z/OS

TSO/E Programming Guide

Version 2 Release 1

SA32-0981-00

Note
Before using this information and the product it supports, read the information in "Notices" on page 155.

This edition applies to Version 2 Release 1 of z/OS (5650-ZOS) and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

Figures

Tables

About this document

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

This document describes how to write and install a command processor in the topic about Part 2, "Writing a command processor," on page 13, and how to provide HELP information for a command in the topic about Chapter 9, "Creating HELP Information," on page 61. It also describes how to compile, assemble, link-edit, execute, and test a program in the TSO/E environment in the topic about Part 3, "Preparing, Executing and Testing a Program," on page 71.

Who should use this document

This document is for the following audience:

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

The reader should be familiar with MVS™ programming conventions and the assembler language, and should know how to use TSO/E.

How this document is organized

This document is divided into three parts:

- Part 1, "Introduction," on page 1 provides an overview of the types of programs that run under TSO/E and the services TSO/E offers.
- Part 2, "Writing a command processor," on page 13 describes how to write and install a command processor, and how to provide HELP information for a command. It discusses the TSO/E services that you can use in a command processor, and refers you to *z/OS TSO/E Programming Services* for more information, when needed.
- Part 3, "Preparing, Executing and Testing a Program," on page 71 describes how to use TSO/E to compile, assemble, bind or link-edit and execute a program. It also explains how to use the TSO/E TEST and TESTAUTH commands to test a program.

How to use this document

If you have never used this document, read Chapter 1, "Programming Using TSO/E," on page 3 to become familiar with the types of programs that you can write to run in the TSO/E environment.

If you are writing a command processor, read Part 2, "Writing a command processor," on page 13 of this document, and see *z/OS TSO/E Programming Services* for information on the TSO/E services that you can use in your command processor.

If you have written your program, read Part 3, "Preparing, Executing and Testing a Program," on page 71 of this document for information on how to compile, assemble, link-edit, execute and test your program.

Where to find more information

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

How to send your comments to IBM

We appreciate your input on this publication. Feel free to comment on the clarity, accuracy, and completeness of the information or provide any other feedback that you have.

Use one of the following methods to send your comments:

1. Send an email to mhvrcfs@us.ibm.com.
2. Send an email from the Contact z/OS.
3. Mail the comments to the following address:
   IBM Corporation
   Attention: MHVRCFS Reader Comments
   Department H6MA, Building 707
   2455 South Road
   Poughkeepsie, NY 12601-5400
   US
4. Fax the comments to us, as follows:
   From the United States and Canada: 1+845+432-9405
   From all other countries: Your international access code +1+845+432-9405

Include the following information:

- Your name and address.
- Your email address.
- Your telephone or fax number.
- The publication title and order number:
  z/OS V2R1.0 TSO/E Programming Guide
  SA32-0981-00
- The topic and page number that is related to your comment.
- The text of your comment.

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

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

If you have a technical problem

Do not use the feedback methods that are listed for sending comments. Instead, take one of the following actions:

- Contact your IBM service representative.
- Call IBM technical support.
- Visit the IBM Support Portal at IBM support portal.

z/OS Version 2 Release 1 summary of changes

See the following publications for all enhancements to z/OS Version 2 Release 1 (V2R1):

- *z/OS Migration*
- *z/OS Planning for Installation*
- *z/OS Summary of Message and Interface Changes*
- *z/OS Introduction and Release Guide*

Part 1. Introduction

You can use TSO/E to help you write, execute and test programs. Also, you can write programs to run in the TSO/E environment and use the services provided by TSO/E. The following topics provide an overview of the types of programs that run under TSO/E and the services TSO/E offers.

Chapter 1. Programming Using TSO/E

There are several types of programs that run under TSO/E: CLISTs, REXX execs, command processors and servers. Although these types of programs are introduced in the following topics, this document gives detailed information on command processors only. For information on writing REXX execs, see *z/OS TSO/E REXX User's Guide* and *z/OS TSO/E REXX Reference*. For a complete discussion of CLISTs, see *z/OS TSO/E CLISTs*. For more information on writing servers, see *z/OS TSO/E Guide to SRPI*.

What is a REXX Exec?

The REstructured eXtended eXecutor (REXX) language is a high-level interpretive language that enables you to write programs in a clear and structured way. You can use the REXX language to write programs, called execs, that perform a given task, or tasks, or group of tasks.

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

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

However, a significant difference between REXX execs and CLISTs is that you can execute CLISTs only in a TSO/E environment. REXX execs do not require a TSO/E environment, and can execute in any MVS address space.

TSO/E REXX is the implementation of the Systems Application Architecture® (SAA) Procedures Language on the MVS system. By using the instructions and functions defined for the SAA Procedures Language, you can write REXX execs that will run in any of the supported SAA environments, such as VM/SP (CMS). *SAA Common Programming Interface Procedures Language Level 2 Reference* describes the instructions and functions the SAA Procedures Language offers.

You can also write APPC/MVS transaction programs in the REXX language. The host command environments, CPICOMM, LU62, and APPCMVS, allow you to invoke the SAA common programming interface (CPI) Communications calls and APPC/MVS calls, which are based on the SNA LU 6.2 architecture, respectively. The CPICOMM host command environment allows transaction programs written in the REXX language to be ported across Systems Application Architecture (SAA) environments. The LU62 host command environment allows you to use specific features of MVS in conversations with transaction programs on other systems.

Note: APPC/MVS calls that are based on the SNA LU 6.2 architecture are referred to as APPC/MVS calls throughout the document.

For information about writing APPC/MVS transaction programs, see *z/OS MVS Programming: Writing Transaction Programs for APPC/MVS*. For information about writing and executing REXX execs, see *z/OS TSO/E REXX User's Guide* and *z/OS TSO/E REXX Reference*.

Another advantage to using REXX is that TSO/E provides support for a REXX compiler and run-time library, such as the IBM® Compiler and Library for REXX/370. Using a compiler provides significant benefit for programmers during program development and for users when a program is run.

What is a CLIST?

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

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

The CLIST language supports a range of programming functions that include:

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

CLISTs can complete a range of tasks. For example, use CLISTs to:

- Allocate data sets that are required for particular programs.
- Write structured applications, use the CLIST language, to call other CLISTs, define common data among nested CLISTs, and pass parameters between CLISTs.

  Write interactive applications by issuing commands of ISPF to display full-screen panels.
- Provide easy-to-use interfaces to applications written in other languages. CLISTs can prompt terminal users for information about requesting tasks. CLISTs can set up the environment that is needed for an application, and then issue the commands to use the application program.

For information about creating, using, and testing CLISTs, see *z/OS TSO/E CLISTs*.

What is a command processor?

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

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

A command processor is a program (written in assembler language, PL/1, or compiled REXX then linked into a load module) that receives control when a user

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

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

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

To write a command processor, you can use many of the services documented in *z/OS TSO/E Programming Services*. For example, you can use the TSO/E service facility (IKJEFTSR) to invoke other commands, CLISTs or programs. Part 2, "Writing a command processor," on page 13 provides guidelines about which TSO/E services to use, and how to write, test, and install a command processor.

Considerations for Writing REXX Execs, CLISTs and command processors

Often, you can perform the same programming functions by writing a program that is a command processor, a REXX exec, or a CLIST. For example, you can write any of these types of programs to do the following:

- List all of the users who have exclusive use of a resource, such as a data set
- List all of the data sets having a certain attribute, such as a record format of FB
- Process and display accounting information
- List the members of a partitioned data set

You can often solve application problems by combining the best attributes of REXX execs, CLISTs and command processors. For example, you can write a REXX exec or CLIST that interacts with the user, and then invokes a command processor to perform the primary processing.

Consider the following when deciding whether to implement a function as a command processor, REXX exec or CLIST:

- Environment in which the program executes.

  CLISTs and command processors execute in a TSO/E environment only. However, you can write REXX execs to execute in both TSO/E and other environments, because a subset of the REXX language is independent of both operating system software and hardware.
- Features of the programming language used.

  A command processor is typically written in assembler language. Thus, you must assemble and link-edit your program before executing it. However, a benefit of assembler language is that a command processor can access system services, such as those provided by data management and supervisor macro instructions.

  The REXX and CLIST languages are high-level languages that support a wide range of programming functions. For example, REXX and CLIST allow you to

perform I/O, invoke TSO/E commands and perform arithmetic operations. Both languages are interpretive; therefore, you are not required to compile or link-edit the program before executing it.

- Performance considerations.

  Because REXX is an interpretive language, each line is scanned, processed, and executed every time the program is run. For large REXX execs, this can result in reduced performance when compared to a program that is in load module form for execution.

  If you make your program available to many people, or if the program is used often, you should consider the performance aspect carefully. A large program may be more efficient if it is written as a command processor.

  REXX execs that are allocated to the SYSPROC system level file or the SYSPROC application level file are compressed when they are stored in the VLF data repository. In general, compression strips comments and leading and trailing blanks. Blank lines are replaced with null lines. For details on file compression restrictions, see "Storing REXX Execs in VLF Storage."

  The compression provides a potential performance benefit by reducing the amount of data being stored in the VLF data space for each REXX exec:

  – The reduction in file size means a reduction in the amount of central, expanded, and auxiliary storage used to process the REXX execs.
  – There are fewer page-in I/O operations and cross-memory move operations on each invocation of the REXX execs.
  – File compression allows for more storage in the repository for the same storage utilization.

Storing REXX Execs in VLF Storage

The author of the REXX exec can control whether a REXX exec allocated to a SYSPROC system-level or application-level CLIST library is to be compressed. In general, compression strips comments and leading and trailing blanks. Blank lines are replaced with null lines. Details on file compression restrictions for REXX are explained below.

Note: Automatic compression governed by the occurrence or non-occurrence of SOURCELINE in an exec can be overridden by explicit use of a compression indicator. You can specify either COMMENT or NOCOMMENT in a special comment in line 1 of the exec, as described in "Controlling REXX Exec Compression" to control compression.

When the system compresses a REXX exec, it removes the comment text and leaves the beginning and ending comment delimiters.

Controlling REXX Exec Compression

The following describes the control that you have over the compression of REXX execs in the VLF repository.

If you do not want an exec to be compressed, you can allocate the exec to either the SYSEXEC file or the SYSPROC user-level file.

If the system finds an explicit occurrence of the characters SOURCELINE outside of a comment in the exec and does not find a special comment with a compression indicator, as explained below, it does not compress the exec. For example, if you use the SOURCELINE built-in function, the system does not compress the exec. If

you use a variable called ASOURCELINE1 in the exec, the system does not compress the exec because it locates the characters SOURCELINE within that variable name. Note that the system does compress the exec if the exec contains a "hidden" use of the characters SOURCELINE. For example, you may concatenate the word SOURCE and the word LINE and then use the INTERPRET instruction to interpret the concatenation or you may use the hexadecimal representation of SOURCELINE. In these cases, the system does compress the exec because the characters SOURCELINE are not explicitly found.

The author of a REXX exec can control whether an exec is compressed by specifying a special comment in line 1 of the exec which contains a compression indicator (that is, either COMMENT or NOCOMMENT).

To specify a compression indicator, the first begin-comment delimiter /* in line 1 of the exec must be immediately followed by the special comment trigger character %. If the REXX processor finds a special comment begin-delimiter /*%, it scans the remainder of the comment looking for COMMENT or NOCOMMENT. Scanning terminates when one of the following is found:

- An end-comment delimiter */
- Another begin-comment delimiter /*
- End of line 1

The NOCOMMENT compression indicator indicates to the REXX processor that all comment text, except for the comment text within the special comment, is to be removed from the exec at the time it is loaded and before it is stored into VLF. (Comment removal constitutes exec compression.) COMMENT indicates to the REXX processor that the exec should retain its comment text and that no comment text should be removed.

Note:

1. Only execs which are invoked as implicit execs and which are found in SYSPROC or an application-level CLIST library defined by the TSO/E ALTLIB command are eligible for compression. (This is true for both execs which are compressed automatically, and those using controlled compression.)

   Therefore, execs loaded from SYSEXEC are never compressed.
2. When TSO/E REXX scans the special comment for a compression indicator, any keyword options that are not recognized are ignored. COMMENT and NOCOMMENT are the only recognized keyword options.
3. If both COMMENT and NOCOMMENT are specified within a special comment, only the last one specified is used.
4. Only EBCDIC characters should be used within the special comment.
5. If an exec is compressed, all comment text, except the comment text of the special comment, and leading and trailing blanks within each line of the exec are removed. However, the text of the special comment, up to the first begin-comment delimiter /*, end-comment delimiter */, or end-of-line (whichever is first) is not removed.
   - Although the comment text is removed, the begin-comment delimiter /* and end-comment delimiter */ are not removed. After compression, a typical comment will look like /**/.
   - Blanks and comments within literal strings (delimited by single or double quotation marks) are not removed.
   - Blanks and comments within a DBCS string (delimited by shift-out (X'0E') and shift-in (X'0F') are not removed.

- If comment lines are continued across several lines, blanks and text are removed from the line, but the line itself and the begin- and end-comment delimiters are kept to preserve relative line numbering.

Table 1 provides an overview of the conditions that cause an exec to be compressed.

Table 1. Determining When an Exec is Compressed

Invoked through CALL Instruction or External Function	**Invoked through EXEC Explicit Form**	**Invoked through EXEC Implicit Form**					
		User Library	Application- or System-Level Library				
			Exec Library	CLIST Library			
				COMMENT Trigger	NOCOMMENT Trigger	Neither COMMENT nor NOCOMMENT Trigger	
						SOURCELINE String Not Used in Exec	SOURCELINE String Used in Exec
Do NOT Compress	Do NOT Compress	Do NOT Compress	Do NOT Compress	Do NOT Compress	Compress	Compress	Do NOT Compress

What is an APPC/MVS Transaction Program?

APPC/MVS transaction programs are application programs running on MVS or TSO/E that use special calls to communicate with partner programs on the same MVS system, other MVS systems, or other operating systems in a System Network Architecture (SNA) network. To communicate, the transaction programs hold APPC conversations; in the conversations, they exchange information using specific calls and follow established conversation protocols.

APPC/MVS transaction programs can be involved with TSO/E in the following ways:

- You can write transaction programs in REXX using the LU62, CPICOMM, or APPCMVS host command environments to issue APPC calls to a partner transaction program. For more information on these host command environments, see *z/OS TSO/E REXX Reference*.
- Programs on MVS, including APPC/MVS transaction programs, can use the TSO/E environment service (IKJTSOEV) to create a TSO/E environment outside of TSO/E. The programs can then use TSO/E services as described in *z/OS TSO/E Programming Services*.
- Programmers can use the TSO/E TEST and TESTAUTH commands to test certain transaction programs. The commands can be used with inbound transaction programs—those that are initiated in response to inbound conversation requests from their partner transaction programs. These inbound transaction programs are specified on the TEST or TESTAUTH commands and, when requested by their partner programs, are initiated for testing under TSO/E. Outbound APPC/MVS transaction programs are those transaction programs that are not initiated by inbound conversation requests. You can test these transaction programs as ordinary programs using TEST or TESTAUTH (that is, do not specify the TP, LU, BASELU, or KEEPTP keywords). For more information about testing transaction programs on TSO/E, see *z/OS TSO/E Command Reference*.

For more information about APPC/MVS transaction programs in general, their benefits for cross-system communication and how to write and use transaction programs, see *z/OS MVS Programming: Writing Transaction Programs for APPC/MVS*.

What is a Server?

The command processor provides a standard way for application programs to share services. With this facility, programs on properly-configured IBM Personal Computers (PCs) can obtain services from programs on IBM host computers. The PC programs issue service requests and the host programs issue service replies, which the TSO/E Enhanced Connectivity Facility passes between the systems.

The PC programs that issue service requests are called *requesters*, and the host programs that issue replies are called *servers*. Servers can give PC requesters access to host computer data, commands and resources such as printers and storage. You can write servers to receive service requests, process the requests, and return replies to the requester.

For information on how to write, install, test and debug a server program, see *z/OS TSO/E Guide to SRPI*.

Overview of TSO/E Programming Services

TSO/E provides services that your programs can use to perform various tasks. Although some of these services are discussed in this document, see *z/OS TSO/E Programming Services* for a complete description.

Table 2 lists each TSO/E service and the task it supports.

Table 2. Summary of TSO/E Services

Task	Service
Invoking TSO/E service routines	CALLTSSR macro instruction
Checking the syntax of subcommand names	Command scan service routine
Checking the syntax of command and subcommand operands	parse service routine
Controlling terminal functions and attributes	Terminal control macro instructions
Processing terminal I/O	BSAM and QSAM TSO/E I/O service routines TGET/TPUT/TPG macros TSO/E Message Handling Routine
Handling attention interruptions	STAX service routine CLIST attention facility
Obtaining a list of data set names	ICQGCL00
Ensuring that data sets contain enough space	Space management
Changing alternative library environments	Alternative library interface routine
Allocating, concatenating and freeing data sets	Dynamic allocation interface routine
Retrieving information from the system catalog	Catalog information routine
Constructing a fully-qualified data set name	Default service routine
Analyzing return codes	DAIRFAIL GNRLFAIL/VSAMFAIL

Table 2. Summary of TSO/E Services (continued)

Task	Service
Searching lists of authorized commands, as well as programs and commands not supported in the background	Table look-up service
Invoking commands, CLISTs, REXX execs and programs	TSO/E service facility
Accessing CLIST and REXX variables	Variable access routine
Retrieving information from the names directory	ICQCAL00
Displaying printers	Printer support CLISTs
Invoking Information Center Facility applications	Application invocation function
Retrieving system messages issued during a console session	GETMSG service
Establishing a TSO/E environment outside of the TSO/E TMP	TSO/E environment service

Syntax Notational Conventions

The following paragraphs describe the notation that this document uses to define the command syntax and format.

1. The set of symbols listed below is used to define the format. Do not type them when you enter the command.
   — hyphen
   _ underscore
   { } braces
   [ ] brackets
   ... ellipsis

   The special uses of these symbols are explained in the following paragraphs.
2. You can type uppercase letters, numbers, and the set of symbols listed below exactly as shown in the statement definition when you enter the command.
   ' apostrophe
   ***** asterisk
   , comma
   = equal sign
   ( ) parentheses
   . period
3. Lowercase letters and symbols appearing in a command definition represent variables for which you can substitute specific information when you enter the command.

   For example, if *name* appears in a command definition, you can substitute a specific value (for example, ALPHA) for the variable when you enter the command.
4. Hyphens join lowercase words and symbols to form a *single* variable.

   For example, if *member-name* appears in the command syntax, you should substitute a specific value (for example, BETA) for the variable when you enter the command.
5. The default option is indicated by an underscore. If you do not specify anything, you automatically get the default option. For example,

```
LOGOFF    [ DISCONNECT]
          [ HOLD      ]
```

indicates you can select DISCONNECT or HOLD. However, if no operand is specified, the default is DISCONNECT.

