUTLINE to Write a Line to the Output Data Set

Use the PUTLINE macro instruction to prepare a line and write it to the output data set. Use PUTLINE to put out data lines and informational message lines; use PUTGET to put out lines and obtain a line of input.

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) and the type of processing to be performed on the line (format only, second level informational message chaining, text insertion).

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 output line address is required for each issuance of the PUTLINE macro instruction, but it can be supplied in the execute form of the macro instruction.

The other operands and sublists are optional because you can supply them in the execute form of the macro instruction, or they will be supplied by the macro expansion if you want the default values. Figure 97 shows the list form of the PUTLINE macro instruction; each of the operands is explained following the figure.

[symbol]	PUTLINE	[OUTPUT=(output address {,TERM / ,FORMAT} {,SINGLE / ,MULTLVL / ,MULTLIN} {,INFOR / ,DATA})] ,MF=L

Figure 97. The List Form of the PUTLINE Macro Instruction

OUTPUT = output address
Indicates that an output line is to be written to the output data set. 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 and DATA. The default values are TERM, SINGLE, and INFOR.

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 output data set. 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 output data set by the PUTLINE service routine.

TERM
Write the line out to the output data set.

FORMAT
The output request is only to format a single message and not to put the message out to the output data set. The PUTLINE service routine returns the address of the formatted line by placing it in the third word of the PUTLINE parameter block.

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
The output line is an informational message.

DATA
The output line is a data 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 write a line or lines to the output data set, 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:

- To set up the input/output parameter list (IOPL).
- To 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.
- To 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 operand and its sublist operands initialize the PUTLINE parameter block. The PUTLINE parameter block is referenced 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 operand 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 98 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})]
                                             {,FORMAT}{,MULTLVL}{,DATA }
                                                      {,MULTLIN}

                     [,ENTRY={entry address}],MF=(E{,list address})
                             {    (15)     }       {    (1)     }
```

Figure 98. 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 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 and DATA. The default values are TERM, SINGLE, and INFOR.

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 output data set. 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 output data set by the PUTLINE service routine.

TERM

Write the line out to the output data set.

FORMAT

The output request is only to format a single message and not to put the messages out to the output data set. The PUTLINE service routine returns the address of the formatted line by placing it in the third word of the PUTLINE parameter block.

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

The output line is an informational message.

DATA

The output line is a data line.

ENTRY = entry address or (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. Since 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. Figure 99 describes the contents of the PTPB.

Figure 99. The PUTLINE Parameter Block

Number of Bytes	Field	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	Reserved bits.
	Byte 2	
	..1.	The format only function was requested.
	xx.x xxxx	Reserved bits.
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), and whether the line is to be written to the output data set (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 output data set. PUTLINE does not format the line or process it in any way; it merely writes the line, as it appears, to the output data set. 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 writes to the output data set as one logical line. PUTLINE accepts single line data in the format shown in Figure 100.

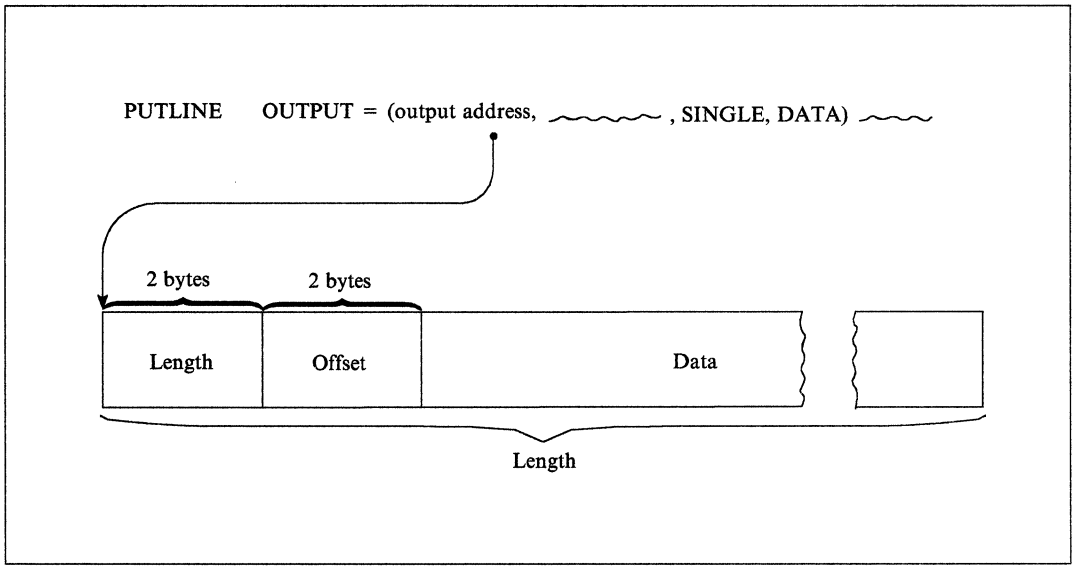


Figure 100. PUTLINE Single Line Data Format

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 102 on page 184 is an example of the code that could be used to write a single line of data to the output data set 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.

- **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 output data set. By specifying multiline data (MULTLIN) in the PUTLINE macro instruction, you can put out several variable length, non-contiguous lines 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 101 shows the format of PUTLINE multiline data.

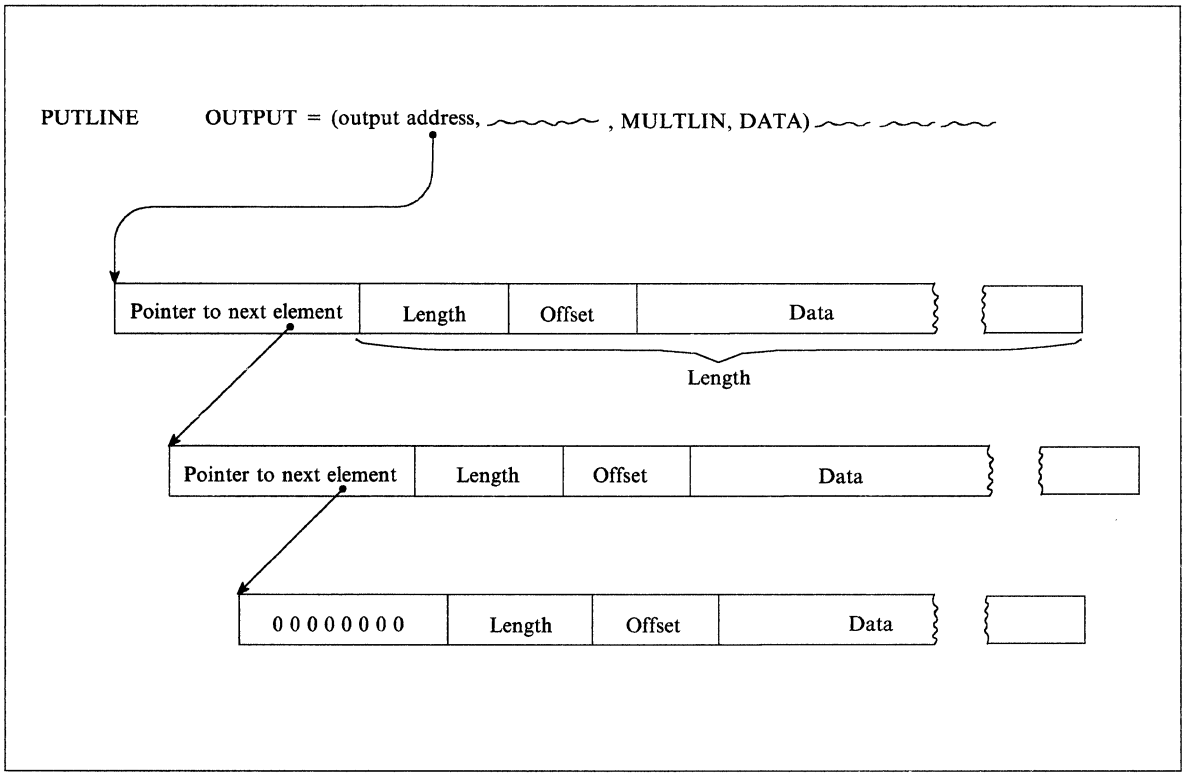


Figure 101. PUTLINE Multiline Data Format

Each of the forward-chain pointers points to the next data line to be written to the output data set. 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 will place this multiline data address into the second word of the PUTLINE parameter block.

Figure 103 on page 185 is an example of the code required to write multiple lines of data to the output data set 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 instruction, 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 output data set. It is used solely to pass output to the TSO user; no input from the input stream is required after an informational message.

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 output data set 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 output data set, and one or more message segments to be formatted and placed on an internal chain in shared subpool 78. This internal chain is written to the output data set. Use the MULTLVL operand on the PUTLINE macro instruction to indicate that a multilevel message is to be written to the output data set.

End of GENERAL-USE PROGRAMMING INTERFACE