6. Braces group related items, such as alternatives. You must choose one of the items enclosed within the braces. For example,

```
CALL     {dsname              }
         {dsname (membername) }
```

indicates if you select dsname (membername), the result is CALL dsname (membername).

7. Brackets also group related items. However, everything within the brackets is optional and can be omitted. For example,

```
PROTECT    data-set-name[ PWREAD    ]
                        [ NOPWREAD  ]
```

indicates you can choose one of the items enclosed within the brackets or you can omit both items within the brackets.

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

```
DELETE (entryname[/ password][...])
```

indicates an entry name and associated optional password you can repeat any number of times in succession.

Part 2. Writing a command processor

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

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

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

If you choose to write your own command processors, you can use the command processors and service routines provided by TSO/E to perform many of the functions required by a command processor. The programming services available in TSO/E consist of service routines, macros, SVCs and CLISTs, and are discussed in *z/OS TSO/E Programming Services*.

Part 2, “Writing a command processor”of this document contains several chapters that describe what you must do to write, install, execute and test a . command processorChapter 2, “What is a command processor?,” on page 15 presents the concepts and terminology that you must understand before you read the later chapters. Chapter 3, “Writing a command processor,” on page 19outlines the steps to follow when writing a and refers you to later chapters for the details of each step. Read all of command processorChapter 2, “What is a command processor?,” on page 15 and Chapter 3, “Writing a command processor,” on page 19 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 a user at a terminal enters a command name. 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 terminal user, responds to abnormal terminations and processes attention interruptions.

The TSO/E Environment

When a user logs on to TSO/E, the program specified on the EXEC statement of the user's LOGON procedure is attached during logon processing as the TMP. After the logon is complete, the TMP writes a `READY` message to the terminal to request that the terminal user enters a command name. The TMP determines whether the user's response is a command name. If a command is entered, the TMP attaches the requested command processor and the command processor then performs the functions requested by the user.

The command processor parameter list (CPPL)

The interface between the TMP and an attached command processor is shown in Figure 1 on page 16.

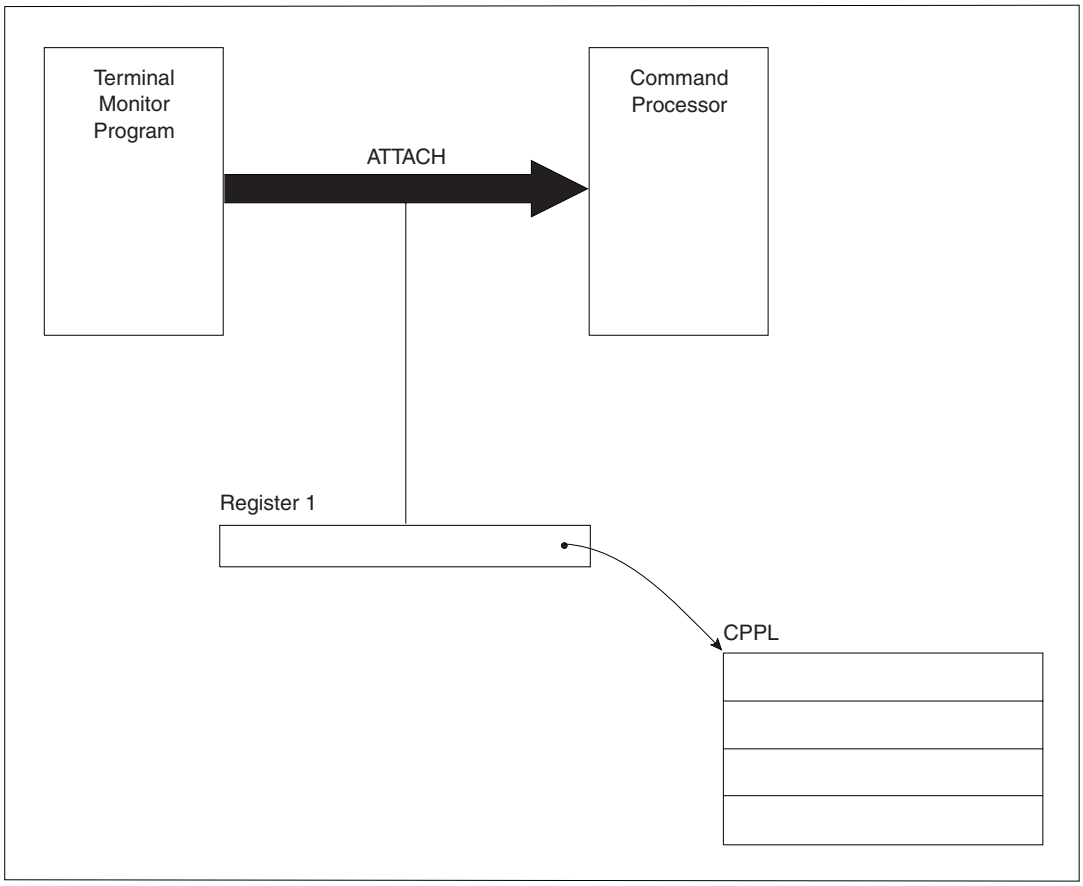


Figure 1. Interface between the TMP and a command processor

When the TMP attaches a command processor, register 1 contains a pointer to command processor parameter list (CPPL) containing addresses required by the command processor. The command processor parameter list CPPL is a four-word parameter list that is located in subpool 1. Table 3 describes the contents of the CPPL.

Table 3. The command processor parameter list (CPPL)

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

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 terminal user, the input is placed into the command buffer. After determining that the user has entered a command name, the TMP attaches the appropriate

command processor. Figure 2 shows the format of the command buffer.

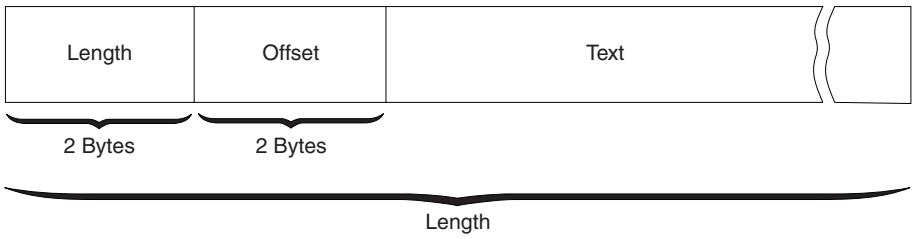


Figure 2. Format of the Command Buffer

When your command buffer 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 terminal 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.

Accessing the user profile table without a CPPL

A program that is invoked in a TSO/E environment may require information from the user profile table (UPT). For example, the language a user has selected is defined by the UPTLANG field of the UPT. The program, however, might not have access to the command processor parameter list (CPPL), which contains the UPT address. A program that does not have access to the CPPL can access the UPT though the protected step control block (PSCB). The PSCB contains the PSCBUPT field, which points to the UPT. MVS provides the mapping macro IKJPSCB for the PSCB.

You can access the PSCB by issuing the EXTRACT macro to request the address of the PSCB. For information about the EXTRACT macro, see *z/OS MVS Programming: Authorized Assembler Services Reference EDT-IXG*.

Figure 3 illustrates the two ways a program can access the PSCB.

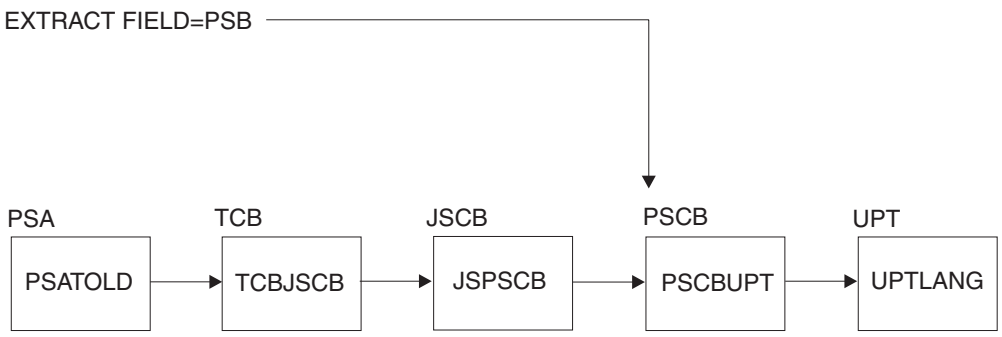


Figure 3. Chaining of Control Blocks and Fields to the UPT

Accessing the user profile table without a CPPL

Figure 3 on page 17 shows the control blocks and fields that support the path to the UPT. MVS provides the mapping macros for these control blocks. Table 4 relates the mapping macros to the control blocks shown in Figure 3 on page 17.

Table 4. Mapping Macros for Control Blocks that Chain to the UPT

Control Block	Mapping Macro
PSA	IHAPSA
TCB	IKJTCB
JSCB	IEZJSCB
PSCB	IKJPSCB
UPT	IKJUPT

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. For example, the first two operands for the RENAME command identify the data set to be renamed and specify the new name:

RENAME	**OLDNAME**	**NEWNAME**	**[ALIAS]**
command name	operand_1 (old data set name)	operand_2 (new data set name)	operand_3

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. The terminal 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.

In the example above, `OLDNAME` and `NEWNAME` are positional operands; `ALIAS` is a keyword operand. The braces around `ALIAS` indicate that the operand is not required.

The terminal 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*. The user requests one of the operations by first entering the name of the command, and then entering a subcommand to indicate which individual operation should be performed. For example, the TSO/E EDIT command has subcommands. After entering the EDIT command, the user can then enter the subcommands for EDIT.

Chapter 3. Writing a command processor

This topic describes the steps to follow when writing, installing and using a command processor. Further details are contained in subsequent topics.

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, and prompt the user if operands are incorrect or if required operands are missing. See Chapter 4, “Validating command operands,” on page 21 for a description of the functions provided by the parse service routine.
   - Communicate with the user at the terminal.

     Your command processor might need to obtain data from the terminal, prompt the user for input, and write messages or data to the terminal. You may also want to display full-screen panels. For information on terminal I/O and full-screen processing, see Chapter 5, “Communicating with the Terminal User,” on page 31.
   - 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/E to perform many functions. For example, your command processor can use the TSO/E service facility to invoke other commands, programs, CLISTs, or REXX execs.
   - 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 entered by the terminal user 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 45.
   - Intercept and process abnormal terminations.

     Your command processor must be able to intercept abnormal terminations and perform the processing needed to keep the system operable. For information on writing error handling routines, see Chapter 7, “Processing Abnormal Terminations,” on page 49.
   - Respond to and process attention interruptions entered from the terminal.

     If your command processor accepts subcommands or operates in full-screen mode, it must be able to respond to an attention interruption entered by the terminal user. Your command processor must provide an attention exit to

obtain a line of input from the terminal after an attention interruption occurs. For more information, see Chapter 8, “Processing Attention Interruptions,” on page 53.

- 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. CLISTs that invoke your command processor can check the return code, which is contained in the variable &LASTCC, to determine whether processing was successful. Your command processor can set one of the following return codes in register 15:

Table 5. Command processor return codes for 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. Note that an error does not occur when the command processor is able to obtain the required information by prompting the user.

2. **Create HELP information.**

   If you plan to make your command processor available to other TSO/E users, provide HELP information about the command, subcommands, and all operands. HELP information is displayed at the terminal when the user enters the HELP command and specifies the name of the command or subcommand. See Chapter 9, “Creating HELP Information,” on page 61.

3. **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. For more information, see Chapter 13, “Compiling and Assembling Programs,” on page 77.

4. **Install the command processor .**

   For methods that you can use to add your new command processor to TSO/E, see Chapter 10, “Installing a command processor,” on page 67.

5. **Test the command processor and correct errors.**

   See Chapter 11, “Executing and Testing a command processor,” on page 69.

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 topic 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 *z/OS TSO/E Programming Services*.

Using the parse service routine

When you write a command processor to run under TSO/E, you need a method to determine whether the command operands specified by the user are syntactically correct. The parse service routine (IKJPARS) checks syntax by searching the command buffer for valid operands. If a required operand is missing, or if the user enters an operand incorrectly, parse can prompt the user. The user can enter question marks to receive any second-level messages that are supplied by your command processor that are associated with the operand. Second-level messages provide more explanation of the initial message. Parse can also display HELP information for an operand after the second-level messages have been issued.

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 when they follow all of the positional operands.

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

Before you invoke the parse service routine, your command processor must use the parse macro instructions to create a parameter control list (PCL). The PCL describes the permissible operands. Next, call the parse service routine to compare the information that is supplied by your command processor in the PCL to the operands in the command buffer. Each acceptable operand must have an entry that is 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 calls the parse service routine, it must pass a parse parameter list (PPL). The PPL 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 server routine finishes processing, it passes a return code in register 15 to your command processor. Your command processor must issue meaningful error messages for all non-zero return codes. To issue meaningful error messages for all non-zero return codes, use the GNRLFAIL routine, in *z/OS TSO/E Programming Services* .

Figure 4 shows the interaction between the command processor and the parse service routine.

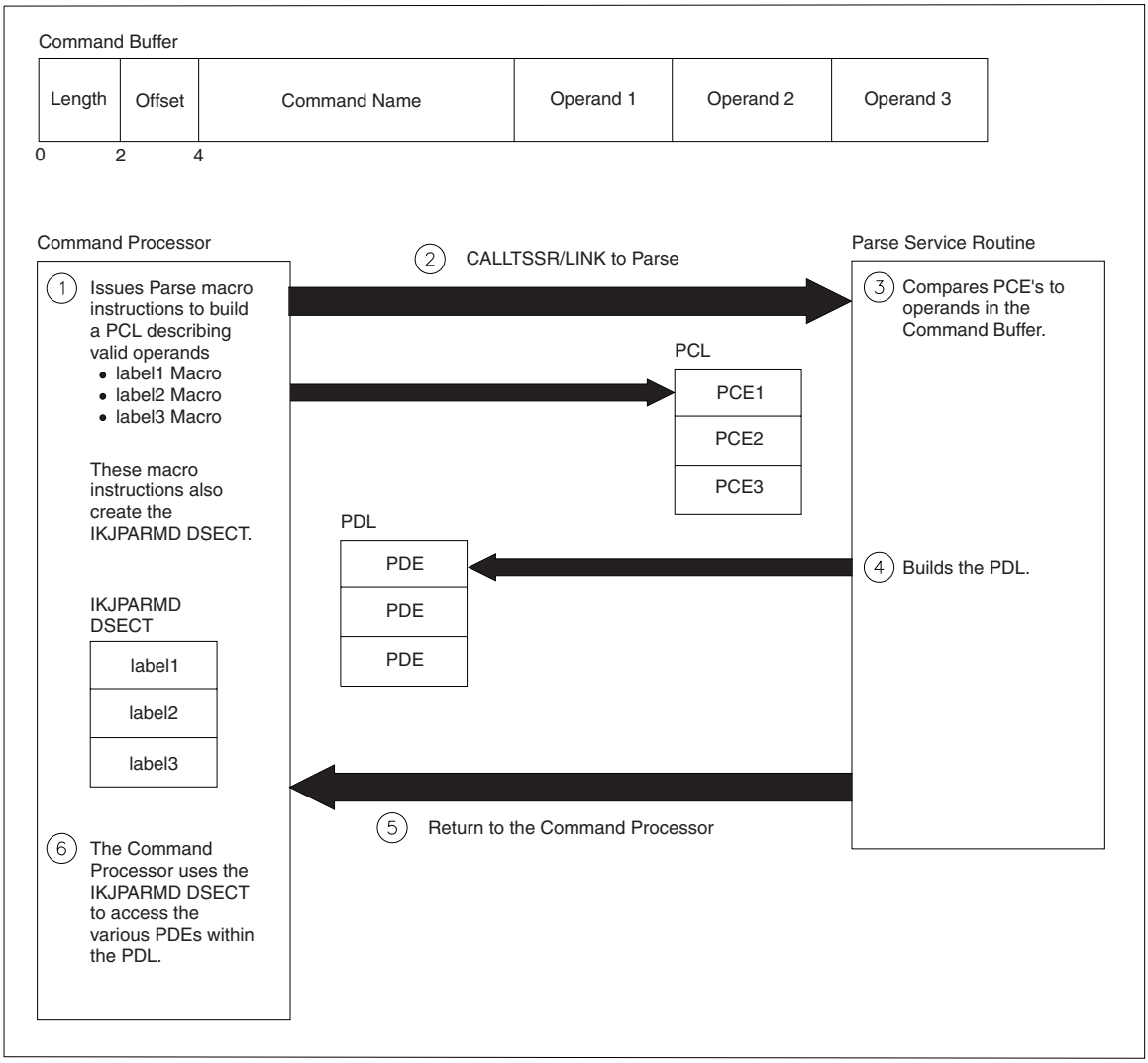


Figure 4. A command processor that is using the parse service routine

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.

Checking unidentified keyword operands

For certain keyword operands, you might want to bypass the syntax checking facility of the parse service routine and provide an alternate method of determining the validity of the operand. To accomplish this, use the parse macro instruction IKJUNFLD and provide a verify exit routine to determine if the operand is valid.

Use the IKJUNFLD macro instruction to indicate that the parse service routine can accept an unidentified keyword operand that is present in the command buffer. An unidentified keyword operand is an operand that is not defined in the parameter control list (PCL). This macro can also be used to indicate that parse can accept unidentified keyword operands within a subfield.

If you code the IKJUNFLD macro instruction, parse accepts unidentified keyword operands, but does not perform any validity checking on them. Your command processor must supply a verify exit routine to perform checking on these operands. Indicate that a verify exit routine is to receive control by coding the entry point address of the routine on the IKJUNFLD macro instruction.

When parse passes control to a verify exit routine, it passes a verify exit parameter list, which contains the address of a parse parameter element that parse built to describe the operand being processed. Your verify exit routine can use this information to examine the operand and determine its validity.

When processing is complete, the verify exit 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 check and determines the action that parse takes.

Using the prompt mode HELP

When the parse service routine prompts the user to enter a required operand, or to reenter a syntactically incorrect operand, the user can enter question marks to receive second-level messages associated with the operand. If a question mark is entered and no second-level messages were provided, or they have all been issued in response to previous question marks, parse determines whether it can generate a valid HELP command to provide the user with additional information.

Whether parse can generate a HELP command depends upon the setting of the ECTNOQPR bit in the environment control table (ECT). The ECT is pointed to by the command processor parameter list (CPPL) that is passed to your command processor when it receives control. For information on the ECT, see Table 3 on page 16.

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

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

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

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

To make this function available for your command processor, create a HELP member as described in Chapter 9, “Creating HELP Information,” on page 61.

A sample command processor

The sample command processor in Figure 5 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.