```
* 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 AND TO BUILD THE IOPL.
*
         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 102. 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 EXECPT 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 103. 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 zeros or an offset value if you use the text insertion facility. See "Using the PUTLINE Text Insertion Function" on page 188 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. Figure 104 shows the format of the output line descriptor.

Figure 104. The Output Line Descriptor (OLD)

Number of Bytes	Field	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 instruction expands, it places the address of the output line descriptor into the second word of the PUTLINE parameter block.

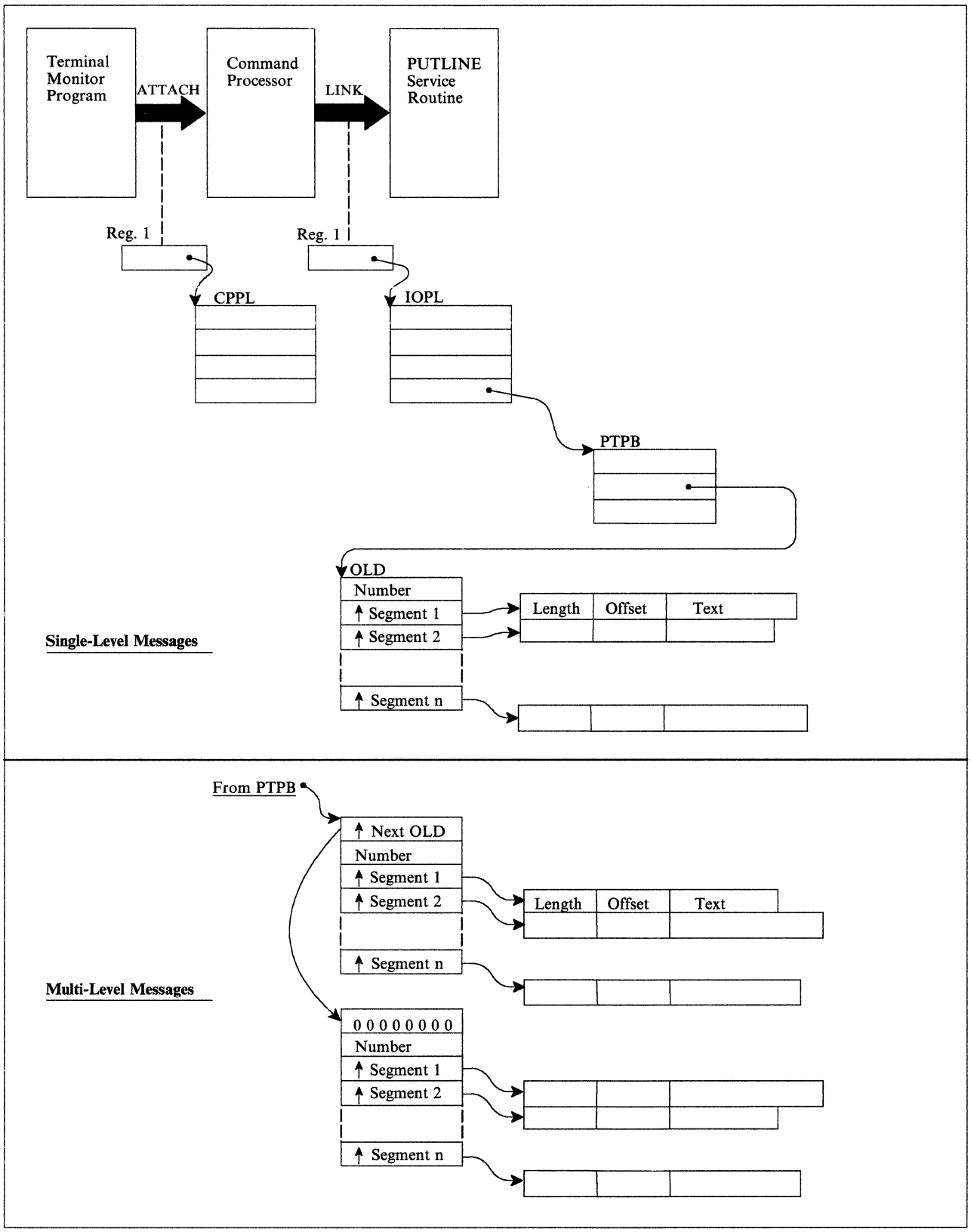


Figure 105. Control Block Structures for PUTLINE Messages

PUTLINE Message Line Processing

The PUTLINE service routine allows you to specify message identifiers that precede the text of the message. 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. If the message segment does not contain at least one blank, PUTLINE will return a code of 12, which indicates invalid parameters, in register 15.

In addition to writing a message out to the output data set, the PUTLINE service routine provides the following additional functions for message line processing when the INFOR operand is specified:

- Text insertion
- Formatting only
- Second level informational chaining

Figure 106 shows the output message types for which these PUTLINE service routine functions can be used.

Functions	Message Types	
	Single Level	Multilevel
Text Insertion	x	x
Formatting Only	x	
Second Level Informational Chaining		x

Figure 106. PUTLINE Functions and Message Types

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 write them to the output data set. 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. Each segment can contain from 0 to 255 characters, as long as the total number of characters in all the segments does not exceed 256. 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. 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 writes it out.

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 invalid 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	14	TEXT
13	17	CONTINUE

PUTLINE would place the first insert, TEXT, after the 13th character, and the second insert, CONTINUE, after the 17th 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 output data set.

Figure 108 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 to the output data set. 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.

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 writes it to the output data set. 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 write it to the output data set in its entirety.

To purge the chain without putting it out to the output data set, 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 write the entire chain, 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 output data set and eliminate the internal chain.

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. Also, do not place a message identifier on a second level message.

Return Codes from PUTLINE

When the PUTLINE service routine returns control to the program that invoked it, it provides one of the following return codes in general register 15:

Figure 107. Return Codes from the PUTLINE Service Routine

Return Code Dec(Hex)	Meaning
0(0)	PUTLINE completed normally.
12(C)	Invalid parameters were supplied to the PUTLINE service routine.
16(10)	A conditional GETMAIN was issued by PUTLINE for output buffers and there was not sufficient real storage to satisfy the request.

Note: See Chapter 18, "Analyzing Error Conditions with the GNRLFAIL/VSAMFAIL Routine (IKJEFF19)" on page 235 for information on how to issue meaningful error messages for PUTLINE error codes.

End of GENERAL-USE PROGRAMMING INTERFACE

```
* 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 108. Example Showing PUTLINE Text Insertion

Using PUTGET to Put a Message Out and Obtain a Line of Input in Response

Use the PUTGET macro instruction to write MODE messages to the output data set and to obtain input data from the input stream. MODE messages indicate the processing mode that TSO is in. An example of a mode message is the READY message, which is written to the output data set by the terminal monitor program to indicate that it expects to retrieve a command name from the input stream.

The input line returned by the PUTGET service routine can come from the data set indicated by the SYSTSIN DD statement in the TSO user's JCL, or from an in-storage list; PUTGET determines the source of input from the top element of the input stack.

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
- 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 the following is required:

PUTGET	MF=L

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 operand and its sublist 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 109 shows the list form of the PUTGET macro instruction; each of the operands is explained following the figure.

```
[symbol]    PUTGET    [OUTPUT=(output address, SINGLE, MODE) ]
                      ,MF=L
```

Figure 109. 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 output data set. 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.

SINGLE
The output message is a single level message.

MODE
The output line is a mode message.

MF = L
Indicates that this is the list form of the macro instruction.

The Execute Form of the PUTGET Macro Instruction

Use the execute form of the PUTGET macro instruction to do the following:

- Prepare a mode message for output.
- 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 operand and sublists are optional because you can supply them in the list form of the macro instruction or in a previous execute form.

The operands you specify in the execute form of the PUTGET macro instruction set up control information used by the PUTGET service routine. You can use the PARM, UPT, ECT, and ECB operands of the PUTGET macro instruction to build, complete,

or modify an IOPL. The OUTPUT operand and its sublist operands initialize the PUTGET parameter block. The PUTGET parameter block is referenced by the PUTGET service routine to determine which functions you want PUTGET to perform.

Figure 110 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]
                    [,OUTPUT=(output address, SINGLE, MODE )]
                    [,ENTRY= {entry address}]  ,MF=(E{list address})
                             {    (15)     }]       {    (1)     }
```

Figure 110. 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.

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. 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.

SINGLE
The output message is a single level message.

MODE
The output line is a mode message.

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.

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. Since 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. Figure 111 describes the contents of the PUTGET parameter block.

Figure 111. The PUTGET Parameter Block

Number of Bytes	Field	Contents or Meaning
2		PUT control flags. These bits describe the output line to the PUTGET service routine.
	Byte 1	
	..0.	Always zero for PUTGET.
	...1	The output line is a single level message.
	.... 0...	Must be zero for PUTGET.
	xx.. .xxx	Reserved.
	Byte 2	
	1...	The output line is a MODE message.
	.xxx xxxx	Reserved.
2		Reserved.
4		The address of the output line descriptor.
2		GET control flags.
	Byte 1	
	.00.	Always zero for PUTGET.
	x..x xxxx	Reserved bits.
	Byte 2	
	xxxx xxxx	Reserved.
2		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.

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.

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).

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. Figure 112 shows the format of the output line descriptor.

Figure 112. The Output Line Descriptor (OLD)

Number of Bytes	Field	Contents or Meaning
4		This field must contain a value of zero.
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 113 shows the control block structures possible when passing an output message to the PUTGET service routine.

Message segments for PUTGET must follow the same rules as those for PUTLINE informational processing. (See "PUTLINE Message Line Processing" on page 188.) Note that if a PUTGET message segment does not contain at least one blank, PUTGET sets a return code of 24, indicating invalid parameters, in register 15.

End of GENERAL-USE PROGRAMMING INTERFACE

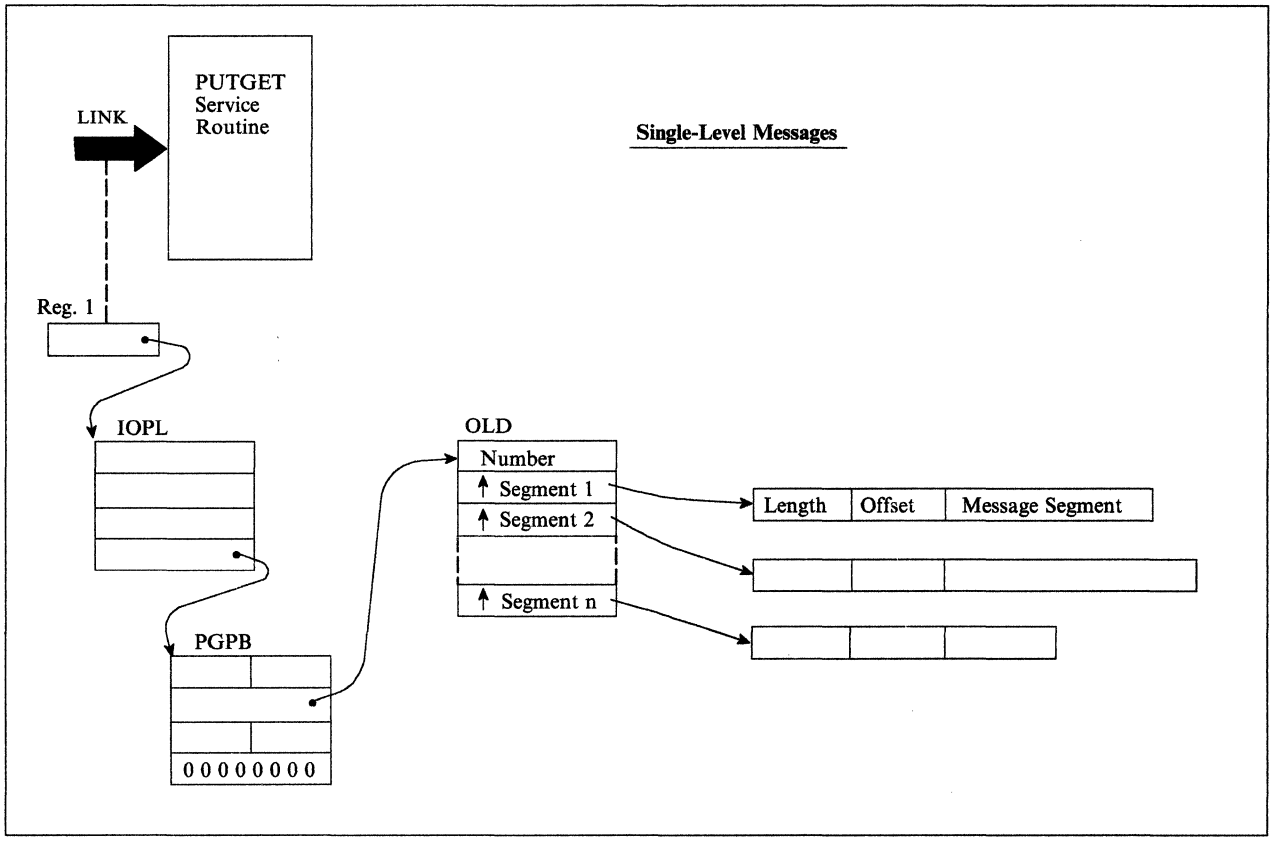


Figure 113. Control Block Structures for PUTGET Output Messages

PUTGET Processing

Text insertion is 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 188.