Figure 5. A sample command processor

```
PROCESS  TITLE 'SAMPLE TSO/E COMMAND PROCESSOR                    '
PROCESS  CSECT ,
PROCESS  AMODE 31                    COMMAND'S ADDRESSING MODE
PROCESS  RMODE 31                    COMMAND'S RESIDENCY  MODE
*********************************************************************
*                                                                   *
* TITLE - PROCESS                                                   *
*                                                                   *
* DESCRIPTION - SAMPLE TSO/E 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
         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
*
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

********************************************************************
* 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    *
* TERMINAL AND PROMPT THE USER TO RE-ENTER THE DATA SET NAME.      *
*                                                                  *
```

```
* 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
         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
*********************************************************************
*                                                                   *
* PARSE MACROS USED TO DESCRIBE THE COMMAND OPERANDS                *
*                                                                   *
*********************************************************************
*
PCLSTART DS       0H
PCLDEFS  IKJPARM  DSECT=PRDSECT
DSNPCE   IKJPOSIT DSNAME,                                          +
               PROMPT='THE NAME OF THE DATA SET YOU WANT TO PROCESS.  +
               ENTER '?'' FOR HELP',                               +
               HELP=('A DATA SET NAME WHICH HAS A FIRST-LEVEL QUALIFIER+
                OTHER THAN 'SYS1'.'),                             +
               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

*********************************************************************
*                                                                   *
*  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
*
         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
```

Sample command processor

```
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
```

Chapter 5. Communicating with the Terminal User

Your command processor may need to obtain data from the terminal, prompt the user for input, or write messages or data to the terminal. You may also want to use the full-screen capabilities of TSO/E to display full-screen panels.

This topic provides an overview of how to issue messages, perform terminal I/O, change the source of input, and use the full-screen capabilities of TSO/E in your command processor. For additional information on the macros and services discussed in this topic, see *z/OS TSO/E Programming Services*.

Issuing Messages

TSO/E supports three classes of messages:

1. *Prompting messages* begin with "ENTER" or "REENTER", and require a response from the user. For example, prompting messages are issued by the parse service routine when the user has entered an incorrect operand or when a required operand is missing.

   Issue prompting messages from your command processor to obtain data from the terminal when additional information is required to perform the requested function.
2. *Mode messages* inform the terminal user which command is in control and indicate that the system is waiting for the terminal user to enter a new command or subcommand. For example, the READY message is a mode message.

   If you have chosen to implement subcommands, your command processor can issue a mode message to inform the terminal user that the system is waiting for the user to enter a subcommand.
3. *Informational messages* are issued for information only, and do not require a response from the user. Issue informational messages to notify the terminal 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 a TSO/E user should usually have *second-level messages* associated with them. Second-level messages provide additional explanation of the initial message. They are displayed only if the user specifically requests them by entering a question mark (?).

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/E to issue messages and obtain the user's response.

Use the PUTLINE service routine, which writes a line of data to the terminal, to display informational messages. Use the PUTGET service routine, which writes a line of data to the terminal and obtains a line of input in response, to issue prompting and mode messages.

Both PUTLINE and PUTGET provide support for displaying messages in the user's desired language as specified in the user profile table. The MVS message service must be active to use this support. In addition, a translated version of the message in the specified language must exist.

You can also use PUTLINE and PUTGET to perform the following functions when issuing prompting or informational messages:

- Remove message identifiers before issuing the message. This is done if the terminal user has used the PROFILE command to indicate that message identifiers are not to be displayed.
- Place inserts into message text.
- Chain second-level messages.

The PUTLINE service routine provides special support for second-level messages that are associated with an informational message. Because many informational messages might be displayed at the terminal before the user enters a question mark, PUTLINE saves them for you. The area in which they are saved exists from one PUTGET to another. In other words, whenever the user can enter a new subcommand, the user can enter a question mark instead, requesting all the second-level messages for informational messages issued during execution of the previous subcommand. If the user does not enter a question mark, PUTGET deletes the second-level messages and frees the area they occupy.

Mode messages cannot have second-level messages, because a question mark entered in response to a mode message is defined as a request for the second levels of previous informational messages. If your command processor supports subcommands, you should use the PUTGET service routine to issue all mode messages so that the second level informational messages are properly handled.

When PUTGET returns a line of data from the terminal, 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/E Message Issuer Routine (IKJEFF02)

If your command processor issues messages with numerous inserts, you should use the TSO/E 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 or prompting messages.

IKJEFF02 also provides support for displaying messages in the user's desired language as specified in the user profile table. The MVS message service must be active to use this support. In addition, a translated version of the message in the specified language must exist.

Using generalized routines for issuing messages

If your command processor invokes TSO/E 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/E 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 Terminal I/O

Your command processor may need to write lines of data to the terminal or obtain data from the terminal. This topic discusses how to perform terminal I/O for data other than messages, message responses, and subcommand requests.

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

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

  The major benefit of using BSAM or QSAM to process terminal I/O is that these access methods are not device dependent or TSO/E 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 terminal and write data to the terminal, 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 terminal.
- **The TGET, TPUT, and TPG macro instructions** to perform terminal I/O. Your command processor can use the TPUT macro instruction to write a line of output to the terminal, and can use the TGET macro instruction to read a line of input. However, the TGET, TPUT and TPG macro instructions are intended only for terminal I/O. To allow your command processor to be executed in a background TSO/E session, use the I/O service routines (STACK, GETLINE, PUTLINE and PUTGET).

Changing Your command processor's Source of Input

TSO/E maintains a pushdown list or stack that determines the source of input.

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

You may want to obtain input from a source other than the terminal, such as a data set containing records to be processed. TSO/E also allows an *in-storage list* to be used as the source of input. An in-storage list can be either a command procedure (CLIST) 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.

Writing a Full-Screen command processor

If your command processor needs to display panels, it must be able to issue full-screen messages to the terminal and obtain input from the user. When your command processor displays full-screen messages, it must prevent the screen from being overlaid by non-full-screen messages, such as messages sent by the system operator or other TSO/E users. A full-screen command processor must also provide the necessary processing to allow the terminal user to read non-full-screen messages before they are overlaid by full-screen messages.

Use one of the following methods to write a full-screen command processor:

- If your command processor is to execute in an ISPF environment, use ISPF services to receive requests and data from a terminal user and give appropriate responses. For information on using ISPF services, see *z/OS ISPF Services Guide*.
- If your command processor is to be used outside of an ISPF environment, or if it must perform functions not available through ISPF services, use VTAM® macros for full-screen processing.

This topic outlines the steps for writing a full-screen command processor using VTAM macros, and contains several examples illustrating the processing that occurs when running a full-screen command processor. You must use the following macros when writing a full-screen command processor:

STFSMODE
: Set full-screen mode.

STLINENO
: Set the line number.

STTMPMD
: Set terminal display manager options.

TGET Get a line from the terminal.

TPUT Write a line to the terminal.

See *z/OS TSO/E Programming Services* for a complete description of each of these macros.

Follow these steps when writing a full-screen command processor:

1. Set full-screen mode on (see “Set Full-Screen Mode On” on page 35).
2. If replacing a display terminal manager, such as Session Manager, put the command processor in control (see “Give Control to the command processor” on page 36).

3. Write to and get information from the terminal as necessary (see “Write to and Get Information from the Terminal” on page 36).
4. Exit and reenter full-screen mode as necessary (see “Exiting and Reentering Full-Screen Mode” on page 37).
5. Terminate the full-screen command processor and, if it replaced a display terminal manager, return control to the display terminal manager (see “Full-Screen command processor Termination” on page 38).

Table 6 shows the macros used when writing a full-screen command processor.

Table 6. Macros Used to Write a Full-Screen command processor

(1)	STFSMODE ON,INITIAL=YES	Set full-screen mode on
(2)	STTMPMD ON	Give control to the command processor
(3)	TPUT FULLSCR	Issue a full-screen message
	TGET ASIS	Get input from the terminal
	.	
	.	
	.	
(4)	STLINENO LINE=1	Clear the screen and exit full-screen mode expecting to reenter it later
	.	
	.	
	.	
	TPUT EDIT	Issue a non-full-screen message
	.	
	.	
	.	
	STFSMODE ON	Reenter full-screen mode
	TPUT FULLSCR	Issue a full-screen message
	TGET ASIS	Obtain RESHOW request
	TPUT FULLSCR	Reissue the previous full-screen message
	TGET ASIS	Get input from the terminal
	.	
	.	
	.	
(5)	STLINENO LINE=1	Clear the screen
	STFSMODE OFF	Exit full-screen mode and set defaults
	STTMPMD OFF	Return control to the display terminal manager
	TPUT EDIT	Display session summary information
	.	
	.	
	.	

Set Full-Screen Mode On

Use the STFSMODE macro to set full-screen mode on. This macro prevents unexpected non-full-screen messages from overlaying the screen. For example, unexpected messages from the operator or from other TSO/E users could cause incorrect input to be sent to the command processor. Also, STFSMODE prevents full-screen messages from overlaying unexpected non-full-screen messages before the user has a chance to read them.

To prevent unnecessary protection of the screen contents, specify INITIAL=YES when you use the STFSMODE macro. If you specify INITIAL=YES and the first message is a full-screen message, TSO/VTAM does not display three asterisks at the terminal (which would require the user to press the Enter key). TSO/VTAM sets the INITIAL keyword indicator to NO after the command processor sends the first full screen of information. For subsequent full screens of output that follow non-full screens of output, TSO/VTAM displays the three asterisks at the terminal before processing the full-screen output. For a description of the processing that takes place when INITIAL=YES and INITIAL=NO, see “Examples of Full-Screen command processor Operation” on page 38.

TERMINAL BREAK Support for Full-Screen Mode

When a command processor establishes full-screen mode, VTAM treats all devices as if the terminal user had entered the TERMINAL NOBREAK command. If the user specifies TERMINAL BREAK before a full-screen command processor is invoked, VTAM supports the BREAK mode before the command processor enters full-screen mode and whenever the command processor exits from full-screen mode. See *z/OS TSO/E Command Reference* for a description of the TERMINAL command.

Give Control to the command processor

If your command processor replaces a display terminal manager, such as Session Manager, use the STTMPMD macro to put the command processor in control. If you do not use this macro, the display terminal manager traps line-mode messages so the user does not see them in the ordinary way. If your command processor does not replace a display terminal manager, you do not need this macro.

Write to and Get Information from the Terminal

Use the TPUT and TGET macros to provide interaction between the user and the command processor TPUT FULLSCR, TPUT NOEDIT, and TPG transmit a full-screen of output to the terminal.

Unlocking the Keyboard

When a command processor issues a TGET following a TPUT FULLSCR, VTAM unlocks the display keyboard. When a command processor issues a TGET following a TPUT NOEDIT or a TPG, VTAM does not unlock the keyboard. Programs that use TPUT NOEDIT and TPG are responsible for all device command and write-control-character bit settings.

Receiving Data

TGET ASIS reads a full screen of input containing the user's reply from the terminal. You can also use the NOEDIT keyword on the STFSMODE macro along with the TGET macro to get a full-screen message from the terminal.

NOEDIT Mode

To obtain a full screen of input (via a TGET macro) that is not edited in any way, the command processor can specify the NOEDIT keyword on STFSMODE. Regardless of the options the command processor specifies on the TGET macro, in NOEDIT mode, VTAM does not edit the data, break it into separate input lines, or modify it. VTAM receives the input from the terminal and puts it on the input queue intact. To establish NOEDIT mode, the command processor must issue:

```
STFSMODE ON,NOEDIT=YES
```

Use of the NOEDIT keyword has no effect on the treatment of TPUTs and TPGs.

Considerations for Invoking an External Function:

Before a command processor calls an external function or system service, it must check that the full-screen mode and input mode (normal or NOEDIT) are acceptable to the function or service. For example, if the NOEDIT input mode is in effect when control passes to an external routine, and the invoked routine does not change the input mode, TGET returns data in an unedited format.

Protection of Screen Contents

When non-full-screen messages are issued in full-screen mode, TSO/VTAM clears the screen and sends the non-full-screen messages to the screen. When the next full-screen message is issued, TSO/VTAM protects the screen contents to allow the

user time to read the non-full-screen messages. TSO/VTAM protects the screen by displaying three asterisks (***) after the last non-full-screen message and unlocking the keyboard. When finished reading the screen, the user presses Enter to allow full-screen processing to resume.

Restoration of Screen Contents

As part of the screen protection function, TSO/VTAM discards the full-screen message that immediately follows non-full-screen messages, unless issued with the HOLD option of the TPUT macro. To receive the RESHOW code, the full-screen command processor must issue a TGET macro after every TPUT FULLSCR macro.

The RESHOW indicator tells the command processor to completely restore the screen contents. That is, the command processor must reissue the previous full-screen message. For an example of RESHOW processing, see "Examples of Full-Screen command processor Operation" on page 38.

RESHOW requests can come from VTAM and from terminal users. Terminal users can request a restoration of the screen by pressing the RESHOW key. The VTAM default RESHOW code is X'6E', which represents the PA2 key. If the command processor uses a PF key for the RESHOW key, it must specify the RSHWKEY keyword on the STFSMODE macro when it first turns on full-screen mode. To set the RESHOW key, issue:

```
STFSMODE ON,RSHWKEY=n
```

where *n* is the PF key number. VTAM uses the hexadecimal representation of the specified PF key as the RESHOW code.

Exiting and Reentering Full-Screen Mode

If the command processor issues non-full-screen messages (or invokes routines that issue non-full-screen messages), it can issue the STLINENO macro to set full-screen mode off and to set the line number for the next non-full-screen message. In so doing, the command processor eliminates the screen protection function and determines where the next non-full-screen message appears. If the line number is set to 1, VTAM clears the screen. When the command processor issues the last non-full-screen message (or when the invoked routine returns control to the command processor), the command processor must issue STFSMODE ON to reestablish full-screen mode. The command processor must issue the STFSMODE macro before it issues the next full-screen message macro.

If the command processor exits full-screen mode, expecting to reenter full-screen mode at a later time before termination, the command processor must use STLINENO to set full-screen mode off. (Use of STFSMODE to set the mode off results in the RESHOW key being set to the default.) After a TGET request, the command processor can issue:

```
STLINENO LINE=n
```

where *n* is the desired line number. The command processor not have to specify MODE=OFF on the STLINENO macro because that is the default for the MODE keyword.

When all non-full-screen messages are completed, issue STFSMODE ON before issuing the next full-screen message macro. When the command processor returns to full-screen mode, it must issue the TGET macro to read the RESHOW request in the input queue. It can then continue to transmit and receive information from the terminal.

You may want either to clear part of the screen before issuing STLINENO, or to display information that is to remain on the screen after the STLINENO macro is issued. In either case, issue a full-screen TPUT or TPG macro (including the HOLD option) before issuing the STLINENO. The HOLD option specifies that the program that issued the TPG macro cannot continue its processing until the output line is written to the terminal or deleted. Therefore, the full-screen message reaches the terminal before the STLINENO macro takes effect.

Clearing the Terminal Screen

Because VTAM clears the screen when the line number is set to 1, STLINENO LINE=1 is an efficient way for the command processor to clear the screen. Use of a full-screen TPUT or TPG macro (including the HOLD option) to clear the screen reduces performance because it causes a swap-out of the address space to wait for the I/O to complete.

Full-Screen command processor Termination

When a TGET is satisfied with data that causes the command processor to begin exit processing, the following termination procedure is recommended:

STLINENO LINE=1
Causes VTAM to clear the screen.

STFSMODE OFF
Exits full-screen mode and resets the RESHOW key and NOEDIT mode to the defaults.

STTMPMD OFF
Returns control to a display terminal manager, such as Session Manager.

Non-full-screen TPUTs
Optional macros that provide session summary information or other types of termination information.

If the command processor issues a TPUT or TPG macro before (or instead of) issuing the STLINENO macro, it must use the HOLD option to guarantee that the message reaches the terminal before VTAM sets full-screen mode off. If the macro is handling a full-screen message, the command processor must issue a TCLEARQ INPUT macro just before termination to clear the RESHOW code that VTAM put on the input queue for screen protection.

Examples of Full-Screen command processor Operation

Examples show these functions:

- RESHOW in full-screen message processing
- INITIAL=YES on the STFSMODE macro when the first message is a full-screen message
- INITIAL=YES on the STFSMODE macro when the first message is a non-full-screen message
- INITIAL=NO on the STFSMODE macro

Each example consists of a figure followed by an explanation. The heading for each figure lists the three components involved in the processing: the command processor, TSO/VTAM, and the terminal. The items listed under each component relate to that component. The numbers in the left-hand column of the figures refer to the events described in the explanation. The arrows in the figure indicate the flow of the processing.

Function of RESHOW in Full-Screen Message Processing

Figure 6 shows the use of RESHOW when a command processor, operating in full-screen mode, issues a full-screen message while non-full-screen messages are being displayed at the terminal.

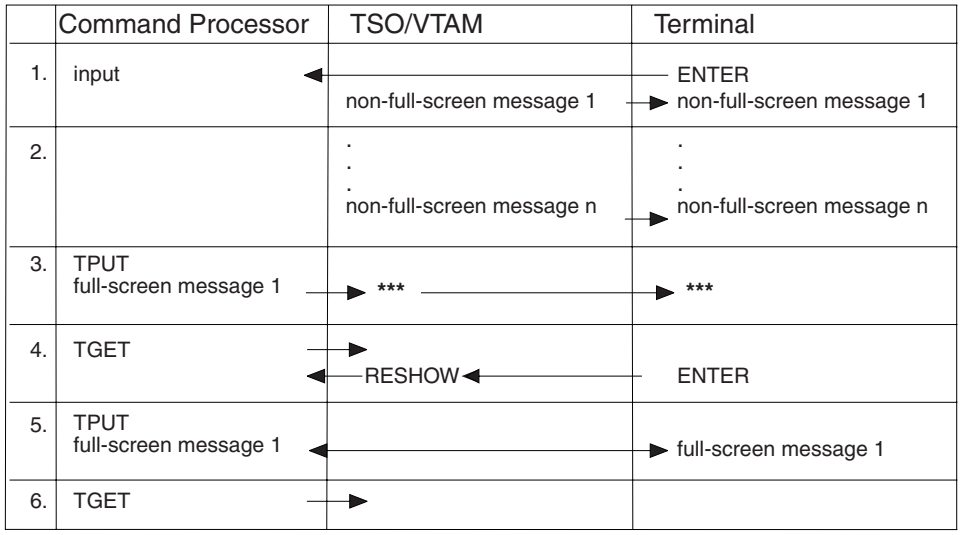


Figure 6. Function of RESHOW in Full-Screen Message Processing

The following events occur in Figure 6:

1. When the user presses the Enter key to send input to the command processor, TSO/VTAM:
   - Clears the screen.
   - Sounds the alarm (if the terminal has an alarm).
   - Displays non-full-screen messages. The operator or some other user could have sent these messages.
2. As long as TSO/VTAM receives non-full-screen messages, it displays them, one after another on the screen.
3. The command processor's normal processing of input (see step 1) may cause it to send a full-screen message using the TPUT macro. When TSO/VTAM receives the full-screen message, it:
   - Displays three asterisks (***) at the terminal.
   - Unlocks the keyboard to ensure that the user has time to view the non-full-screen messages.
   - Discards the full-screen message that the command processor sent.
4. After each full-screen message, the command processor issues a TGET macro. When the user presses the Enter key to acknowledge having seen the non-full-screen messages, TSO/VTAM puts a RESHOW request on the input queue to tell the command processor to completely restore the screen contents. The command processor's current TGET picks up this RESHOW request.
5. The command processor responds to the RESHOW request by issuing a full-screen TPUT to restore the screen contents. TSO/VTAM displays the message at the terminal.
6. The command processor issues a TGET macro.