Mode Message Processing: A mode message is a message written to the output data set when a command or a subcommand is anticipated.

You are not required to provide an output line to the PUTGET service routine. If you provide an output line address then PUT processing will take place. However, if you do not provide an output line (OUTPUT=0) then only the GET function of the PUTGET service routine takes place.

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. The input buffer belongs to the program that issued the PUTGET macro instruction. The buffer or buffers returned by PUTGET are automatically freed when your code relinquishes control. You can free the input buffer with the FREEMAIN macro instruction after you have processed or copied the input line.

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 114 shows the format of the input buffer returned by the PUTGET service routine.

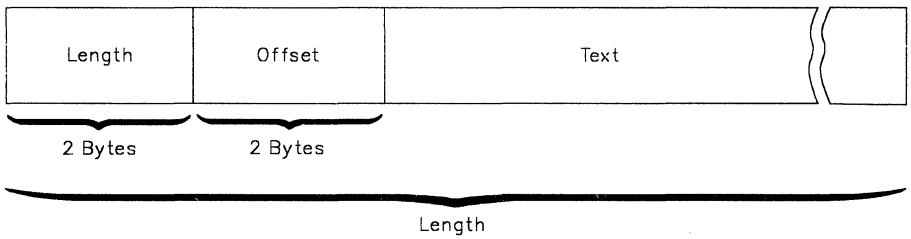


Figure 114. Format of the PUTGET Input Buffer

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.

Return Codes from PUTGET

When the PUTGET service routine returns control to the program that invoked it, it provides one of the following return codes in general register 15.

Return Code Dec(Hex)	Meaning
4(4)	PUTGET completed normally.
12(C)	A line could not be obtained after a MODE request.
24(18)	Invalid parameters were supplied to the PUTGET service routine.
28(1C)	A conditional GETMAIN was issued by PUTGET for output buffers and there was not sufficient space to satisfy the request.

Figure 115. Return Codes from the PUTGET Service Routine

Chapter 15. Using the TSO Message Handling Routine (IKJEFF02)

This chapter describes how to use the TSO message handling routine (IKJEFF02) in a command processor to issue messages.

GENERAL-USE PROGRAMMING INTERFACE

Functions of the TSO Message Issuer Routine (IKJEFF02)

The TSO 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 mode messages, you should indicate to IKJEFF02 that PUTGET should be used to issue the message; for prompting and informational messages, PUTLINE should be used.

You can invoke IKJEFF02 just to issue the message, 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 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 Message Issuer Routine

Your command processor can invoke the TSO message issuer routine by 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 202.

IKJEFF02 must receive control in 24-bit addressing mode. If your program uses the CALLTSSR macro instruction to invoke IKJEFF02, and IKJEFF02 resides in the link pack area, your program must issue the CALLTSSR macro instruction in 24-bit addressing mode. However, if IKJEFF02 does *not* reside in the link pack area, your program can issue the CALLTSSR macro instruction in either 24- or 31- bit addressing mode.

All input passed to IKJEFF02 must reside below 16 megabytes in virtual storage.

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. It also contains the address of a CSECT that contains the text of the message.

The IKJEFFMT macro is provided in SYS1.MACLIB. Use the MTDSECT=YES option to map the MTDSECTD DSECT, instead of obtaining storage. MTDSECT=NO is the default.

The Input Parameter List for IKJEFF02

The IKJEFFMT macro generates the input parameter list described below.

Figure 116 (Page 1 of 3). Standard Format of Input Parameter List

Offset Dec(Hex)	Field Name	Contents
0(0)	MTPLPTR	Address of message description section of this parameter list. (The message description section begins with the MTCSECTP entry.)
4(4)	MTCPPLP	Address of TMP's CPPL control block (required for PUTLINE or PUTGET).
8(8)	MTECBP	Address of optional communications ECB for PUTLINE or PUTGET.
12(C)	MTRESV1	Reserved.
12(C)	MTHIGH	High-order bit of reserved field turned on for standard linkage.
16(10)	MTCSECTP	Address of an assembly module or a CSECT containing IKJTSMSG macros that build message identifications and associated texts.
20(14)	MTSW1	One byte field of switches.
	MTNOIDSW	1... .... Message is printed; no messageid 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.

Figure 116 (Page 2 of 3). Standard Format of Input Parameter List

Offset Dec(Hex)	Field Name	Contents
	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 two-byte field containing length of text plus header field.
24(18)	MTSW2	One 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, using the delete operator message macro.
	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.
25(19)	MTRESV2	Reserved.
28(1C)	MTOLDPTR	Pointer to OLD for second level message, required if MT2OLDSW bit is on.
32(20)	MTEXTRLN	Length of extract buffer.
33(21)	MTEXTRBF	Pointer to extract buffer supplied by caller.
36(24)	MTEXTRL2	Length of extract buffer for second level message.
37(25)	MTEXTRB2	Pointer to extract buffer supplied by caller for second level message.
40(28)	MTMSGID	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.

Figure 116 (Page 3 of 3). Standard Format of Input Parameter List		
Offset Dec(Hex)	**Field Name**	**Contents**
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).
44(2C)	MTINSRT	Refers to an insert entry.
45(2D)	MTADDR	Address of an insert for the message.

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 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.

To issue the IKJTSMSG macro instruction, your program must reside below 16 megabytes in virtual storage.

Figure 117 shows the syntax of the IKJTSMSG macro instruction; each of the operands is explained following the figure.

```
[symbol]   IKJTSMSG ('msgid msgtext'),id1[,id2]
```

Figure 117. The IKJTSMSG Macro Instruction

msgid
The identifier which will be used 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, parentheses, or an apostrophe. Pass this id to IKJEFF02 in the MTMSGID 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 Message Issuer Routine

When the TSO message issuer routine returns control to its caller, register 15 contains one of the following return codes:

Figure 118. Return Codes from the TSO 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 190 or "Return Codes from PUTGET" on page 199.

End of GENERAL-USE PROGRAMMING INTERFACE

An Example Using IKJTSMSG

Figure 119 on page 206 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.

```
*
* 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 NATIONAL 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 119. An Example Using the IKJTSMSG Macro Instruction

Chapter 16. 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 environment, dynamic allocation permits the terminal monitor program, command processors, and other problem programs to allocate and free data sets during execution of the job step.

Programs that execute in the TSO environment can access dynamic allocation directly, using SVC 99, or through the dynamic allocation interface routine (DAIR). *Though its use is not recommended 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 *SPL: Application Development 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)."

The DAIR service routine can be invoked in either 24- or 31-bit addressing mode. When invoked in 31-bit addressing mode, DAIR can be passed input above 16 megabytes 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 command processor parameter list (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 Figure 13 on page 51.

You can use the IKJDAPL DSECT, which is provided in SYS1.MACLIB to map the fields in the DAPL. Figure 120 shows the format of the DAPL.

Figure 120. The DAIR Parameter List (DAPL)

Field Label	Byte Offset	Byte Length	Contents or Meaning
DAPLUPT	0(0)	4	The address of the user profile table.
DAPLECT	4(4)	4	The address of the environment control table.
DAPLECB	8(8)	4	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.
DAPLPSCB	12(C)	4	The address of the protected step control block.
DAPLDAPB	16(10)	4	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. Figure 121 lists the DAIR entry codes and the functions requested by those codes.

Figure 121. 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'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 Figure 122 to request that DAIR determine whether or not the specified DSNAME or DDNAME is allocated. Use the IKJDAP00 mapping macro, which is provided in SYS1.MACLIB, to map this DAIR parameter block.

Figure 122. DAIR Parameter Block for Entry Code'00'

Number of Bytes	Field	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.
	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	
	0000 0000	Reserved. Set to zero.
2		Reserved bytes; 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 229 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 Figure 123 to request that DAIR determine whether or not 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.

Figure 123. DAIR Parameter Block for Entry Code X'04'

Number of Bytes	Field	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 bits. Set 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	
	0000 0000	Reserved. Set to zero.
2		Reserved bytes; 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 one of the following codes in register 15:

0, 4, 8, or 52

See "Return Codes from DAIR" on page 229 for return code meanings.

Allocating a Data Set by DSNAME (Entry Code X'08')

Build the DAIR parameter block shown in Figure 124 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. 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, unallocate 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 Figure 124.

Figure 124 (Page 1 of 3). DAIR Parameter Block for Entry Code X'08'

Number of Bytes	Field	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 bits. Set 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 "Return Codes from Dynamic Allocation" on page 230.)
2	DA08CTRC	This field contains the error code, if any, returned from catalog management routines. (See "Return Codes from DAIR" on page 229.)
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	This is an eight-byte field containing an esoteric group name, a generic group name, or a specific device address (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 obtain information from the protected step control block. If there is no unit information in the PSCB, then a default of all direct access devices is used. The specified unit information will be ignored if volume information is obtained from the catalog, unless the unit specification is a subset of that obtained from the catalog. In this case the specified unit information will override the returned information.
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.

Figure 124 (Page 2 of 3). DAIR Parameter Block for Entry Code X'08'

Number of Bytes	Field	Contents or Meaning
4	DA08BLK	This is a four-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 low-order two 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 low-order three 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 low-order three 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 low-order three 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.
1	DA08DPS2	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 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 bits. Set them to zero.
	.... 1...	KEEP
	.... .1..	DELETE
	.... ..1.	CATLG
	1	UNCATLG
		If this byte is zero, DA08DPS2 will be used.

Figure 124 (Page 3 of 3). DAIR Parameter Block for Entry Code X'08'

Number of Bytes	Field	Contents or Meaning
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).
	..0.	Reserved bit; set to zero.
	...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 bit; set to zero.
3		Reserved bytes; set them to zero.
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 DD name 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 one of the following codes in register 15:

0, 4, 8, 12, 16, 20, 28, 32, 44, or 52

See "Return Codes from DAIR" on page 229 for return code meanings.

Concatenating the Specified DDNAMES (Entry Code X'0C')

Build the DAIR parameter block shown in Figure 125 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.

Figure 125. DAIR Parameter Block for Entry Code X'0C'

Number of Bytes	Field	Contents or Meaning
2	DA0CCD	Entry code X'000C'
2	DA0CFLG	Reserved. Set this field to zero.
2	DA0CDARC	This field contains the error code, if any, returned from the dynamic allocation routines. (See "Return Codes from Dynamic Allocation" on page 230.)
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 one of the following codes in register 15.

0, 4, 12, or 52

See "Return Codes from DAIR" on page 229 for return code meanings.

Deconcatenating the Indicated DDNAME (Entry Code X'10')

Build the DAIR parameter block shown in Figure 126 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.

Figure 126. DAIR Parameter Block for Entry Code X'10'

Number of Bytes	Field	Contents or Meaning
2	DA10CD	Entry code X'0010'
2	DA10FLG	Reserved. Set this field to zero.
2	DA10DARC	This field contains the error code, if any, returned from the dynamic allocation routines. (See "Return Codes from Dynamic Allocation" on page 230.)
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 one of the following codes in register 15:

0, 4, 12, or 52

See "Return Codes from DAIR" on page 229 for return code meanings.

Returning Qualifiers for the Specified DSNAME (Entry Code X'14')

Build the DAIR parameter block shown in Figure 127 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.

Figure 127. DAIR Parameter Block for Entry Code X'14'

Number of Bytes	Field	Contents or Meaning
2	DA14CD	Entry code X'0014'
2	DA14FLG	Reserved. Set this field to zero.
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.
	Byte 1 0000 0000	Reserved. Set to zero.
3		Reserved bytes. Set this field to zero.

After attempting the requested function, DAIR returns one of the following codes in register 15:

0, 4, 36, or 40

See "Return Codes from DAIR" on page 229 for return code meanings.

Freeing the Specified Data Set (Entry Code X'18')

Build the DAIR parameter block shown in Figure 128 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.

Figure 128. DAIR Parameter Block for Entry Code X'18'

Number of Bytes	Field	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 "Return Codes from Dynamic Allocation" on page 230.)
2	DA18CTRC	This field contains the error code, if any, returned from catalog management routines. (See "Return Codes from DAIR" on page 229.)
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:
	000. 0000	Reserved bits; set them to zero.
	...1	If this bit is on, permanently allocated data sets are unallocated. 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 one of the following codes in register 15:

0, 4, 12, 24, 28, or 52

See "Return Codes from DAIR" on page 229 for return code meanings.

Allocating a Data Set by DDNAME (Entry Code X'24')

Build the DAIR parameter block shown in Figure 129 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.

Figure 129 (Page 1 of 3). DAIR Parameter Block for Entry Code X'24'

Number of Bytes	Field	Contents or Meaning
2	DA24CD	Entry code X'0024'
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. Register 15 contains an error code.
	.... 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 "Return Codes from Dynamic Allocation" on page 230.)
2	DA24CTRC	This field contains the error code, if any, returned from catalog management routines. (See "Return Codes from DAIR" on page 229.)
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.

Figure 129 (Page 2 of 3). DAIR Parameter Block for Entry Code X'24'

Number of Bytes	Field	Contents or Meaning
8	DA24UNIT	This is an eight-byte field containing an esoteric group name, a generic group name, or a specific device address (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 obtain information from the protected step control block. If there is no unit information in the PSCB, then a default of all direct access devices is used. The specified unit information will be ignored if volume information is obtained from the catalog, unless the unit specification is a subset of that obtained from the catalog. In this case the specified unit information will override the returned information.