Function of INITIAL=YES when the First Message is Full Screen

Figure 7 shows a situation in which the command processor specifies INITIAL=YES on the STFSMODE macro and issues a full-screen message as the first message.

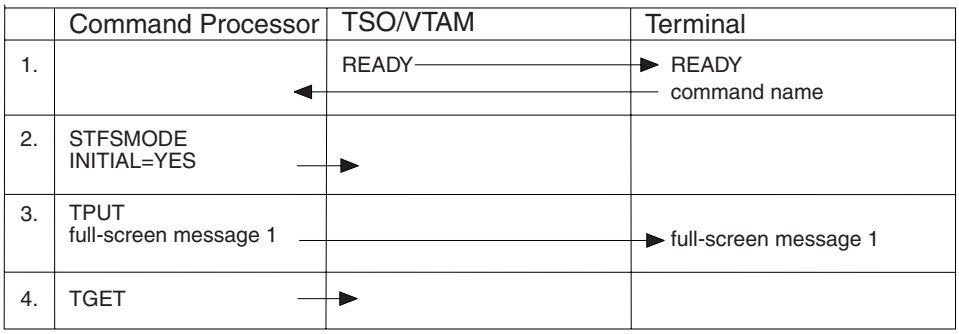


Figure 7. Function of INITIAL=YES when First Message is Full-Screen

The following events occur in Figure 7:

1. TSO/VTAM displays the READY message at the terminal. In response to the READY message, the user enters a command name, such as ISPF. The command processor receives the command name.
2. The command processor issues the STFSMODE macro with INITIAL=YES.
3. The command processor issues a full-screen message to the terminal. TSO/VTAM sends the message without warning because the command processor specified INITIAL=YES and because its previous interaction with the terminal involved input, not output. There is nothing to protect.
4. The command processor issues a TGET macro.

Function of INITIAL=YES when the First Message is Non-Full-Screen

Example 1: If the command processor specifies INITIAL=YES on the STFSMODE macro, and the first message is a non-full-screen message, VTAM ignores the keyword and protects the screen contents. Figure 8 on page 41 shows this situation when the STFSMODE macro is issued before the non-full-screen message.

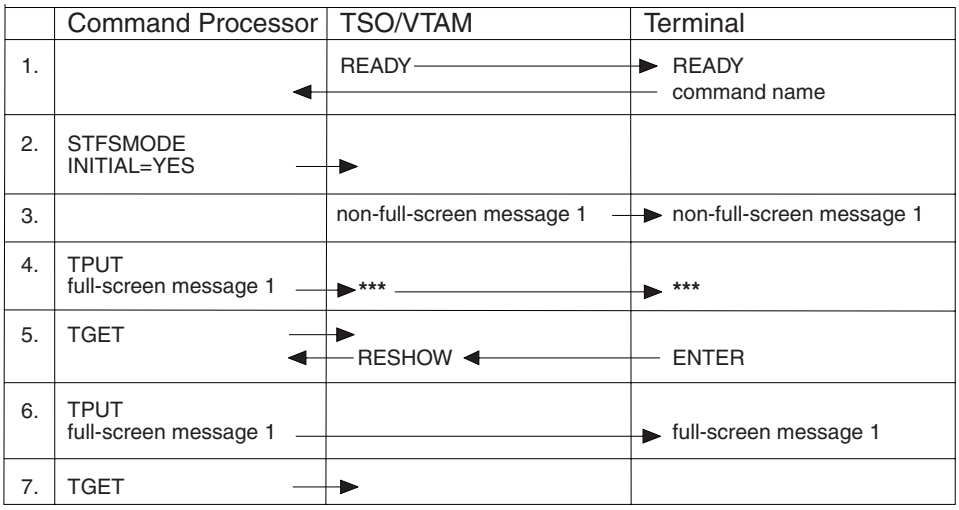


Figure 8. Function of INITIAL=YES when First Message is Non-Full-Screen, Example 1

The following events occur in Figure 8:

1. TSO/VTAM displays the READY message at the terminal. In response to the READY message, the user enters a command name. The command processor receives the command name.
2. The command processor issues the STFSMODE macro with INITIAL=YES.
3. TSO/VTAM displays a non-full-screen message. This could be a warning from the operator or a message from another user.
4. The command processor sends a full-screen message to the terminal. TSO/VTAM protects the screen contents by sending three asterisks to the terminal and discarding the full-screen message.
5. After each full-screen message, the command processor issues a TGET macro. When the user presses the Enter key to acknowledge having seen the non-full-screen message, TSO/VTAM puts a RESHOW request on the input queue to tell the command processor to completely restore the screen contents. The command processor's current TGET picks up the RESHOW request.
6. The command processor responds to the RESHOW request by issuing a full-screen message to restore the screen contents. TSO/VTAM displays the full-screen message at the terminal.
7. The command processor issues a TGET macro.

Example 2: If the command processor specifies INITIAL=YES on the STFSMODE macro, and the first message is a non-full-screen message, VTAM ignores the keyword and protects the screen contents. Figure 9 on page 42 shows this situation when the STFSMODE macro is issued after the non-full-screen message.

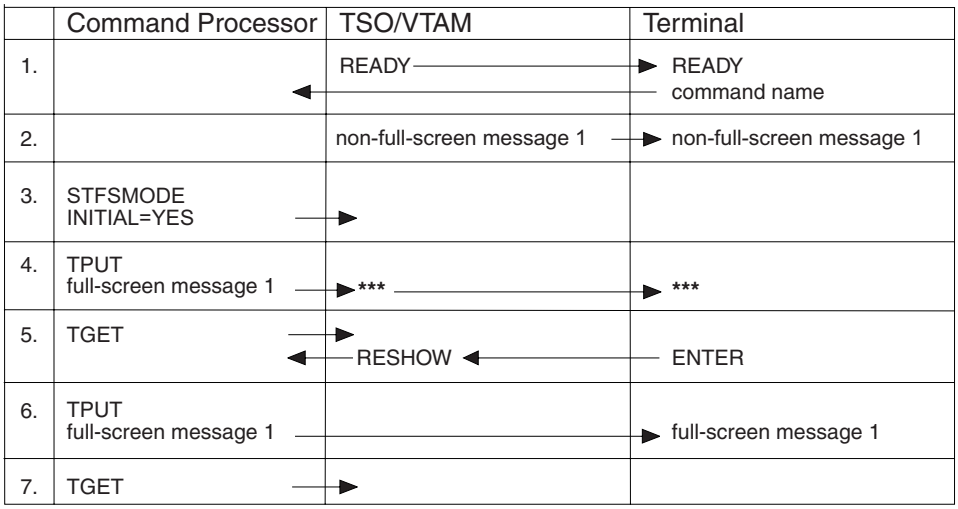


Figure 9. Function of INITIAL=YES when First Message is Non-Full-Screen, Example 2

The following events occur in Figure 9:

1. TSO/VTAM displays the READY message at the terminal. In response to the READY message, the user enters a command name. The command processor receives the command name.
2. TSO/VTAM displays a non-full-screen message. This could be a warning from the operator or a message from another user.
3. The command processor issues the STFSMODE macro with INITIAL=YES.
4. The command processor sends a full-screen message to the terminal. TSO/VTAM protects the screen contents by sending three asterisks to the terminal and discarding the full-screen message.
5. After each full-screen message, the command processor issues a TGET macro. When the user presses the Enter key to acknowledge having seen the non-full-screen message, TSO/VTAM puts a RESHOW request on the input queue to tell the command processor to completely restore the screen contents. The command processor's current TGET picks up the RESHOW request.
6. The command processor responds to the RESHOW request by issuing a full-screen message to restore the screen contents. TSO/VTAM displays the full-screen message at the terminal.
7. The command processor issues a TGET macro.

Function of INITIAL=NO

If the command processor specifies INITIAL=NO or INITIAL=NO is the default, TSO/VTAM protects the screen before displaying the first full-screen message. Figure 10 on page 43 shows an example of this situation.

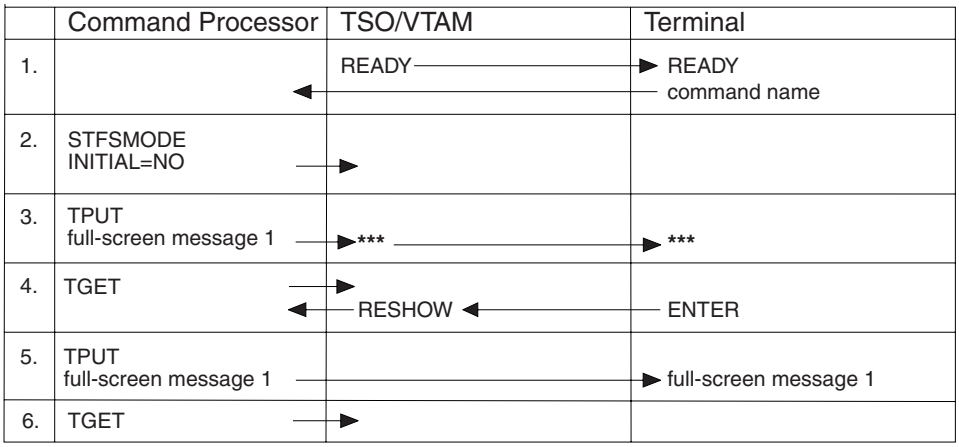


Figure 10. Function of INITIAL=NO

The following events occur in Figure 10:

1. TSO/VTAM sends a READY message to the terminal. In response to the READY message, the user enters a command name. The command processor receives the command name.
2. The command processor issues the STFSMODE macro with INITIAL=NO.
3. The command processor sends a full-screen message to the terminal. TSO/VTAM protects the screen contents by sending three asterisks to the screen and discarding the full-screen message that the command processor sent.
4. After each full-screen message, the command processor issues a TGET macro. When the user presses the Enter key, TSO/VTAM puts a RESHOW request on the input queue to tell the command processor to completely restore the screen contents. The command processor's current TGET picks up the RESHOW request.
5. The command processor responds to the RESHOW request by issuing a full-screen message to restore the screen contents. TSO/VTAM displays the full-screen message at the terminal.
6. The command processor issues a TGET macro.

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 entered by a terminal user and pass control to the requested subcommand processor. This topic outlines the steps you must follow to support subcommands.

Command scan, the PUTGET service routine and the parse service routine are discussed in this topic; see *z/OS TSO/E Programming Services* 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 completes.

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

Use the PUTGET service routine to issue a mode message to the terminal and return a line of input. A mode message informs the terminal user which command is in control and lets him know that the system is waiting for him to enter a new subcommand. For example, the `TEST` message issued by the TEST command processor is a mode message.

When PUTGET returns a line of data from the terminal, it places this data in an input buffer that is owned by your command processor. Figure 11 shows the format of the input buffer returned by the PUTGET service routine.

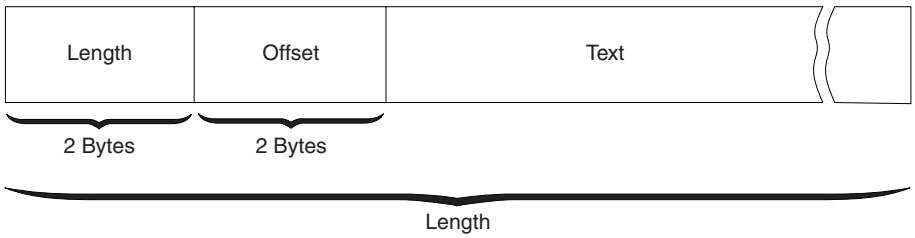


Figure 11. Format of the Input Buffer

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.

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 entered a valid subcommand name, use the ATTACH macro instruction to pass control to the requested subcommand processor.

You should code your ATTACH macro to specify that subpool 78 is to be shared with lower-level tasks.

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 49. For information on the ATTACH macro instruction, see *z/OS MVS Programming: Authorized Assembler Services Reference ALE-DYN*.

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, you must pass a pointer to a command processor parameter list.

Writing a subcommand processor

When you write a subcommand processor, it is suggested that you 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 entered with the subcommand using the parse service routine.
3. Communicate with the user at the terminal.
4. Perform the function of the subcommand according to any operands the user specified.
5. Intercept and process abnormal terminations.
6. Respond to and process attention interruptions entered from the terminal.
7. Set the return code in register 15 and return to the command processor.

These steps are discussed in more detail in Chapter 3, "Writing a command processor," on page 19.

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 *z/OS MVS Programming: Authorized Assembler Services Reference ALE-DYN*.

Step 4. Releasing the subcommand processor

Chapter 7. Processing Abnormal Terminations

Depending on the function and complexity of your command processor, you may need to provide error handling routines to process abnormal terminations (abends). This topic 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 a user's TSO/E session. 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.
- 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.

For information on writing ESTAE type exits, see *z/OS MVS Programming: Authorized Assembler Services Guide*.

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 to clear the input stack before returning to the TMP. Failure to clear the stack causes CLIST processing to be handled incorrectly.

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.

Figure 12 shows the relationship between the command processor, subcommand processor, and the error handling routines.

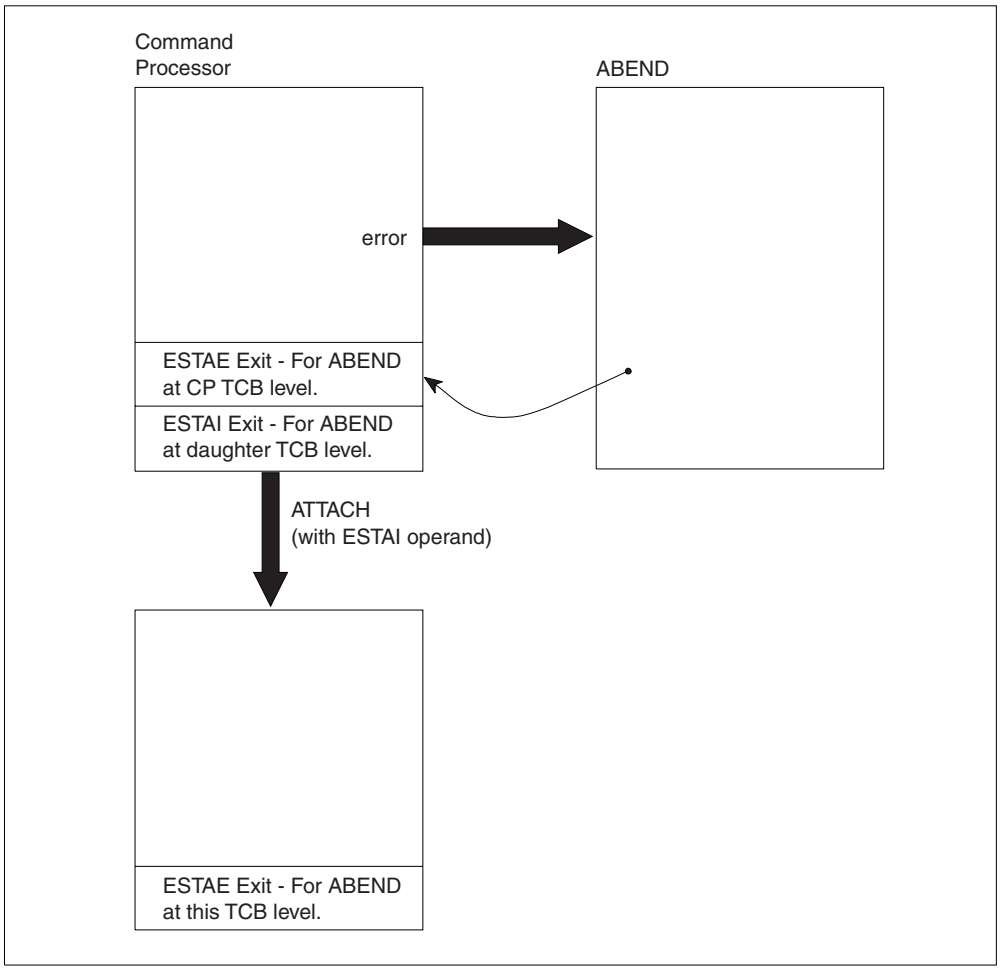


Figure 12. ABEND, ESTAI, ESTAE Relationship

Guidelines for Writing ESTAE and ESTAI Exit Routines

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. The ESTAE instruction should be issued before the STAX macro instruction is used to establish an attention exit routine.
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.

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.

Linkage Considerations

Your command processor can issue the ESTAE macro, and the ATTACH macro with the ESTAI operand, in either 24-bit or 31-bit addressing mode. The ESTAE and ESTAI exit and recovery routines receive control in the same addressing mode in which the corresponding macros are issued. When the macros are issued in 31-bit addressing mode, ESTAE and ESTAI routines can reside above 16 MB in virtual storage.

Chapter 8. Processing Attention Interruptions

This topic describes how TSO/E processes attention interruptions, and what you must do to provide an attention exit. Use the STAX service routine to specify and cancel attention exits, and defer the dispatching of attention exits. After reading this topic, see *z/OS TSO/E Programming Services* for information on the STAX service routine.

If your command processor accepts subcommands or operates in full-screen mode, provide an attention handling routine to receive control when a user enters an attention interruption from the terminal.

If your command processor does not establish its own attention exit, the TMP's attention exit receives control when a terminal user enters an attention interruption while the command processor is executing. Therefore, simple command processors should not establish an attention handling routine unless the routine provided by the TMP cannot process an attention interruption adequately.

Responding to Attention Interruptions

TSO/E interprets an attention interruption as a signal that the user wants to halt current program execution, possibly to request a new command or subcommand. If your command processor accepts subcommands, you must provide an attention handling routine to obtain a line of input from the terminal and respond to that input.

Use the STAX service routine to create the control blocks and queues necessary for the system to recognize and schedule installation exits that receive control as a result of attention interruptions. Your command processor provides the address of an attention exit to the STAX service routine by issuing the STAX macro instruction.

Use the STAX service routine to respond to an attention interruption that occurs during the processing of a CLIST that contains a CLIST attention exit. To establish an attention exit, issue the STAX macro with the CLSTATTN=YES and IGNORE=YES operands. This allows your attention exit to receive control and enables it to invoke the CLIST attention facility. The CLIST attention facility is a program and a recovery routine that any program can call to process a CLIST attention exit. For information on the CLIST attention facility, see *z/OS TSO/E Programming Services*.