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 four-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 low-order two bytes. The high-order byte must be zero.
4	DA24PQTY	Primary space quantity desired. The high-order byte must be set to zero; the low-order three 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 low order three 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 low order three 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.
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.

Figure 129 (Page 3 of 3). DAIR Parameter Block for Entry Code X'24'

Number of Bytes	Field	Contents or Meaning
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.
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).
	..0.	Reserved; set this bit to zero.
	...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.
	0	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 DD name 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 one of the following codes in register 15:

0, 4, 8, 12, 16, 20, or 52

See "Return Codes from DAIR" on page 229 for return code meanings.

Performing a List of DAIR Operations (Entry Code X'28')

Build the DAIR parameter block shown in Figure 130 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.

Figure 130. DAIR Parameter Block for Entry Code X'28'

Number of Bytes	Field	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 one of the following codes in register 15:

0, 4, 8, 12, 16, 20, 24, 28, 32, 44, or 52

For return code meanings see "Return Codes from DAIR" on page 229.

Marking Data Sets as Not in Use (Entry Code X'2C')

Build the DAIR parameter block shown in Figure 131 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.

Figure 131. DAIR Parameter Block for Entry Code X'2C'

Number of Bytes	Field	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** 0000 — Mark all data sets of the indicated TCB 'not in use'. 0001 — Mark the specified DDNAME 'not in use'. 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 hex 0000.
8	DA2CDDN	Place in this field the DDNAME to be marked 'not in use'. DA2CFLG must be set to hex 0001.

After attempting the requested function, DAIR returns one of the following codes in register 15:

0, 4, or 52

For return code meanings see "Return Codes from DAIR" on page 229.

Allocating a SYSOUT Data Set to the Message Class (Entry Code X'30')

Build the DAIR parameter block shown in Figure 132 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 by using DAIR entry code X'18'.

When setting disposition in a parameter list, only one bit should be on.

Figure 132 (Page 1 of 2). DAIR Parameter Block for Entry Code X'30'

Number of Bytes	Field	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 "Return Codes from Dynamic Allocation" on page 230.)
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 eight-byte field containing an esoteric group name, a generic group name, or a specific device address (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 obtain unit information from the protected step control block. If there is no unit information in the PSCB, then a default of all direct access devices is used. The specified unit information will be ignored if volume information is obtained from the catalog, unless the unit specification is a subset of that obtained from the catalog. In this case the specified unit information will override the returned information.
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 low-order three 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.
4	DA30SQTY	Secondary space quantity desired. The high-order byte must be set to zero; the low-order three bytes should contain the secondary space quantity required. If the quantity is omitted, the entire field must be set to zero.

Figure 132 (Page 2 of 2). DAIR Parameter Block for Entry Code X'30'

Number of Bytes	Field	Contents or Meaning
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).
	..0.	Reserved; set this bit to zero.
	...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 one of the following codes in register 15:

0, 4, 12, 16, 20, 28, or 52

See "Return Codes from DAIR" on page 229 for return code meanings.

Associating DCB Parameters with a Specified Name (Entry Code X'34')

Build the DAIR parameter block shown in Figure 133 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).

Figure 133. DAIR Parameter Block for Entry Code X'34'

Number of Bytes	Field	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:
	Byte 1 DA34FIND 1... .... 0... .... .000 0000	 An attribute list name was found. An attribute list name was not found. 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 "Return Codes from Dynamic Allocation" on page 230.)
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. ....0 0000	 Delete an attribute list name. 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 one of the following codes in register 15:

0, 4, 12, or 52

See "Return Codes from DAIR" on page 229 for return code meanings.

The DAIR Attribute Control Block (DAIRACB)

Build the DAIRACB shown in Figure 134 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 Figure 133. Use the IKJDACB mapping macro, which is provided in SYS1.MACLIB, to map the DAIRACB.

Figure 134 (Page 1 of 2). DAIR Attribute Control Block (DAIRACB)

Number of Bytes	Field	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.
	DAIINOUT	
	1...	Use the INOUT option.
	DAIOUTIN	
	.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.
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). ;row.
	.... .1..	ASCII printer characters (A).
	.... ..1.	Machine control characters (M).
	0	Reserved bit. Should be set to zero.

Figure 134 (Page 2 of 2). DAIR Attribute Control Block (DAIRACB)

Number of Bytes	Field	Contents or Meaning
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:

Figure 135. Return Codes from DAIR

Return Code Dec(Hex)	Meaning
0(0)	DAIR completed successfully.
4(4)	The parameter list passed to DAIR was invalid.
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.
48(30)	Reserved.
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 *SPL: Application Development Guide*.

Return Codes from Dynamic Allocation

The codes returned in the DAxxDARC field of the DAIR parameter block, when a DAIR return code of 12 is returned, are the dynamic allocation error reason codes. (See *SPL: Application Development Guide*.) 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:

Figure 136. Return 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	Invalid characters are present in the ddname provided by the caller.
031C	Invalid characters are present in the membername provided by the caller.
0320	Invalid characters are present in the dsname provided by the caller.
0324	Invalid characters are present in the SYSOUT program name provided by the caller.
0328	Invalid characters are present in the SYSOUT form number provided by the caller.
032C	An invalid 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 invalid.
0348 through 034C	Reserved.

End of GENERAL-USE PROGRAMMING INTERFACE

Chapter 17. 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.

GENERAL-USE PROGRAMMING INTERFACE

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 must receive control in 24-bit addressing mode, and be passed input that resides below 16 megabytes in virtual storage. If your program executes in 31-bit addressing mode, you can use the LINK macro instruction to invoke DAIRFAIL without switching addressing modes. The LINK macro instruction ensures that DAIRFAIL receives control in 24-bit addressing mode.

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 or NO
Use the DFDSECT = YES option to map the DFDSECTD DSECT, instead of obtaining storage. DFDSECT = NO is the default.

DFDSEC2 = YES or 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:

Figure 137. The Parameter List (DFDSECTD DSECT)

Offset Dec(Hex)	Field Name	Contents
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.
8(8)	DFJEFF02	Address of a fullword containing either the entry point address of IKJEFF02 (message writer routine) or zeros, if that address is unknown. This field (DFJEFF02) must always contain an address.
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 (WTO). Byte 2 Caller identification number X'01' - DAIR X'32' - SVC 99
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 Figure 138, 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).

Figure 138. The Parameter List (DFDSECT2 DSECT)

Offset Dec(Hex)	**Field Name**	**Contents**
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:

Figure 139. Return Codes from DAIRFAIL

Return Code Dec(Hex)	Meaning
0(0)	A message was issued successfully.
4(4)	An invalid 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.

End of GENERAL-USE PROGRAMMING INTERFACE

Chapter 18. Analyzing Error Conditions with the GNRLFAIL/VSAMFAIL Routine (IKJEFF19)

This chapter describes how to use the GNRLFAIL/VSAMFAIL routine 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 *VSAM Administration: Macro Instruction Reference*.

GENERAL-USE PROGRAMMING INTERFACE

Passing Control to GNRLFAIL/VSAMFAIL

Your program can invoke the GNRLFAIL/VSAMFAIL routine by using the LINK macro instruction, specifying IKJEFF19 as the entry point name. However, you must first create the parameter list and place its address into register 1.

GNRLFAIL/VSAMFAIL must receive control in 24-bit addressing mode, and be passed input that resides below 16 megabytes in virtual storage. If your program executes in 31-bit addressing mode, you can use the LINK macro instruction to invoke GNRLFAIL/VSAMFAIL without switching addressing modes. The LINK macro instruction ensures that GNRLFAIL/VSAMFAIL receives control in 24-bit addressing mode.

The Parameter List

Use the IKJEFFGF macro, which is provided in SYS1.MACLIB, to map the parameter list for IKJEFF19. 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 parameter list:

Figure 140 (Page 1 of 2). The Parameter List (GFDSECTD DSECT)

Offset Dec(Hex)	Field Name	Contents
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 message issuer routine (IKJEFF02) if already loaded.

Figure 140 (Page 2 of 2). The Parameter List (GFDSECTD DSECT)

Offset Dec(Hex)	Field Name	Contents
12(C)	GFCALLID	ID for caller's failing VSAM macro, or other failure. This field can have the following values: **Value** **Meaning** GFCHECK (X'01') VSAM CHECK macro error GFCLOSE (X'02') VSAM CLOSE macro error GFENDREQ (X'03') VSAM ENDREQ macro error GFERASE (X'04') VSAM ERASE macro error GFGET (X'05') VSAM GET macro error GFOPEN (X'06') VSAM OPEN macro error GFPOINT (X'07') VSAM POINT macro error GFPUT (X'08') VSAM PUT macro error GFPARSE (X'15') Parse service routine error, other than a return code of 4 or 20. GFPUTL (X'16') PUTLINE service routine error GFABEND (X'1F') Issue ABEND message GFSSREQ (X'20') Subsystem interface request 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.
15(F)	GFRESV1	Reserved.
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).
36(24)	GFRESV2	Reserved.
40(28)	GFRESV3	Reserved.

Return Codes from GNRLFAIL/VSAMFAIL

When GNRLFAIL/VSAMFAIL returns to its caller, register 15 contains one of the following return codes:

Figure 141. 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 invalid. A message is also issued.
Other	This error return code is from either PUTLINE, PUTGET or the message issuer routine (IKJEFF02).

End of GENERAL-USE PROGRAMMING INTERFACE

Chapter 19. 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 ID. 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.

GENERAL-USE PROGRAMMING INTERFACE

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.

The catalog information routine resides in SYS1.LPALIB and executes with the protection key of the caller. IKJEHCIR can be invoked in either 24- or 31-bit addressing mode. However, all input passed to IKJEHCIR must reside below 16 megabytes in virtual storage. IKJEHCIR executes in 24-bit addressing mode and returns control in the same addressing mode in which it is invoked.

The Catalog Information Routine Parameter List (CIRPARM)

The catalog information routine parameter list (CIRPARM) is shown in Figure 142.

Figure 142. 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 Figure 143 on page 241.
1(1)	2		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 high-level qualifier or a name of the form *high-level-qualifier.user-supplied-name*. In this case, the search argument is **not** a fully-qualified data set name. For entry code X'04', the search argument is a high-level qualifier and a data set name which are names of catalog index levels. For entry codes X'05' and X'06', the search argument is a high-level qualifier followed by a period.
8(8)	4	CIRCVOL	Address of the volume ID of CVOL. If not given, SYSRES is assumed.
12(C)	4	CIRWA	Address of the user work area. See Figure 144 on page 242 for a description of the user work area.
16(10)	4	CIRSAVE	Address of a 72-byte save area.
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 Figure 143.

Figure 143. 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.	8-byte qualifiers are moved into the user work area.
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 Figure 145 on page 242 for volume information format.
X'05'	Retrieve the next level data set name and volume information.	45-byte data set name and volume information is moved to the user work area.
X'06'	Retrieve all data set names and volume information.	45-byte data set name followed by volume information is moved to the user work area for all levels.

Note: For codes X'02', X'05', and X'06', a one-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

The user work area that is based on CIRWA is shown in Figure 144.

Figure 144. 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'.

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 2-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. Each 12-byte entry consists of a 4-byte device code, a 6-byte volume serial number, and a 2-byte sequence number. As many as 20 of these 12-byte entries can be built in your work area. The volume list has an end indicator of X'FF'. Figure 145 shows the format of the volume list.

Figure 145. 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.)
1		Reserved. (Contains X'FF'.)

Return Codes from IKJEHCIR

When IKJEHCIR returns to its caller, register 15 contains one of the following return codes:

Figure 146. Return Codes from IKJEHCIR

Return Code Dec(Hex)	Meaning
0(0)	Successful completion of the request.
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:

Figure 147. Return Codes from LOCATE

Return Code Dec(Hex)	Meaning
0(0)	Successful completion of the request.
4(4)	The required catalog does not exist, it cannot be opened, or there is a closed chain of OS CVOL pointers.
8(8)	One of the following occurred: • The entry was not found. If in an OS CVOL, register 0 contains the number of valid index levels. If in an ICF or a VSAM catalog, 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)	One of the following occurred: • An index or generation data group base entry was found when the list of qualified names was exhausted. Register 0 contains the number of valid index levels. The work area contains the first block of the specified index. • An alias entry was found. The alias name was replaced in the user parameter list by the true name. • An invalid low-level GDG name was found.
16(10)	A data set exists at other than the lowest index level specified. Register 0 contains the number of the index level where the data set was encountered.
20(14)	A syntax error exists in the name.
24(18)	One of the following occurred: • Permanent I/O error occurred. Register 0 contains the VSAM or ICF return code, or 0 if in an OS CVOL. • Nonzero ESTAE return code. • Error in parameter list.
28(1C)	The relative track address supplied to the LOCATE routine is outside of the SYSCTLG data set extents.
32(20)	Reserved.

For additional LOCATE return codes, see the description of message IDC3009I in *Message Library: System Messages.*

End of GENERAL-USE PROGRAMMING INTERFACE

Part III. TSO Commands

This part describes the functions and syntax of TSO commands. It includes:

- The general format and syntax rules for the commands
- A description of each command including return code information and examples. The commands are described in alphabetical order.

Chapter 20. Command Format and Syntax

This chapter discusses the general format and syntax rules for the TSO commands.

Using a TSO Command

A command consists of a command name usually followed by one or more operands. Operands provide the specific information required to perform the requested operation.

You can use two types of operands with the commands: *positional* and *keyword*.

Positional Operands

Positional operands follow the command name in a certain order. In the command descriptions in Chapter 21, "Command Descriptions" on page 251, the positional operands are shown in lowercase characters.

When you specify a positional operand that is a list of several names or values, you must enclose the list within parentheses.

Keyword Operands

Keyword operands (keywords) are specific names or symbols that have a particular meaning to the system. You can include keywords in any order following the positional operands. In the command descriptions in Chapter 21, "Command Descriptions" on page 251, keywords are shown in uppercase characters.

You can specify values with some keywords. Enclose the value with parentheses following the keyword.

If you specify conflicting, mutually exclusive keywords, the last keyword you specify overrides the previous ones.

Figure 148 on page 248 describes the syntax notation for the TSO commands.

TSO Command Syntax

The following figure summarizes the notation for the TSO commands.

Figure 148. TSO Command Syntax

Notation	Meaning
Delimiters (blank or comma)	Separates operands or characters from a command or an operand.
Brackets [ ]	Indicates that the operands within the brackets are optional and you can omit them. Do not type the brackets when specifying the command.
Braces { }	Indicates that you must specify one of the items. You cannot specify more than one. Do not type the braces when specifying the command.
Hyphen –	Joins lowercase words to form a single variable. Do not type the hyphen when specifying the command.
lowercase	Represents variables for which you are to substitute specific information. You can specify the information in uppercase or lowercase.
Parentheses ( )	Enter the parentheses as shown. You do not have to type the closing parentheses if it is last character in the command.
Quotes '	Separates text strings. You must specify a single quote in the character string as two adjacent quotes.
Underscore _	Indicates the keyword or value is the default for the operand.
Uppercase	Spell the operand as shown (or its abbreviation). You can specify the operand in either upper or lowercase.
Ellipsis (...)	Indicates that you can repeat the operand or the value. Do not specify the ellipsis.
Special characters such as * = . ? %	Use them as shown in the syntax diagram.

Abbreviating Keyword Operands

You can specify keywords spelled exactly as they are shown or you can use an acceptable abbreviation. You can abbreviate any keyword by specifying only the significant characters; that is, you must type as much of the keyword as is necessary to distinguish it from the other keywords of the command or subcommand.

Comments

You can include comments in a TSO command anywhere a blank might appear. To include a comment, start with delimiter /*. If you want to continue the command after the comment, close the comment with delimiter */.