How Attention Interruptions are Processed

An attention interruption is a way that a terminal user can interrupt the current processing that the system is performing. Two examples when a user wants to interrupt the current processing are:

- When a user specifies a command to the system but does not supply all required information, the command prompts the user for more information. At this point, the user can enter the required input or press the attention interrupt key. When the user presses the attention interrupt key, the system displays a "|" message (attention accepted) and immediately terminates the processing of the command.

- When a REXX exec has starting executing but before its completion, the user may decide to enter interactive debug mode and start tracing the REXX exec. In this case, the system displays a "|" message and the message `ENTER HI TO END, A NULL LINE TO CONTINUE, OR AN IMMEDIATE COMMAND.+`. In response, the user can enter REXX commands such as `HI` (halt interpretation), `TS` (trace start), or `TE` (trace end).

In both these cases, the system takes the appropriate action via an attention exit.

In an application program, you can indicate that you want to be notified when an attention interruption occurs to provide your own appropriate action in an attention exit. To do so, you issue the STAX macro instruction.

When you issue STAX, your request is queued, which means that your attention routine can get control along with the other previously identified attention routines in the system.

Remember that when the user presses the attention interrupt key and your exit gets control, it is possible that the user could press the attention interrupt key again. In this case, the system (by default) gives control to the next higher-level attention exit. This happens as long as the user continues to press the attention interrupt key and attention exits are identified to the system. When all attention exits have been exhausted, the user sees a `|I` message.

Similarly, you can invoke a program that in turn issues the STAX macro. In this case when the user presses the attention interrupt key, your attention exit only gets control when the user presses the attention interrupt key a second time.

Your application program has no control over lower-level (future) attention exits getting control. That is, you cannot inhibit an application program from issuing the STAX macro and processing attention interruptions before you get control. However, you can inhibit higher-level (previous) attention exits from getting control. To do so, you specify the TOPLEVEL=YES operand on the STAX macro. When you code TOPLEVEL=YES and the user presses the attention interrupt key while your exit is in control, the system displays the "|" message each time that the user presses the attention interrupt key, but it does not give control to a higher-level attention routine; your attention exit remains in control.

The preceding time-oriented view is only part of the order which the system uses to give control to attention exits. Actually, the current MVS task structure plays a part in the order in which attention routines get control. Normally, you need not be concerned with this, unless your application program attaches one or more subtasks that also issue the STAX macro. In this case, the following rules apply:

1. When you attach a subtask that issues the STAX macro, the attention exit that the subtask establishes gets control before your attention exit, regardless of the order in which your task and the subtask established the attention exits.

   This rule applies when you attach more than one subtask. The attention exit(s) for each of the subtasks receive control before your attention exit at the higher task.
2. When you attach more than one subtask, the time-oriented view takes effect for the order in which each of the subtasks' attention exits receive control.

   Note that the system uses the order in which each subtask issues the STAX macro, not the order in which you attach the subtasks, to control the order in which it gives control to attention exits. This means that you need to

synchronize the order that your subtasks issue the STAX macro to guarantee consistency each time that you attach multiple subtasks.

Some other considerations to remember, when establishing attention exits:

- You can establish more than one attention exit for a task. In this case, the time-oriented rule is used to give control to attention exits.
- To cancel the most recent attention exit at the current task level prior to task termination, you can issue the STAX macro without any operands.
- When a task terminates, either normally or abnormally, any attention exits that it established are removed from the system.
- When an attention exit terminates normally, control can be returned to the point of interruption. In this case, TGET, TPUT, and TPG buffers are flushed. If the program attempts to continue processing from the point of interruption, data in these buffers (an input or output record or an output message from the system) is lost.
- Depending on how many attention interrupts are being processed by the system, your attention request may be ignored by the system. In this situation, the system will send a "|A" to your terminal to indicate that your request will be ignored. You may wait a few seconds and retry your request; but do not press the attention key over and over again.

Deferring Attention Exits

After you have established an attention exit to the system via the STAX macro, it may become necessary for you to postpone the processing of an attention interruption until some processing completes. For example, you want to invoke a routine, and you do not want that routine interrupted by the user pressing the attention interrupt key. The STAX DEFER=YES operand gives you this ability. When you invoke STAX with the DEFER=YES operand, you in effect postpone any attention exits for the current task and any higher-level tasks.

After you invoke the STAX DEFER=NO operand, any attention interruption that is pending will be processed. That is, your previously identified attention exit will receive control.

System Use of STAX DEFER=YES

When the user presses the attention interrupt key, the system gives control to your attention exit at the task level at which the attention exit was established. However, before your attention exit gets control, the system will stop (make non-dispatchable) any of that task's subtasks. The only exception to this rule is if a system routine (that is, an SVRB) is currently running on any of the subtasks and did not specify a STAX DEFER=NO. In this case, the system defers the attention interrupt until the system routine completes or issues STAX DEFER=YES.

When your attention exit return control to the system, the system will start (make dispatchable) all of the subtasks under the task at which the attention exit executed.

If, for any reason, your attention exit requires one of the subtasks to be restarted, it is the responsibility of the attention exit to restart the task using the status start facility. Similarly, if the attention exit requires that the subtasks not be restarted on completion of the attention exit, it is the responsibility of the attention exit to use the status stop facility to ensure that the subtasks will not become dispatchable when the attention exit completes processing.

Note: When the system stops tasks within a tree structure, they will be stopped in an indeterminate order when any are deferring attention exits. As a result, care must be taken to control intertask dependencies and dependencies on scheduling attention exits. Failure to do so can result in an intertask deadlock that can only be relieved by canceling the TSO/E user.

Writing Attention Handling Routines

Use the STAX service routine in your command processor to provide the address of an attention exit routine that receives control when an attention interruption occurs.

When your attention exit routine receives control, you must issue a mode message to the terminal indicating the name of the program that was interrupted. You must then allow the user to enter a line of input. Use one of the following methods to accomplish this:

- Omit the IBUF or the OBUF operand from the STAX macro instruction that sets up the attention handling exit. Instead, use the PUTGET macro instruction, specifying the TERM operand, to send a mode message to the terminal identifying the program that was interrupted, and to obtain a line of input from the terminal.
- Specify the OBUF operand on the STAX macro instruction without an IBUF operand, or with an IBUF length of 0. The OBUF operand specifies the address of a buffer containing the text of the mode message to be issued to the terminal user who entered the attention interruption. Then issue the PUTGET macro instruction, specifying the ATTN operand. Use the ATTN operand to cause the PUTGET service routine to inhibit the writing of the mode message, because a message was already written to the terminal from the output buffer specified in the STAX macro instruction. The PUTGET service routine merely returns a logical line of input from the terminal.

In either of the above cases, if the user enters a question mark, the PUTGET service routine automatically causes the second level informational message chain (if one exists) to be written to the terminal, puts out the mode message again, and returns a line from the terminal.

Note: If you use the IBUF operand on the STAX macro instruction, no logical line processing or question mark processing is performed. If the user returns a question mark, you will have to use the PUTLINE macro instruction to write the second level informational message chain to the terminal. Then issue a PUTGET macro instruction, specifying the TERM operand, to write a mode message to the terminal and to return a line of input from the terminal.

Whether you use the IBUF operand on the STAX macro instruction or the PUTGET macro instruction to return a line from the terminal, you can use the command scan service routine to examine what the user has entered. Use command scan to determine that the line of input is syntactically correct in the input buffer returned by the PUTGET service routine, or in the attention input buffer (pointed to by the second word of the attention exit parameter list).

If the user enters a null line, the attention handling routine should return to the point of interruption. Note that, with the exception of the TPUT ASID buffers for TCAM, the TGET, TPG, and TPUT buffers are flushed during attention interruption processing. If any data was present in these buffers, it is lost.

If a new command or subcommand is entered, your attention handling routine must:

- Pass the input line to the mainline command processor for processing.
- Post the command processor's event control block to cause active service routines to return to the command processor.
- Exit.

If your command processor must reset the input stack, do so in the command processor mainline. A stack flush in an attention routine may cause severe errors.

Parameters Received by Attention Handling Routines

The parameter structure received by your attention exit routine is shown in Figure 13 on page 58.

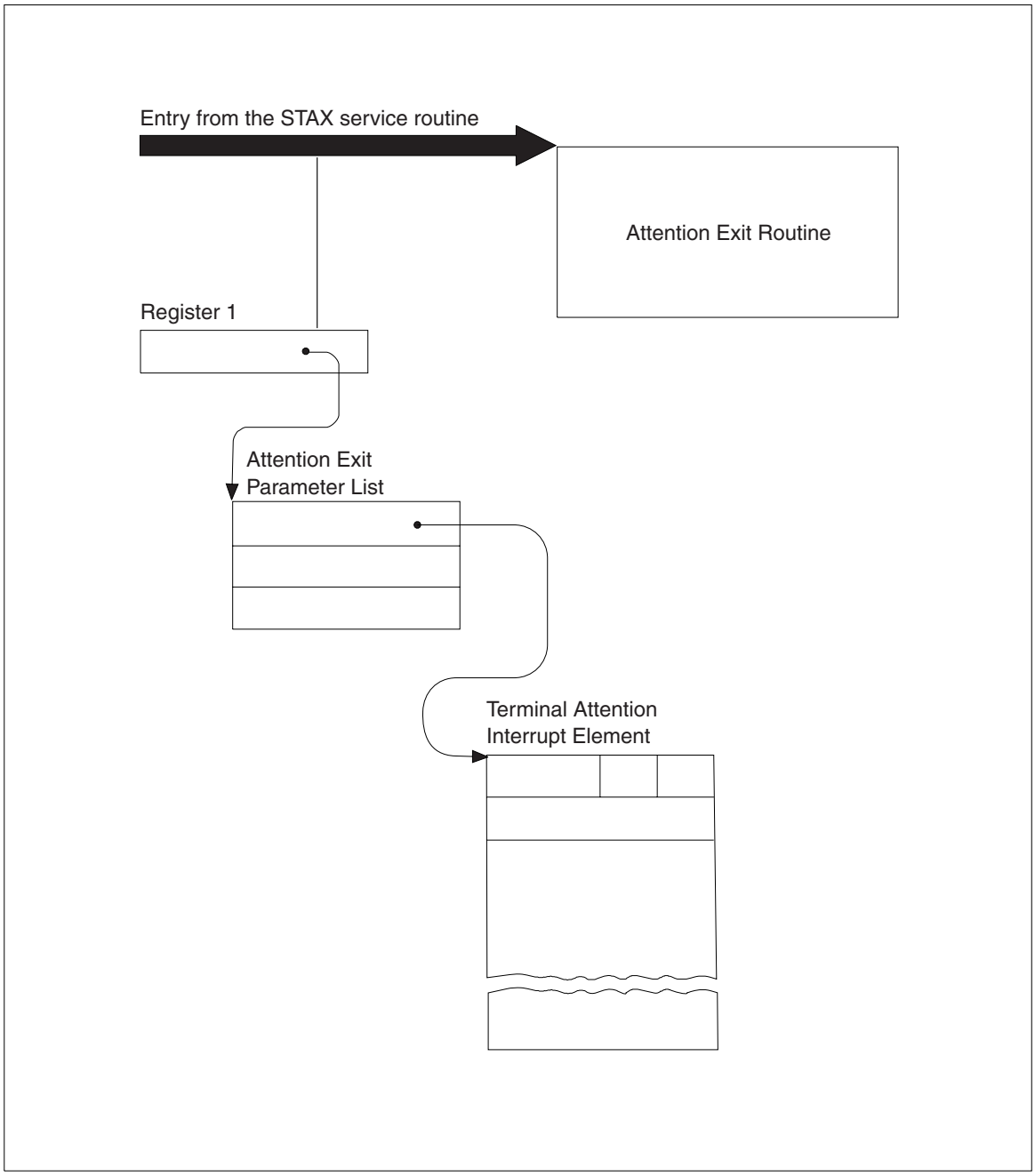


Figure 13. Parameters Passed to the Attention Exit Routine

When your attention handling routine receives control, the general registers contain the following information:

Register 0
Irrelevant

Register 1
The address of the attention exit parameter list

Registers 2–12
Irrelevant

Register 13
Save area address

Register 14
Return address

Register 15
Entry point address of the attention handling routine

The attention exit parameter list pointed to by register 1 contains the address of a terminal attention interruption element (TAIE).

The Attention Exit Parameter List

Table 7 shows the format of the attention exit parameter list.

Table 7. The Attention Exit Parameter List

Number of Bytes	Field Name	Contents or Meaning
4		The address of the terminal attention interrupt element (TAIE)
4		The address of the input buffer you specified as the IBUF operand of the STAX macro instruction. This field is zero if you did not include the IBUF operand in the STAX macro instruction.
4		The address of the user parameter information you specified as the USADDR operand of the STAX macro instruction. This field is zero if you did not include the USADDR operand in the STAX macro instruction.

The Terminal Attention Interrupt Element (TAIE)

The first word of the attention exit parameter list contains the address of an eighteen-word terminal attention interrupt element (TAIE). Table 8 shows the format of the TAIE. Use the IKJTAIE macro, which is provided in SYS1.MACLIB, to map the TAIE.

Table 8. The Terminal Attention Interrupt Element

Number of Bytes	Field Name	Contents or Meaning
2	TAIEMSGL	The length in bytes of the message placed into the input buffer you specified as the IBUF operand on the STAX macro instruction. This field is zero if you did not code the IBUF operand in the STAX macro instruction.
1	TAIETGET	The return code from the TGET macro instruction issued to get the input line from the terminal.
1	TAIEATTN	The terminal attention interrupt element flag that indicates whether the stack contains a CLIST attention exit. When this field is non-zero, a CLIST with an attention exit is in the stack.
4	TAIEIAD	The interruption address, which is the right half of the interrupted PSW. This is the address at which the program (or a previous attention exit) was interrupted.
64	TAIERSAV	The contents of general registers, in the order 0–15, of the interrupted program.

Full-Screen Protection Responsibilities of Attention Exit Routines

If you are writing a full-screen mode command processor, you should provide an attention exit routine to maintain screen protection when the user presses the PA1 or attention key. When the terminal user presses the PA1 or attention key, VTAM sets FULLSCR to OFF, the RESHOW key to the default, and NOEDIT mode to NO. If the command processor does not have an attention exit and the user presses the Enter key (in response to the attention indication), the command processor resumes execution at the point of interruption with these default values. If the command processor has an attention exit routine, the exit routine must issue the STFSMODE

macro to reestablish full-screen mode, the desired RESHOW key, and NOEDIT mode. In this way, the attention exit routine maintains screen protection.

Chapter 9. Creating HELP Information

If you write a command processor and make the command available to other users, you should also provide help information for the command. The help information you provide should include information about the command's function, syntax, operands, and messages. If the command processor has subcommands, the help information should also include a description of the subcommands and their operands.

You provide help information in a *help data set*. A help data set is a cataloged, partitioned data set consisting of individual members that contain help information for the commands. Each member contains help information for one command and its subcommands. The name of the member is the name of the command itself. For example, if you write a command processor named PRTDATA, you provide help information in the member named PRTDATA. The members of a help data set contain fixed-length 80-character records.

You can provide help information in the SYS1.HELP data set or in a data set that you or your installation creates. TSO/E provides the SYS1.HELP data set that contains help information for TSO/E commands and subcommands. You must be authorized to update SYS1.HELP.

If you use your own help data set rather than SYS1.HELP, you must define the data set to the system. You can concatenate your help data set to the SYS1.HELP data set or concatenate SYS1.HELP to your data set. Allocate the data sets to the system file SYSHELP. You can allocate the data sets either in a logon procedure or using the ALLOCATE command. For example, if your help data set is called MYCOMMS.HELPCOM, you can use the ALLOCATE command as follows:

```
ALLOCATE FILE(SYSHELP) DA('SYS1.HELP' MYCOMMS.HELPCOM) SHR REUSE
```

The help data sets that you concatenate to the SYS1.HELP data set do not have to have the same attributes as SYS1.HELP. However, the first concatenated data set must have the largest block size of the data sets and must specify a fixed block size.

You can also define your help data set using the IKJTSOxx member of SYS1.PARMLIB. In SYS1.PARMLIB, you use the HELP statement to define the help data sets for your installation. You must be authorized to update the IKJTSOxx member of SYS1.PARMLIB. Using SYS1.PARMLIB to define your help data sets is beneficial because you can then use the TSO/E PARMLIB command to dynamically list and change the HELP PARMLIB statement. For information about specifying help data sets using SYS1.PARMLIB, see *z/OS TSO/E Customization*.

One advantage of using the SYS1.HELP data set is that the system need only search one data set for help information. An advantage of using your own help data set is that you avoid the possibility of overlaying your help information when you install a new release of TSO/E.

In deciding how to provide help information, consider how the HELP command locates the information. When a user issues the HELP command to obtain help for a command or subcommand, HELP locates the information as follows:

- If the SYSHELP file is allocated when the user issues the HELP command, HELP searches only the data sets that are allocated to SYSHELP.
- If the SYSHELP file has not been allocated, HELP searches the data sets defined on the HELP statement in the IKJTSOxx member of SYS1.PARMLIB. Note that SYS1.PARMLIB is used only if no data sets have been allocated to the SYSHELP file.
- If HELP cannot locate the requested help information in the previous two situations, HELP searches the SYS1.HELP data set, if the data set has not already been searched.

One of the members in SYS1.HELP is the COMMANDS member. The COMMANDS member contains a list of the commands users can use and a brief description of each command. If you issue the HELP command without any operands, HELP displays the list of commands and their descriptions that are in the COMMANDS member.

If you provide help information for your own command processor, you may also want to add your command to the list of commands. To do this, you can simply add your information to the COMMANDS member. You can also place the information about your commands in a separate member of SYS1.HELP and include that member from the COMMANDS member. In the COMMANDS member, add the statement

```
)I member
```

where `member` is the name of the member containing your list of commands and command descriptions. An advantage of using your own member and including the member from COMMANDS is that you avoid the possibility of overlaying your information when you install a new release of TSO/E.

Writing HELP Members

If you choose to add HELP information to SYS1.HELP, use the IEBUPDTE utility program or the TSO/E EDIT command to update SYS1.HELP. SYS1.HELP is a system data set, so it will generally require operator intervention when it is updated.

To add a new member, MBRNAME, to a data set named PRIVATE.HELP using the EDIT command, enter:

```
edit 'private.help(mbrname)' data new
```

Format of HELP Members

Each of the HELP members, other than the COMMANDS member, contains the following categories of information, each of which can be displayed at the terminal:

Table 9. Categories of Information in HELP Members

Type of Information	Purpose
Subcommand list	Lists the names of subcommands. This appears only if the command has subcommands.
Functional description	Provides a brief description of the function of the command or subcommand.
Syntax	Describes the syntax of the command or subcommand.

Table 9. Categories of Information in HELP Members (continued)

Type of Information	Purpose
Message identifier description	Provides information pertaining to messages issued by the command or its subcommands.
Operand description	Provides information on the command operands. It includes individual sections containing brief descriptions of each positional operand, and of each keyword and its subfield operands.

Use the information described in Table 10 when you add to SYS1.HELP or set up your own HELP data set. The control characters, beginning in column 1, divide the data set into the categories described in Table 9 on page 62. These control characters allow the HELP command processor to display text according to the operands supplied on the HELP command. Each HELP data set member should contain at least the)F,)X, and)O control characters.

Table 10. Format of a HELP Data Set Member

Control Character	Purpose of Statement
)S	Indicates that a list of commands or subcommands follows.
)F	Indicates that the functional discussion of the command or subcommand follows.
)X	Indicates that the syntax description of the command or subcommand follows.
)M	Indicates that message ID information follows. The information is only printed by the HELP command when the MSGID keyword is specified.
)I *membername*	This statement includes additional HELP information in the specified member. The include control character,)I, can appear anywhere within a data set member. If the HELP information you plan to add is not available yet, you can specify the control character and later add the information. No error messages are issued. The member name can be up to eight characters long. There must be at least one blank before and after the member name.
))*messageid*	Indicates that information follows describing the named *messageid*. One of these control statements should be present for each message issued by the command. Each statement contains the identifier of the message it describes. Message IDs can be any length and the first character must be alphabetic.
)O	Indicates that the command operands and their descriptions follow. Positional operands must follow immediately after the)O control statement and before the))keyword control statements.
)P	Indicates that a positional operand description follows. One of these control statements is required for each positional operand within the command. Each statement contains the name of the positional operand it describes.

Table 10. Format of a HELP Data Set Member (continued)

Control Character	Purpose of Statement

))*keyword* or keyword with alias	Indicates that information follows describing the named keyword. One of these control statements must be present for each keyword operand within the command. Each statement contains the name of the keyword it describes.
	
	An example of an alias used with a keyword (taken from the help member for the ALLOC command is):
	`))DATASET('DSNAME(S)'/*)`
	
	or
	`DSNAME('DSNAME(S)'/*)`
	
	which indicates the name of the data set to be allocated. List of dsnames (with blanks) specifies the data sets are to be concatenated. The * indicates the terminal to be allocated.

=*subcommandname*	Indicates that information follows concerning the subcommand named after the equal sign. One of these statements is required for each subcommand accepted by the command being described. This statement merely names the subcommand; it does not describe it. Describe the subcommand in the same manner you would describe a command. If the subcommand has an alias name, you may include the alias name on the control statement, in the format `=subcommandname=subcommandalias`. No blanks can appear between the subcommand and the alias.
*	Indicates a comment. The data on this statement is not processed or displayed.

All statements, except the =subcommandname statement, can contain additional information. If you include additional information on the statements, the control characters)S,)F,)X,)I, and)O must be followed by at least one blank, and the control character))keyword must be followed by at least one blank or a left parenthesis. Any information after the membername field on the include,)I, statement is treated as a comment. Use the left parenthesis when the keyword you are describing is followed by operands enclosed in parentheses.

The only restrictions on statements are that columns 72-80 are reserved for sequence numbers, and column one must contain a right parenthesis, an equal sign, an asterisk, or a blank. The sequence numbers are not printed when the HELP command is executed.

The prompt mode HELP function

The prompt mode HELP function provides the user of your command processor with information from the HELP data set when the parse service routine issues prompt mode messages.

If you want the prompt mode HELP function to be available for your command processor and its subcommands, enter the positional parameter control character `)P` on the first line of each positional parameter description for the command and its subcommands in the HELP member.

If you do not provide a description for a positional parameter, supply the name of the positional parameter along with the information you want displayed when the user requests information about the parameter. If a description exists, modify it to not repeat information that is provided by the messages. This also applies to the other descriptions in the HELP members.

Note: If you insert)P for only some of the positional parameters for a command or subcommand, unpredictable results can occur when parse processing issues a HELP command for one of its positional parameters.

For more information, see "Using the prompt mode HELP" on page 23.

An Example of a HELP Member

This topic describes how a fictitious command, SAMPLE, whose syntax is described in Figure 14, is formatted for entry into the HELP data set or your own private HELP data set.

```
SAMPLE      posit1 [,(posit2)][KEYWD1(posit3,posit4)]
```

Figure 14. Syntax of the SAMPLE Command

The SAMPLE command has one subcommand, the EXAMPLE subcommand, whose syntax is shown in Figure 15. Both the command and the subcommand can issue messages IKJXX110I and IKJXX111I.

```
                          [KEYWD10]
EXAMPLE      posit10,posit11 [KEYWD11] [KEYWD13(posit12)]
                          [KEYWD12]
```

Figure 15. Syntax of the EXAMPLE Subcommand

Figure 16 on page 66 shows the statements that present and format information about the SAMPLE command and EXAMPLE subcommand.

```
)S       The SAMPLE command has the following subcommands:  EXAMPLE
)F       Functional description of the  SAMPLE command:
               The SAMPLE command is a fictitious command;
         No command processor exists with this name.
               The SAMPLE command is used merely to describe the
         functions of the HELP data set control statements.
)X       The SAMPLE command has the following syntax:
             Describe the syntax of the SAMPLE command here.
)I MBRNAME   Include additional syntax information for the SAMPLE
              command from the indicated HELP member.
)M       The SAMPLE command issues the following messages:
))IKJXX1101 Describe the message IKJXX1101 here.
))IKJXX1111 Describe the message IKJXX1111 here.
)O       The SAMPLE command has the following positional operands:
)P             POSIT1   Describe it here.
)P             POSIT2   Describe it here.
))KEYWD1 Describe the keyword, KEYWD1 here; include a description of
               POSIT3 and
               POSIT4
=EXAMPLE
)F       Functional description of the EXAMPLE subcommand:
               The EXAMPLE subcommand is a fictitious subcommand.
)X       The EXAMPLE subcommand has the following syntax:
               Describe the syntax of the EXAMPLE subcommand here.
)O       The EXAMPLE subcommand has the following positional operands:
)P             POSIT10   Describe it here.
)P             POSIT11   Describe it here.
))KEYWD10 Describe the keyword, KEYWD10 here.
))KEYWD11 Describe the keyword, KEYWD11 here.
))KEYWD12 Describe the keyword, KEYWD12 here.
))KEYWD13 (POSIT12)
                Describe the keyword, KEYWD13, and the positional
          operand, POSIT12, here.
```

Figure 16. Example of a HELP Member for the SAMPLE Command and EXAMPLE Subcommand

Chapter 10. 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 topic describes the methods that you can use to add your new command processor to TSO/E.

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.

However, if you choose to postpone installing your command processor until you have tested it, you can execute it under the control of the TEST command. To prepare to test your command processor, link-edit the object modules that result from assembling your command processor into a partitioned data set. "Testing a command processor not currently executing" on page 70 describes how to invoke the TEST command for a command processor.

Using a Private Step Library

If you are an unauthorized user, you can request that a LOGON procedure be created that defines a private step library on the STEPLIB DD statement. 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/E users, this method is not recommended because of the TSO/E 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/E 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/E 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/E users.

For information about creating new statements in the link list, see *z/OS MVS Initialization and Tuning Reference*.

Chapter 11. Executing and Testing a command processor

After you have installed your command processor, you are ready to execute it. If you encounter errors or abnormal terminations while an *unauthorized* command processor is executing, use the TSO/E TEST command to help determine the reasons for the failures. Use the TSO/E TESTAUTH command to test *authorized* command processors.

This topic describes how to execute your command processor and how to invoke the TEST and TESTAUTH commands for a command processor. This topic only introduces the TEST and TESTAUTH commands; if you are not familiar with the functions and subcommands that they support, see Chapter 16, "Testing a Program," on page 93.

Executing a command processor

To execute your command processor, enter the command name followed by the operands that are needed for the function you want performed.

For example, suppose you have written a command called CONVERT, which converts data records from an input data set into another format, places the results into an output data set, and optionally prints the converted data. Assume also that you have defined the command syntax as follows:

```
CONVERT   input-dsname  output-dsname   [ PRINT   ]
                                        [ NOPRINT ]
```

That is, input-dsname and output-dsname are positional operands and PRINT/NOPRINT are keyword operands where NOPRINT is the default if neither PRINT nor NOPRINT are specified.

To convert the records in INPUT.DATA, place the result in OUTPUT.DATA and print the contents of the output data set, enter the following command:

```
CONVERT  INPUT.DATA  OUTPUT.DATA  PRINT
```

Testing an Unauthorized command processor

You can use the TSO/E TEST command to test an executing, unauthorized command processor if it abnormally terminates or produces incorrect results. You can also use TEST to step through your command processor to verify that it is executing properly.

Testing a command processor That is Terminating Abnormally

If you are executing your command processor, and it has begun to terminate abnormally, you receive a diagnostic message at the terminal followed by a `READY` message. You then have a choice of terminating your program or testing it. If you issue the TEST command without operands, the TEST command processor receives control and you can use the TEST subcommands to test your program. If your response is anything but TEST, a question mark (?) or TIME, your command processor is abnormally terminated.

Testing a command processor not currently executing

You can also use the TEST command to execute and test a command processor that is not currently executing.

Suppose that the load module for the CONVERT command processor described earlier in this topic resides in the partitioned data set *prefix*.LOAD. To test the command processor, enter:

```
TEST (CONVERT) CP
```

Specify the CP keyword on the TEST command to indicate that the program to be tested is a command processor. The TEST routine creates a command processor parameter list (CPPL), and places its address into register 1 before loading the program. If you do not specify the CP keyword, your command processor will fail.

The TEST command prompts you to enter the command and its operands. You could then enter a command such as:

```
CONVERT  INPUT.DATA  OUTPUT.DATA  PRINT
```

When the TEST command processor issues the TEST mode message to the terminal, you can begin using the TEST subcommands to execute and test your program. For more information on how to use the TEST command, see Chapter 16, “Testing a Program,” on page 93 and Chapter 17, “A Tutorial Using the TEST Command,” on page 113.

Testing an authorized command processor

You can use the TESTAUTH command to execute and test an authorized command processor that is *not* currently executing. Unlike the TEST command, which is used for unauthorized programs, you cannot use the TESTAUTH command to test a currently executing program. However, the TEST and TESTAUTH commands are similar in that they support most of the same operands, subcommands and functions.

Suppose that the load module for an authorized command processor, called AUTHCMD, resides in 'SYS1.LINKLIB'. To test the command processor, enter:

```
TESTAUTH  'SYS1.LINKLIB(AUTHCMD)' CP
```

Specify the CP keyword on the TESTAUTH command to indicate that the authorized program to be tested is a command processor. The TESTAUTH routine creates a command processor parameter list (CPPL), and places its address into register 1 before loading the program. If you do not specify the CP keyword, your command processor will fail.

The TESTAUTH command prompts you to enter the command and its operands. You could then enter a command such as:

```
AUTHCMD  operand1  operand2
```

When the TESTAUTH command processor issues the TESTAUTH mode message to the terminal, you can begin using the TESTAUTH subcommands to execute and test your program. For more information about the functions and subcommands of the TESTAUTH command, see Chapter 16, “Testing a Program,” on page 93.

Part 3. Preparing, Executing and Testing a Program

After you have written a program or source code, the next step is to compile or assemble it. The compiler or assembler creates listings that help you diagnose problems in your source statements, and object modules that contain the compiled or assembled code.

If your program requires data from system libraries or from other programs, you can use TSO/E to link-edit a number of object modules together to form a load module.

You can use TSO/E to execute a program in the foreground, or to submit the JCL statements necessary to execute it in the background. A discussion of how to execute background jobs is provided in *z/OS TSO/E User's Guide*.

While you are executing your program, you may encounter errors or abnormal terminations (abends). To help determine the cause of errors and to monitor the execution of your program, use the TEST command for unauthorized programs. Use the TESTAUTH command for authorized programs.

Part 3, "Preparing, Executing and Testing a Program"of this document contains several chapers that describe how to:

- Compile or assemble source code
- Link object modules together to form load modules
- Load programs into real storage and execute them
- Debug programs and monitor their execution

Chapter 12. Overview of Preparing, Executing and Testing a Program

The following figures show the steps and commands you use when executing a program. These commands are discussed in subsequent topics. Table 11 presents the commands you can use when executing a program and shows what each command accomplishes.

Table 11. Commands Used to Prepare, Execute and Test a Program

Command	Compiles	Link-Edits	Loads	Begins Execution	Monitors Execution
RUN For use ONLY with certain program products.	X	X	X	X	
ASM, FORT, PLI, COBOL Command used to compile or assemble; depends on programming language used.	X				
LINK Produces linkage editor listings to help test and debug a program.		X			
LOADGO		X	X	X	
CALL Program must be in executable load module form and must be a member of a partitioned data set.			X	X	
TEST Program must be a link-edited member of a partitioned data set, or an object module in a sequential or partitioned data set.			X	X	X
TESTAUTH Program must be a link-edited member of an APF-authorized library.			X	X	X

Figure 17 on page 74 shows, in diagram form, the compile, link-edit and execution of a PLI language program. The commands, highlighted in dark print, are discussed in later topics. The modules and listings produced by these commands are shown in the boxes.

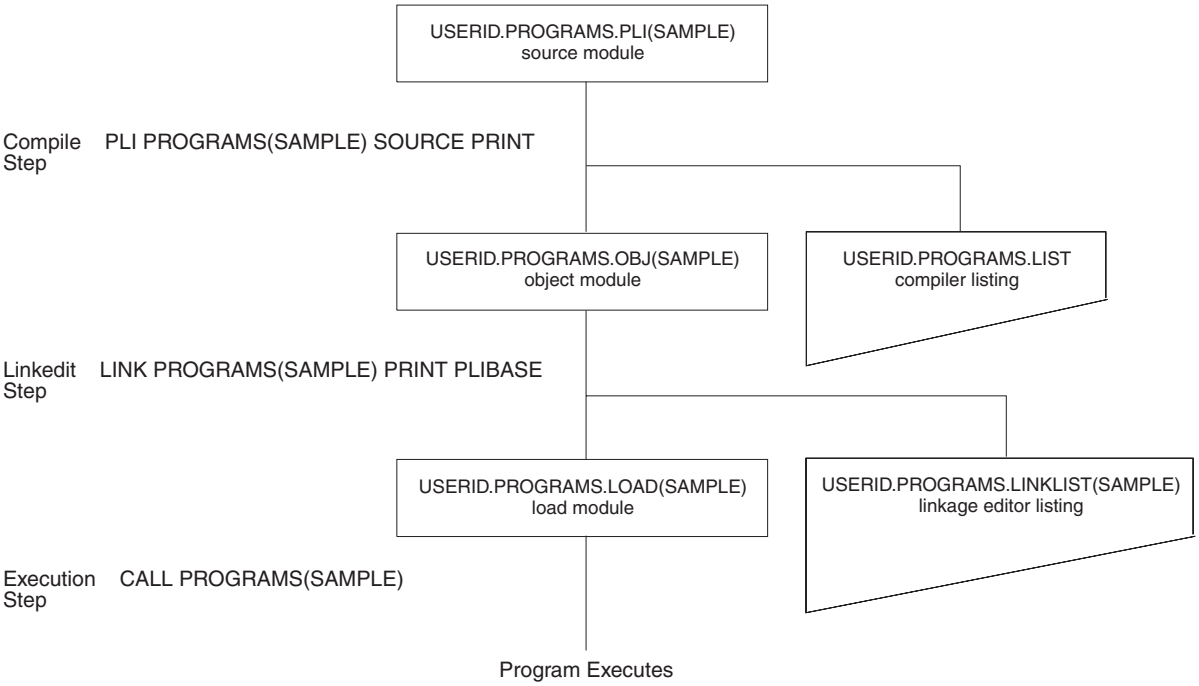


Figure 17. Compiling and Link-Editing a Single Program

Figure 18 on page 75 shows what you would see at your terminal when you execute this program. The commands you enter are in lower case type and the system responses are in upper case type.

```
READY
pli programs(sample) source print
PL/I OPTIMIZER V1 R4.0
OPTIONS SPECIFIED
S;

NO MESSAGES PRODUCED FOR THIS COMPILATION
COMPILE TIME   0.00 MINS     SPILL FILE=      0 RECORDS, SIZE  4051
READY
list programs.list
IKJ52827I PROGRAMS.LIST
PL/I OPTIMIZING COMPILER      VERSION 1  RELEASE 4.0
OPTIONS SPECIFIED
S;
PL/I OPTIMIZING COMPILER      TSOCALL=
                SOURCE LISTING
NUMBER
10000    TSOCALL:
              PROCEDURE OPTIONS(MAIN);
63500    DECLARE (FILEOUT) FILE;/* PLI OUTPUT FILE        */
71000         PUT FILE (FILEOUT) EDIT ('THIS PLI PROGRAM IS EXECUTING')
                                      (A);
90000      END TSOCALL;
PL/I OPTIMIZING COMPILER          TSOCALL:
NO MESSAGES PRODUCED FOR THIS COMPILATION
COMPILE TIME       0.00 MINS      SPILL FILE=      0 RECORDS, SIZE 4051

READY
link programs(sample) print plibase

READY
list programs.linklist(sample)
IKJ52827I  PROGRAMS.LINKLIST(SAMPLE)

       H96-LEVEL LINKAGE EDITOR OPTIONS SPECIFIED TERM
                 DEFAULT OPTION(S) USED _ SIZE=(262144,49152)
       ****SAMPLE   NOW REPLACED IN DATA SET    AMODE 24
       RMODE IS 24
       AUTHORIZATION CODE IS         0.
READY
alloc f(fileout) dsn(*)
READY
call programs(sample)

  THIS PLI PROGRAM IS EXECUTING

READY
```

Figure 18. Terminal Session Showing Execution of a Single Program

Chapter 13. Compiling and Assembling Programs

After you write your source code, you must compile it into object code and place it in an object module (see Figure 17 on page 74). The command you use to compile your source code depends on which programming language you are using. The third, or descriptive qualifier in the source code data set name should identify the programming language used to help the compiler find and process the data.

There are a number of versions of some compilers. You can usually find information on how to use your version of a compiler with the compiler code or in its accompanying reference material. This manual describes the commands used with several common compilers.

The RUN command, which is designed specifically for use with the program products listed in Table 12 on page 78, compiles, loads, and executes source statements. If you cannot use the RUN command, you need the diagnostic information provided by the other commands, or you need to link your program to other modules, you must issue separate commands to link-edit your object modules into load modules, load these modules into main storage, and begin program execution. These commands, listed in Table 11 on page 73, are described in the following topics.