```
CALL  data-set-name   /* MY PROGRAM
```

or

```
CALL  /* MY PROGRAM   */ data-set-name
```

You do not need to end a comment with */ if the comment is the last thing on the line. Ending a comment with */ is a convention, not a requirement in this case.

Line Continuation

When it is necessary to continue to the next input line, use a minus sign as the last character of the line you wish to continue. You can also use a plus sign, but be aware that if you do, leading delimiters will be removed from the continuation line.

```
CALL  data-set-name  /* THIS COMMAND IS USED -
                        TO INVOKE MY PROGRAM */
```

To continue a line that contains a comment, use a continuation character *after* the comment.

Delimiters

When you type a command, you must separate the command name from the first operand by one or more blanks. You must separate operands by one or more blanks or a comma. Do not use a semicolon as a delimiter because any character you specify after a semicolon is ignored. For example, if you use a blank or a comma as a delimiter, you can type the WHEN command as follows:

```
WHEN SYSRC(=0) CALL 'SYS1.LINKLIB(LNKED)'
WHEN SYSRC(=0),CALL 'SYS1.LINKLIB(LNKED)'
WHEN SYSRC(=0)    CALL 'SYS1.LINKLIB(LNKED)'
```

Chapter 21. Command Descriptions

This chapter describes the functions and syntax of the TSO commands. The commands are presented in alphabetical order.

TSO Command Summary

TSO provides the following commands:

Command	Function
CALL	Loads and executes a program.
TIME	Provides the date and time of day.
WHEN/END	Tests return codes from programs invoked from an immediately preceding CALL command, and takes a prescribed action if the return code meets a specified condition.

CALL Command

Use the CALL command to load and execute a program that exists in executable (load module) form. The program can be user-written, or it can be a system module such as a compiler, sort, or utility program.

You must specify the name of the program (load module) to be processed. It must be a member of a partitioned data set.

You can specify a list of parameters to be passed to the specified program. The system formats this data so that when the program receives control, register one contains the address of a fullword. The three low order bytes of this fullword contain the address of a halfword field. This halfword field is the count of the number of bytes of information contained in the parameter list. The parameters immediately follow the halfword field. When you pass parameters to a PL/I program, precede them with a slash (/). PL/I assumes that any value prior to the slash is a run-time option.

Service aids, utilities, and other programs obtaining their input from an allocated file such as SYSIN must have the input in a data set. Once the data set is created, you can use the CALL command to execute the program that accesses the SYSIN data.

Figure 149 illustrates the allocation and creation of an input data set.

```
//jobname  JOB
//EXAMP1   EXEC    PGM=IKJEFT01,DYNAMNBR=20
//SYSTSPRT DD      SYSOUT=A
//SYSTSIN  DD      *
  CALL  'SYS1.LINKLIB(PROG1)'
//SYSPRINT DD      SYSOUT=A
//SYSIN    DD      *
  **input to prog1**
/*
```

Figure 149. Allocating and Creating an Input Data Set

```
CALL              {dsname              }
                  {dsname(membername)  }

                   ['parameter-string']
```

dsname
specifies the name of a partitioned data set to be executed. If you do not specify a data set name, the default is TEMPNAME. A type of load is assumed.

dsname(membername)
specifies the name of a partitioned data set and the member name (program name) to be executed. You must enclose the member name in parentheses.

Note: A temporary tasklib is established when programs are invoked by the CALL command. The tasklib is effective for the execution of the CALL command and the tasklib data set is the same as the data set name specified on the invocation of the CALL command.

Enclose in apostrophes (single quotes), the name of the data set to be executed in the following manner:

```
'SYS1.LINKLIB(IEUASM)'
```

parameter string
specifies up to 100 characters of information that you want to pass to the program as a parameter list. When passing parameters to a program, you should use the standard linkage convention.

Return Codes for the CALL Command

0	Processing successful.
12	Processing unsuccessful. An error message has been issued.
Other	Return code is from the called program.

Examples

Example 1

Operation: Execute a load module.

Known:

The name of the load module: JUDAL.PEARL.LOAD(TEMPNAME)

Parameters: 10,18,23

```
CALL 'JUDAL.PEARL.LOAD' '10,18,23'
```

Example 2

Operation: Execute a load module.

Known:

The name of the load module: JUDAL.MYLIB.LOAD(COS1)

```
CALL 'JUDAL.MYLIB.LOAD(COS1)'
```

Example 3

Operation: Execute a PL/I load module passing a parameter.

Known:

The name of the load module: D58ABC.PCP.LOAD(MOD1)
The parameter to be passed: The character string ABC

```
CALL 'D58ABC.PCP.LOAD(MOD1)' '/ABC'
```

TIME Command

Use the TIME command to obtain the following information:

- Cumulative CPU time (from the start of the session)
- Cumulative session time (from the start of the session)
- Service units used, which can be:

  CPU service units - The task execution time, divided by an SRM constant, that is CPU model-dependent.

  I/O service units - The sum of individual SMF data set activity EXCP counts for all data sets associated with the address space.

  Storage service units - The number of real page frames multiplied by CPU service units, multiplied by .02. The decimal .02 is a scaling factor designed to bring the storage service component in line with the CPU component.

- Local time of day

  Refers to the time of execution for this command. It is displayed as follows:

  ```
  local time of day in hours(HH),
  minutes(MM), and seconds(SS),
  (am or pm is also displayed)
  ```

- Today's date.

  ```
  TIME
  ```

Return Code for the TIME Command

The return code is always zero.

WHEN/END Command

Use the WHEN command to test return codes from programs invoked by an immediately preceding CALL command, and to take a prescribed action if the return code meets a certain specified condition.

```
WHEN            SYSRC(operator  integer)

                [END          ]
                [command-name ]
```

SYSRC
specifies that the return code from the previous function is to be tested according to the values specified for operator and integer.

operator
specifies one of the following operators:

```
EQ or  = means equal to
NE or  ¬= means not equal to
GT or  > means greater than
LT or  < means less than
GE or  >= means greater than or equal to
NG or  ¬> means not greater than
LE or  <= means less than or equal to
NL or  ¬< means not less than
```

integer
specifies the numeric constant that the return code is to be compared to.

END
specifies that processing is to be terminated if the comparison is true. This is the default if you do not specify a command.

command-name
specifies any valid TSO command name and appropriate operands. The command will be processed if the comparison is true.

You can use successive WHEN commands to determine an exact return code and then perform some action based on that return code.

Return Codes for the WHEN Command

The return code is from the command that executed last.

Example

Operation: Using successive WHEN commands to determine an exact return code.

```
CALL     compiler
WHEN     SYSRC(= 0) CALL 'SYS1.LINKLIB(LNKED)'
WHEN     SYSRC(= 4) CALL 'SYS1.LINKLIB(LNKED)'
WHEN     SYSRC(= 8) END
```

Index

D

E

F

G

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Numerics

Reader's Comments

MVS/ESA
TSO Programming

MVS/System Product:
JES2 Version 3
JES3 Version 3

Publication No. GC28-1565-2

Use this form to tell us what you think about this manual. If you have found errors in it, or if you want to express your opinion about it (such as organization, subject matter, appearance) or make suggestions for improvement, this is the form to use.

To request additional publications, or to ask questions or make comments about the functions of IBM products or systems, you should talk to your IBM representative or to your IBM authorized remarketer. This form is provided for comments about the information in this manual and the way it is presented.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

Be sure to print your name and address below if you would like a reply.

Name

Address

Company or Organization

Phone No.

Cut o
Alon

Fold and Tape | **Please do not staple** | Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

International Business Machines Corporation
Department D58, Building 921-2
PO BOX 950
POUGHKEEPSIE NY 12602-9935

Fold and Tape | **Please do not staple** | Fold and Tape

Cu
Alc

Program Number
5685-001
5685-002

File Number
S370-39

GC28-1565-2

Printed in U.S.A.