Some commands used to compile or assemble source statements are:
- ASM
- COBOL
- FORT
- PLI

ASM Command

Using the ASM command, you can process assembler language data sets and produce object modules. For more information about the ASM command, see *z/OS TSO/E Command Reference*.

COBOL Command

Using the COBOL command, you can compile American National Standard (ANSI) COBOL programs. This command reads and interprets parameters for the OS Full American National Standard COBOL Version 3 or Version 4 compiler. It also allocates required data sets and passes parameters to the compiler. For more information about the COBOL command, see *z/OS TSO/E Command Reference*.

FORT Command

Using the FORT command, you can compile a FORTRAN IV (G1) program. The FORT command also allocates required data sets and passes parameters to the FORTRAN IV (G1) compiler. For more information about the FORT command, see *z/OS TSO/E Command Reference*.

PLI Command

Using the PLI command, you can call the PL/I Optimizing compiler. For more information about the PLI command, see *z/OS TSO/E Command Reference*.

RUN Command

Use the RUN command to compile, load, and execute the source statements in the data set that you are editing. The RUN command is designed specifically for use with certain program products; it selects and invokes the particular program product needed to process your source statements. Table 12 shows which program product is selected to process each type of source statement.

Table 12. Source/Program Product Relationship

SOURCE	PROGRAM PRODUCT
ASSEMBLER	Assembler F
COBOL	OS/VS COBOL Release 2.4
FORTRAN	FORTRAN IV (G1)
PL/I	PL/I Checkout Compiler or PL/I Optimizing Compiler
VSBASIC	VSBASIC
Note: User-defined data set types can be executed under the RUN subcommand of the EDIT command if a prompter name was defined by the installation. However, the RUN command does not recognize these same data set types.	

Using the RUN command, you can specify:

- Which data set contains the source code or object module you want to process, and which assembler or compiler you want to use to process your source statements.
- A string of up to 100 characters that is to be passed as parameters to the program you are running.
- The libraries that contain subroutines your program will use during its execution.
- The options you want to use with a VSBASIC program.

The RUN subcommand of the EDIT command is very similar to the RUN command, and you may use it similarly to the way you use the RUN command. Before you use the RUN subcommand, keep in mind:

- Any data sets required by your problem program may be allocated before you enter EDIT mode or may be allocated using the ALLOCATE subcommand of EDIT.
- If you wish to enter a value for “parameters”, you should enter this prior to any of the other keyword operands.

Compiling Source Code Statements

If the data set name follows standard naming conventions, and the descriptive qualifier is the name of the programming language used, you do not need to specify a compiler with the RUN command. If the system cannot determine which compiler to use, you will be prompted for more information. You can, however, specify the compiler on the RUN command.

For example, to compile the source code statements in member SAMPLE of data set PROG1.PLI, enter:

```
RUN PROG1.PLI(SAMPLE)
```

Passing Parameters When Compiling

To pass parameters to a program, enclose the parameters in single quotes and specify them after the name of the data set.

For example, to specify 13, FRIDAY, CAT as parameters to pass to program PROG1, which was written in PLI, enter:

```
RUN PROG1 '13 FRIDAY CAT' PLI
```

Specifying a Subroutine Library When Compiling - the LIB Operand

If you use subroutines that reside in a library data set, specify the LIB operand with the name of the library data set enclosed in parentheses.

For example, when running COBOL program PROG1, you need to use library SUBS.LOAD because it contains the subroutines your program calls:

```
RUN PROG1 COBOL LIB(SUBS.LOAD)
```

You can use the LIB operand when using the assembler, FORTRAN, COBOL, and PLI compilers.

Specifying VSBASIC Compiler Options

When running the VSBASIC compiler, you can use a number of options with the RUN command. Specify the options you want after the data set's descriptive qualifier (in this case, VSBASIC). Following is a list of operands and a description of each:

LPREC
specifies that long precision arithmetic calculations are required by the VSBASIC program.

SPREC
Default specifying that short precision arithmetic calculations are adequate for a VSBASIC program.

TEST
specifies that testing of a VSBASIC program is to be performed.

NOTEST
Default specifying that the TEST function is not desired with a VSBASIC program.

STORE
specifies that the VSBASIC compiler is to store an object program.

NOSTORE
Default specifying that the VSBASIC compiler is not to store an object program.

GO Default specifying that the VSBASIC program is to receive control after compilation.

NOGO
specifies that the VSBASIC program is not to receive control after compilation.

SIZE(value)
specifies the number of 1000-byte blocks of VSBASIC user area where value is an integer of one to three digits.

PAUSE
specifies that the VSBASIC compiler is to prompt the terminal user between program chains, giving the user the chance to change certain compiler options.

NOPAUSE
Default specifying no prompting between program chains.

SOURCE
Default specifying that new VSBASIC source code is to be compiled.

OBJECT
specifies that the data set name entered is a fully-qualified name of an object data set to be executed by the VSBASIC compiler.

For more information about the RUN command, see *z/OS TSO/E Command Reference*.

Chapter 14. Binding or Link-Editing a Program

The binder and linkage editor, invoked by the LINK command, provides a great deal of information to help you test and debug a program. This information includes a cross-reference table and a map of the module that identifies the location of control sections, entry points, and addresses. You can specify which of these types of information you want by including the LIST, MAP, and XREF operands with the LINK command. You can have this information listed at your terminal or saved in a data set.

LINK Command

Use the LINK command to invoke the binder or linkage editor service program. Basically, the binder or linkage editor converts one or more object modules (the output modules from a compiler or the assembler) into a load module or program object that is suitable for execution.

You can specify all the binder or linkage editor options explicitly or you can accept the default values. The default values are satisfactory for most uses. By accepting the default values, you simplify the use of the LINK command. This topic does not describe every operand available with the LINK command. For more information on the LINK command, see *z/OS TSO/E Command Reference*.

In some cases, you might want to use the LOADGO command as an alternative to the LINK command. The LOADGO command may better serve your needs if one of the following situations exists:

- The module that you want to process has a simple structure. That is, it is self contained and does not pass control to other modules.
- You do not require the extensive listings produced by the binder or linkage editor.
- You do not want a program object or load module saved in a library.

On the LINK command, you can specify:

- The names of one or more data sets containing your object modules and/or binder or linkage editor control statements.
- The name of the partitioned data set that will contain the program object or load module after processing by the binder or linkage editor.
- One or more names of library data sets the binder or linkage editor is to search to locate object modules referred to by the module being processed; that is, to resolve external references.
- Whether you want the system to produce binder or linkage editor listings and place them in a data set you specify or at your terminal.
- Whether you want the system to produce and include in the PRINT data set:
  - A map of the output module consisting of the control sections, the entry names, and for overlay structures, the segment number.
  - A list of all binder or linkage editor control statements.
  - A cross-reference table.
- Whether you want the system to place in the output module the symbol tables created by the assembler and contained in the input modules.

- Whether you want error messages directed to your terminal as well as to the PRINT data set.

Creating a Program Object or Load Module

To create a program object or load module by binding or link-editing an object module, issue the LINK command and specify the object module name. If you want to bind or link more than one object module, issue the LINK command and list the object modules' names separated by commas and enclosed in parentheses.

For example, to bind or link-edit sequential data set object modules FIRST.OBJ, SECOND.OBJ and FIFTH.OBJ, enter:

```
LINK (FIRST,SECOND,FIFTH)
```

The specified data sets will be concatenated within the output program object or load module in the sequence that they are included in the list of data set names. If there is only a single name in the data set list, parentheses are not required around the name. To bind or link-edit object modules in members, enclose the member names in parentheses.

For example, to bind or link members ONE, TWO, and THREE of data set D00ABC.SAMPLE.OBJ, enter:

```
LINK (SAMPLE(ONE),SAMPLE(TWO),SAMPLE(THREE))
```

If the data set name is D00ABC.OBJ, enclose the member names within two pairs of parentheses, for example:

```
LINK ((ONE),(TWO),(THREE))
```

You may substitute an asterisk (*) for a data set name to indicate that you will enter binder or linkage editor control statements from your terminal. The system prompts you to enter the control statements, which should begin in column 2. Press the Enter key after you enter each control statement. Enter a null line to indicate the end of your control statements.

When you bind or link-edit an object module that is a sequential data set, the binder or linkage editor generates a load module in a partitioned data set and assigns the program object or load module the member name TEMPNAME.

To store the output from the LINK command in a data set, use the LOAD operand with the user-specified qualifier of the data set enclosed in parentheses.

For example, to store the results of the previous example in data set SALES.LOAD(ONE), enter:

```
LINK (FIRST,SECOND,FIFTH) LOAD(SALES(ONE))
```

If you omit the LOAD operand, the system generates a name according to the data set naming conventions. The default descriptive qualifier for the data set name is LOAD.

Resolving External References - the LIB Operand

To have the system search specific library data sets to resolve any external references in your program, specify the LIB operand, followed by the name of the library data set(s) enclosed in parentheses.

When you specify more than one name, separate the names by a valid delimiter.

For example, to have the system search library data sets MYLIB, YOURLIB, and OURLIB to resolve any external references, enter:

```
LINK SECOND LIB(MYLIB,YOURLIB,OURLIB)
```

If you specify more than one name, the data sets are concatenated to the file name of the first data set in the list. For control statements, the first data set in the list must be preallocated with the ddname or file name SYSLIB before issuing the LINK command. If you specify more than one name, the data sets concatenated to the file name of the first data set lose their individual identity. For details on dynamic concatenation, see *z/OS MVS Programming: Authorized Assembler Services Guide*.

Producing Output Listings - the PRINT Operand

To have the system produce binder or linkage editor output listings and place them into a data set, use the PRINT operand with the name of the data set enclosed in parentheses.

For example, to have the system produce binder or linkage editor listings from object modules FIRST.OBJ, SECOND.OBJ, and FIFTH.OBJ and place them in data set PRINT.DATA, enter:

```
LINK (FIRST,SECOND,FIFTH) PRINT(PRINT.DATA)
```

To have the system produce binder or linkage editor output listings and display them at your terminal, use the PRINT operand with an asterisk enclosed in parentheses.

For example, to have the system produce binder or linkage editor listings from object modules FIRST.OBJ, SECOND.OBJ, and FIFTH.OBJ and display them at your terminal, enter:

```
LINK (FIRST,SECOND,FIFTH) PRINT(*)
```

When you omit the data set name on the PRINT operand, the data set that is generated is named according to the data set naming conventions. The default descriptive qualifier for the data set name is LINKLIST. This is the default value if you specify the LIST, MAP, or XREF operand. If you want to have the listings displayed at your terminal, you may substitute an asterisk (*) for the data set name.

To have the system produce no binder or linkage editor output listings, use the NOPRINT operand.

For example, to have the system not produce binder or linkage editor listings from object modules FIRST.OBJ, SECOND.OBJ, and FIFTH.OBJ, enter:

```
LINK (FIRST,SECOND,FIFTH) NOPRINT
```

NOPRINT causes the MAP, XREF, and LIST options to become not valid. NOPRINT is the default value if both PRINT and NOPRINT are omitted, and you do not use the LIST, MAP, or XREF operand.

Creating a Map of the Program Object or Load Module - the MAP Operand

To have the system include a map of the program object or output module use the MAP and PRINT operands. The program object or output module consists of the control sections, the entry names, and (for overlay structures) the segment number

in the PRINT data set. NOMAP, specifying that you do not want a map of the program object or output module, is the default.

For example, to have the system include a map of the output module from object module SECOND into PRINT data set PRINT.DATA, enter:

```
LINK SECOND PRINT(PRINT.DATA) MAP
```

Producing a List of All Binder or Linkage Editor Control Statements - the LIST Operand

To have the system include a list of all binder or linkage editor control statements and place them in the PRINT data set, use the PRINT and LIST operands. NOLIST, which specifies that you do not want a list of all binder or linkage editor control statements, is the default.

For example, to produce a list of all the binder or linkage editor control statements from object module SECOND.OBJ, and place them in PRINT data set PRINT.DATA, enter:

```
LINK SECOND PRINT(PRINT.DATA) LIST
```

To produce a list of all the binder or linkage editor control statements from object module SECOND.OBJ, and place them in data set SECOND.LINKLIST, where SECOND.OBJ contains both binder or linkage editor control statements and the object module, enter:

```
LINK SECOND LIST
```

Producing a Cross Reference Table - the XREF Operand

To have the system create a cross reference table and place it in the PRINT data set, use the PRINT and XREF operands. The cross reference table includes the module map and a list of all address constants referring to other control sections. Because the XREF operand includes a module map, both XREF and MAP cannot be specified for a particular LINK command. NOXREF, which specifies that you do not want a cross reference table in the PRINT data set, is the default.

For example, to produce a cross reference table from object module SECOND.OBJ, and place it in PRINT data set PRINT.DATA, enter:

```
LINK SECOND PRINT(PRINT.DATA) XREF
```

Producing a Symbol Table - the TEST Operand

To have the system place a symbol table created by the assembler and contained in the input modules into the output module, use the TEST operand. NOTEST, specifying that you do not want a symbol table, is the default.

For example, to have the system place a symbol table created by the assembler and contained in the input module SECOND into the output module, enter:

```
LINK SECOND TEST
```

Sending Error Messages to Your Terminal - the TERM/NOTERM Operand

You can choose whether you want error messages directed to your terminal as well as to the PRINT data set. TERM, which specifies that the system direct error messages to your terminal as well as to the PRINT data set, is the default. If you do not want error messages directed to your terminal, use the NOTERM operand.

For example, enter:

```
LINK SECOND NOTERM
```

Chapter 15. Loading and Executing a Program

Before running a program, you must place the program into main storage. Placing a program into main storage is called *loading* the program. To load and execute a program, use either the LOADGO or CALL command. The LOADGO command loads object modules produced by a compiler or assembler, and program objects and load modules produced by the binder or linkage editor. If you want to load and execute a single program object or load module, the CALL command is more efficient.

LOADGO Command

Use the LOADGO command to load a compiled or assembled program into main storage and begin execution. The LOADGO command invokes the system loader to accomplish this function. The loader combines basic editing and loading services of the binder or linkage editor and program fetch in one job step. Therefore, the *load* function is equivalent to the *link-edit and go* function.

The LOADGO command does not produce objects or load modules for program libraries, and it does not process binder or linkage editor control statements such as INCLUDE, NAME, or OVERLAY. If you need to use these control statements, use the LINK and CALL commands.

The LOADGO command also searches a specified call library (SYSLIB) or a resident link pack area, or both, to resolve external references.

Using the LOADGO command, you can specify:

- The names of one or more data sets containing your program objects or object modules and/or load modules.
- Parameters to pass to the program you execute.
- Whether the system is to produce output listings and place them in a data set you specify or display them at your terminal.
- One or more names of library data sets to be searched by the binder or linkage editor to locate object modules referred to by the module being processed; that is, to resolve external references.
- Whether the system is to send error messages to your terminal as well as to the PRINT data set.
- Whether the system is to include a list of external names and their addresses in the PRINT data set.
- Whether the system is to search the data set(s) you specified on the LIB operand to locate program objects or load modules to which the executing load module refers.
- The external name for the loaded program's entry point.
- The name you want to assign to the loaded program.

This topic shows you how to use the basic functions of LOADGO. For a complete description of the LOADGO command, see *z/OS TSO/E Command Reference*.

Loading and Executing Programs with No Operands

To load and execute the source code in a data set or a group of data sets, issue the LOADGO command and specify the data set name. The rightmost qualifier of the data set name must be OBJ or LOAD. When loading and executing the code in a single data set, specify the data set name without parentheses.

For example, to load and execute the code in data set PAYROLL.LOAD, enter:

```
LOADGO PAYROLL.LOAD
```

When loading and executing the code in a group of data sets, specify the data set names in parentheses, separating each name with a comma. The names may be data set names, names of members of partitioned data sets, or both.

For example, to load and execute the code in data sets FIRST.OBJ, SECOND.OBJ and THIRD.LOAD, enter:

```
LOADGO (FIRST.OBJ,SECOND.OBJ,FIFTH.LOAD)
```

Passing Parameters when Loading and Executing Programs

To pass parameters to a program when loading and executing it, specify the parameter enclosed in single quotes following the data set name field.

For example, to pass THE RAIN IN SPAIN as a parameter to the program in data set FIFTH.LOAD, enter:

```
LOADGO FIFTH.LOAD 'THE RAIN IN SPAIN'
```

Requesting Output Listings when Loading and Executing Programs - the PRINT/NOPRINT and TERM/NOTERM Operands

To produce output listings and place them in a data set, use the PRINT operand with the data set name enclosed in parentheses. The NOPRINT operand, suppressing output listings, is the default. Note that the NOPRINT and MAP operands are mutually exclusive. The MAP operand, discussed below, puts data in the PRINT data set. Therefore, if you want the MAP information, you must also specify the PRINT operand with the LOADGO command.

For example, to send the output listings from the program in data set FIFTH.LOAD to data set OUT5.DATA, enter:

```
LOADGO FIFTH.LOAD PRINT(OUT5.DATA)
```

To produce output listings and send the output to your terminal, use the PRINT operand with an asterisk enclosed in parentheses.

For example, to send the output listings from the program in data set FIFTH.LOAD to your terminal, enter:

```
LOADGO FIFTH.LOAD PRINT(*)
```

All error messages are directed to your terminal as well as to the PRINT data set, as the TERM operand is a default with the LOADGO command. To direct all error messages only to the PRINT data set, not to your terminal, use the NOTERM operand.

For example, to direct all error messages only to the print data set OUT5.DATA, enter:

```
LOADGO FIFTH.LOAD PRINT(OUT5.DATA) NOTERM
```

Resolving External References when Loading and Executing Programs - the CALL/NOCALL and LIB Operands

To resolve external references, you must know in which data sets the code or data being referred to is kept. If you know the program object or load module library data set(s) name, use the LIB operand with the name enclosed in parentheses to tell the system where to look to resolve any external references.

For example, to search library data sets MYLIB, YOURLIB, and OURLIB, to resolve any external references when executing the program in FIFTH.LOAD, enter:

```
LOADGO FIFTH.LOAD LIB(MYLIB,YOURLIB,OURLIB)
```

The CALL operand is a default with the LOADGO command. The system will search for the data set(s) you specified on the LIB operand to locate the program objects or load modules to which the executing code refers.

Use the NOCALL operand following the LIB operand to prevent the system from searching the data set(s) you specified on the LIB operand.

For example, to suppress searching for program objects or load modules within data set MYLIB, YOURLIB, or OURLIB when executing the program in FIFTH.LOAD, enter:

```
LOADGO FIFTH.LOAD LIB(MYLIB,YOURLIB,OURLIB) NOCALL
```

To include a list of external names and their addresses in the PRINT data set, use the PRINT, LIB and MAP operands. The NOMAP operand is the default and does not include the MAP information in the PRINT data set.

For example, to resolve external references found in data set PROLIB, and list them in data set OUT5.DATA, enter:

```
LOADGO FIFTH.LOAD PRINT(OUT5.DATA) LIB(PROLIB) MAP
```

Note that MAP and NOPRINT are mutually exclusive operands.

Specifying an Entry Point when Loading and Executing Programs - the EP Operand

To specify an external name for a program's entry point when loading and executing the program, use the EP operand with the entry point name enclosed in parentheses.

For example, to specify START as the external name for the entry point into the program in data set FIFTH.LOAD, enter:

```
LOADGO FIFTH.LOAD EP(START)
```

If the entry point of the loaded program is a program object or load module, you must specify this operand.

Specifying Names when Loading and Executing Programs - the NAME Operand

To assign a name to a program in a data set, use the NAME operand followed by the name of the program.

For example, to assign the name PROG3 to the program in FIFTH.LOAD, enter:

```
LOADGO FIFTH.LOAD NAME(PROG3)
```

CALL Command

Use the CALL command to load and execute a program that exists in executable (program object or load module) form. The program may be user-written or it may be owned by the system, for example, a compiler, sort, or utility program.

You can specify the name of the program (program object or load module) to be processed. It must be a member of a partitioned data set. If you do not specify a member name, member TEMPNAME is assumed. Also, you can pass parameters to the program.

This topic shows you how to use the basic functions of the CALL command. For a complete description of the CALL command, see *z/OS TSO/E Command Reference*.

Loading and Executing Load Modules

To load and execute a load module in a member of a partitioned data set, specify the name of the data set with the name of the member enclosed in parentheses.

For example, to load and execute the load module in member JOYCE of data set PUBS.LOAD, enter:

```
CALL PUBS(JOYCE)
```

If the partitioned data set does not conform to data set naming conventions, then you must specify the member name that contains the program you want to execute. If you specify a fully-qualified data set name, enclose it in single quotation marks in the following manner:

```
CALL 'D00ABC1.MYPROG.LOADMOD(DISCHARG)'
```

or

```
CALL 'SYS1.LINKLIB(IEUASM)'
```

If you do not enclose your data set name in quotes, a high-level *userid* is prefixed to the data set name and the low-level qualifier, LOAD, is attached as well.

For example, if USER1 accesses `PUBS` with the following:

```
CALL PUBS
```

the actual program accessed is in member TEMPNAME in `USER1.PUBS.LOAD`.

Passing Parameters when Loading and Executing Load Modules

To load and execute the load module in a partitioned data set and pass it parameters, specify the parameters enclosed in single quotation marks following the data set name.

For example, to pass PRANCE as a parameter to the load module in member UDOIT of data set DLW.LOAD, enter:

```
CALL DLW(UDOIT) 'PRANCE'
```

In the previous example, the CALL command translates the parameter list to uppercase characters. The ASIS operand of the CALL command prevents translation to uppercase. Use the ASIS operand to pass lowercase data to programs that accept lowercase characters in the parameter list.

For example, to pass LeaveAsis as a parameter to the load module in member NOTRANS of data set DLW.LOAD, enter:

```
CALL DLW(NOTRANS) 'LeaveAsis' ASIS
```

Chapter 16. Testing a Program

This topic introduces the TSO/E TEST and TESTAUTH commands and describes how to test a program. It also discusses the terminology and concepts that you must understand to use the TEST and TESTAUTH commands. The tutorial topic gives a step-by-step explanation of how to use TEST and shows how TEST can help you determine the cause of a programming error. Because TEST and TESTAUTH support the same subcommands, the tutorial can help you learn about using TESTAUTH.

The TEST and TESTAUTH Commands

The TEST and TESTAUTH commands permit you to test an assembler language program, including a command processor or application program, at your terminal. The TEST and TESTAUTH commands also allow you to test APPC/MVS transaction programs. Use the TEST command to test *unauthorized* programs; use the TESTAUTH command to test *authorized* programs.

The TEST Command

While you are executing an unauthorized program that you have written, you may encounter errors or abnormal terminations (abends). You can use the TEST command to help determine the cause of errors in a program that is currently executing. You can also use the TEST command to load and execute a program, and monitor the program's execution.

The TESTAUTH Command

You can use the TESTAUTH command to test an authorized program that is *not* currently executing. Unlike the TEST command, you cannot use TESTAUTH to test a currently executing program. However, the TESTAUTH command supports most of the same operands, subcommands and functions as the TEST command.

Using TEST or TESTAUTH

Test a program by issuing either the TEST or TESTAUTH command. The TEST command issues the `TEST` mode message to let you know that the system is waiting for you to enter a subcommand. Similarly, the TESTAUTH command issues the TESTAUTH mode message. Then use the various subcommands to perform the following basic functions:

- Supply test data that you want to pass to the program.
- Execute the program from its starting address or from any address within the program.
- Step though sections of the program, checking each instruction for proper execution.
- Display selected areas of the program as they currently appear in virtual storage, or display the contents of any of the registers.
- Interrupt the program at specified locations. After you have interrupted the program, you can display areas of the program or any of the registers, or you can issue other subcommands to be executed before returning control to the program being tested. A location in a program where you interrupt execution is called a *breakpoint*. Breakpoints that are specified for programs that are not yet in

virtual storage are called *deferred breakpoints*. You can establish deferred breakpoints for programs that will be brought into virtual storage during execution of the program being tested.

The following restrictions apply when specifying breakpoints:

- Do not insert breakpoints into the TSO/E service routines or into any of the TEST or TESTAUTH load modules.
- The TESTAUTH command does not support breakpoints in storage that has a protection key other than 8.
- When running in supervisor state or in a PSW protection key other than 8, the TEST command does not honor breakpoints in any section of your program.

- Change the contents of specified program locations in virtual storage or the contents of specific registers. You do this with the assignment function.

  Note: The TESTAUTH command does not allow you to modify the contents of storage that has a protection key other than 8.

In addition to these basic debugging functions, the TEST and TESTAUTH command processors provide other functions, such as listing data extent blocks (DEBs), data control blocks (DCBs), task control blocks (TCBs), program status words (PSWs), and providing a virtual storage map of the program being tested.

The discussion of the TEST and TESTAUTH commands in this book shows you how to use these basic functions. For a complete description of the syntax and functions of the subcommands, see *z/OS TSO/E Command Reference*. For more information on the TESTAUTH command, see *z/OS TSO/E System Programming Command Reference*.

When to Use the TEST and TESTAUTH Commands

You can use the TEST command to:

- Test a currently executing, unauthorized program.
- Test an unauthorized program not currently executing.

You might want to test an executing program because it terminated abnormally or because you want to check the current environment to see that the program is executing properly.

You can use the TESTAUTH command to test an authorized program that is *not* currently executing.

Testing a Currently Executing Program

If an unauthorized program terminates abnormally, you receive a diagnostic message from the terminal monitor program (TMP) followed by a READY message. If you respond to the diagnostic message with anything other than TEST, a question mark (?), or TIME, the TMP terminates your program. If you issue the TEST command without a program name, the currently active program remains in storage when the TEST command processor gets control, and you can use the TEST subcommands to debug the defective program.

You can enter both the ? and the TIME command before issuing the TEST command to debug an abnormally terminating program. If you want a dump,

enter a null line instead of issuing the TEST command. If a SYSABEND, SYSMDUMP, or SYSUDUMP file has already been allocated, the null line results in a dump being printed.

If you want to examine the current environment of an executing program that is not terminating abnormally, enter a single attention interruption. The currently active program remains attached and the TMP responds to your interruption by issuing a READY message. When you issue the TEST command without a program name, the currently active program remains in storage under the control of the TEST command processor. You can then use the TEST subcommands to examine the current environment.

In the case of either an abend or an attention interruption, you should *not* enter a program name following the TEST command. If you do, you lose the current in-storage copy of the program because TEST loads a copy of the program you specified instead. Even if you specify the name of the currently active program, a new copy is loaded and the current one is lost.

Testing a program not currently executing

To test a program not currently executing, specify on the TEST or TESTAUTH command the data set name containing the program to be executed and any other applicable operands.

You can load and execute a program under the control of the TEST command processor if it is either:

- A link-edited member of a partitioned data set or extended partitioned data set (PDSE).
- An object module in a sequential or partitioned data set or PDSE.

You can load and execute a program under the control of the TESTAUTH command processor if it is a link-edited member of an APF-authorized library.

Issue the TEST or TESTAUTH command followed by the program name and the operands of the command that either define the program or are necessary to its operation. For example:

- For the TEST command, the keyword LOAD or OBJECT depending on whether the program is a load or an object module. The TESTAUTH command does not support the OBJECT keyword. LOAD is the default.
- The keyword CP, TP, or NOCP, depending on whether the program to be tested is a command processor, an APPC/MVS transaction program, or is another type of program that is not a command processor or a transaction program. NOCP is the default.

  Note: You only need to specify the TP keyword if you are testing an inbound APPC/MVS transaction program (see “Testing an APPC/MVS Transaction Program” on page 96). You can test outbound transaction programs as ordinary programs.
- Additional parameters necessary for the program being tested.

If the program you are testing is a command processor, specify the keyword CP. The CP keyword causes the test routine to create a command processor parameter list (CPPL), and place its address into register 1 before loading the program. If you do not explicitly specify the CP operand, any parameters that you specify in the TEST or TESTAUTH command are passed to the named program as a standard

operating system parameter list. That is, when the program under TEST or TESTAUTH receives control, register 1 contains a pointer to a list of addresses that point to the parameters.

To test an inbound APPC/MVS transaction program, specify the TP operand.

- Use the LU or BASELU keyword to specify the LU on which to test the transaction program. These operands are valid only when you use the TP keyword operand. BASELU is the default.
- Use the keyword KEEPTP to specify that TEST or TESTAUTH should not clean up the transaction program and its conversations when TEST ends. If you do not specify this keyword, the TSO/E TEST command cleans up the transaction program and its conversations.

Testing an APPC/MVS Transaction Program

The following example outlines the required steps for testing an APPC/MVS transaction program with the TSO/E TEST command:

```
   READY

(1) test (myprog) tp('MAIL') keeptp

(2) IKJ57522I YOU CAN ALLOCATE THE TP NOW

   +++ the user starts a program that allocates the TP to be tested +++

(3) TEST

   +++ the user can now test the TP as if it is an ordinary program +++

(4) TEST

(5) end

(6) READY
```

1. *Prefix*.LOAD(myprog) contains the load module for the transaction program to be tested. MAIL is the transaction program name under which the load module is to be tested. The inbound allocate request will try to allocate MAIL.
2. Wait for TSO/E TEST to prompt you to allocate the transaction program to be tested. The message IKJ57522I indicates that it is your turn to start a program that allocates an APPC/MVS conversation with the transaction program to be tested.
3. TEST displays a `TEST` mode message, indicating that it is your turn to enter TEST subcommands to control TEST's processing.
4. TEST returns control to the terminal with another mode message.
5. To terminate TEST processing, use the END subcommand.
6. TEST returns to READY mode.

Because the LU keyword is not specified in this example, TSO/E TEST uses the base LU for testing (BASELU is the default keyword). Also, the transaction program and its remaining conversations are not cleaned up by the TEST command because the KEEPTP keyword is specified.

Examples of Issuing the TEST and TESTAUTH Commands

The following examples show you how to invoke a program or a command processor for testing:

Example 1

Operation: Enter TEST mode after experiencing either an abnormal termination of an unauthorized program or an attention interruption.

Known:
- Either you have received a message saying that your foreground program has terminated abnormally, or you have pressed the attention interrupt key while your program was executing. In either case, you would like to begin debugging your program.

```
test
```

Example 2

Operation: Invoke an unauthorized program for testing.

Known:
- The name of the data set that contains the program: TLC55.PAYER.LOAD(THRUST)
- The program is a load module and is not a command processor.
- The prefix in the user's profile is TLC55.
- The parameters to be passed: 2048, 80

```
test payer(thrust) '2048,80'
```

or

```
test payer.load(thrust) '2048,80'
```

Example 3

Operation: Invoke an unauthorized program for testing.

Known:
- The name of the data set that contains the program: TLC55.PAYLOAD.OBJ
- The prefix in the user's profile is TLC55.
- The program is an object module and is not a command processor.

```
test payload object
```

Example 4

Operation: Test an unauthorized command processor.

Known:
- The name of the data set containing the command processor: TLC55.CMDS.LOAD(OUTPUT)

```
test cmds(output) cp
```

or

```
test cmds.load(output) cp
```

Note: You will be prompted to enter a command for the command processor.

Example 5

Operation: Invoke an unauthorized command processor for testing.

Known:

- The name of the data set containing the command processor is TLC55.LOAD(OUTPUT).
- The prefix in the user's profile is TLC55.

```
test (output) cp
```

Example 6

Operation: Invoke an authorized program for testing.

Known:
- The name of the data set containing the program: 'SYS1.LINKLIB(AUTHPGM)'
- The program is not a command processor.

```
testauth  'sys1.linklib(authpgm)'
```

Example 7

Operation: Test an authorized command processor.

Known:
- The name of the data set containing the command processor: 'SYS1.LINKLIB(AUTHCMD)'

```
testauth  'sys1.linklib(authcmd)'  cp
```

Example 8

Operation: Test an unauthorized APPC/MVS transaction program.

Known:
- The name of the data set containing the transaction program. USER.APPC.LOAD(TESTTP)

```
test 'user.appc.load(testtp)' tp('TESTTP')
```

Example 9

Operation: Test an unauthorized APPC/MVS transaction program and do not clean up the transaction program and its conversations when TEST ends.

Known:
- The name of the data set containing the transaction program. USER.APPC.LOAD(TESTTP)

```
test 'user.appc.load(testtp)' tp('TESTTP') keeptp
```

Example 10

Operation: Test an authorized APPC/MVS transaction program and specify which LU is to be used.

Known:
- The name of the data set containing the transaction program. USER.APPC.LOAD(TESTTP)
- VTAMNODE.LU1 is the LU on which the transaction program is to be tested.

```
testauth 'user.appc.load(testtp)' tp('TESTTP') lu('VTAMNODE.LU1')
```

TEST and TESTAUTH Subcommands

The TEST command issues the TEST mode message to let you know that the system is waiting for you to enter a subcommand. Similarly, the TESTAUTH command issues the TESTAUTH mode message. The subcommands of the TEST and TESTAUTH commands are shown in Table 13. The tutorial in the next topic shows you how to use many of these subcommands to help determine the cause of errors and to monitor the execution of your program. For a complete description of the syntax and function of each subcommand, and for a list of the TSO/E commands that you can use under TEST and TESTAUTH, see *z/OS TSO/E Command Reference*.

Table 13. The TEST and TESTAUTH Subcommands

Subcommand	Function
AND	Performs a logical AND operation on data in two locations, placing the results in the second location specified.
ASSIGNMENT OF VALUES(=)	Modifies values in virtual storage and in registers.
AT	Establishes breakpoints at specified locations.
CALL	Initializes registers and initiates processing of the program at a specified address using the standard subroutine linkage.
COPY	Moves data.
DELETE	Deletes a load module from virtual storage.
DROP	Removes symbols established by the EQUATE command from the symbol table of the module being tested.
END	Terminates all operations of the TEST or TESTAUTH command and the program being tested.
EQUATE	Adds a symbol to the symbol table and assigns attributes and a location to that symbol.
FREEMAIN	Frees a specified number of bytes of virtual storage.
GETMAIN	Acquires a specified number of bytes of virtual storage for use by the program being processed.
GO	Restarts the program at the point of interruption or at a specified address.
HELP	Lists the subcommands of TEST and TESTAUTH and explains their function, syntax, and operands.
LIST	Displays the contents of a virtual storage area or registers.
LISTDCB	Lists the contents of a data control block (DCB). You must specify the address of the DCB.
LISTDEB	Lists the contents of a data extent block (DEB). You must specify the address of the DEB.
LISTMAP	Displays a map of the user's virtual storage.
LISTPSW	Displays a program status word (PSW).
LISTTCB	Lists the contents of the current task control block (TCB). You can specify the address of another TCB.
LISTVP	Lists the vector section size and the partial sum number.
LISTVSR	Displays the vector status register (VSR™).
LOAD	Loads a program into virtual storage for execution. An authorized program loaded by the TESTAUTH command must reside in an APF-authorized library.

Table 13. The TEST and TESTAUTH Subcommands (continued)

Subcommand	Function
OFF	Removes breakpoints.
OR	Performs a logical OR operation on data in two locations, placing the results in the second location specified.
QUALIFY	Establishes the starting or base location for resolving symbolic or relative addresses; resolves identical external symbols within a load module.
RUN	Terminates TEST or TESTAUTH and completes execution of the program.
SETVSR	Changes the vector mask register control mode. Updates the vector count (VCT), the vector interruption index (VIX), and the vector in-use bits (VIU).
WHERE	Displays the virtual address of a symbol or entry point, or the address of the next executable instruction. WHERE can also be used to display the module and CSECT name and the displacement into the CSECT corresponding to an address.

Addressing Conventions Associated with TEST and TESTAUTH

Many of the tasks you can perform with TEST and TESTAUTH involve using subcommands that require you to specify an address. For example, to display the contents of a storage location, you must indicate the address of the area to be displayed. An address used as an operand for a subcommand must be one of the following types:

- Absolute address
- Relative address
- Symbolic address
- Module name and entry name (separated by a period)
- Qualified address
- General register
- Floating-point register
- Vector register
- Vector mask register
- Access register
- Indirect address
- Address expression

These address types are described in the following topics.

Absolute Address

A virtual storage address. An absolute address is 1 to 8 hexadecimal digits followed by a period and not exceeding X'7FFFFFFF'. For example,

```
BC3D60.
```

is an absolute address.

Relative Address

A one-to eight-digit hexadecimal number preceded by a plus sign (+). A relative address specifies an offset from the currently qualified virtual storage address. For example,

```
+A0
```

is a relative address. See the discussion on qualified addresses that follows.

Symbolic Address

One to eight alphameric characters, the first of which is an alphabetic character. A symbolic address corresponds to a symbol in a program or a symbol defined by the EQUATE subcommand. Qualified symbolic addressing is discussed below. For a detailed description on the use of symbols, see "Restrictions on the Use of Symbols" on page 105.

[Module-Name].Entry-Name

A name within a module capable of being externally referenced, preceded by a period (.), and optionally preceded by a name by which the module is known. An entry name is the symbolic address of an entry point into the module; for example, a CSECT name. A module name can be the name or alias of a load module or the name of an object module. Module or entry names can be any combination of up to eight alphameric characters, the first of which is alphabetic or national.

Qualified Addresses

You can qualify symbolic or relative addresses to indicate they apply to a particular module and CSECT. To do this, you must precede the address by the name of the load or object module and the name of the CSECT. The qualified address must be in the form:

```
modulename.csect.address
```

If the address is to apply to the current module, you only need to specify the CSECT name in the following form:

```
csect.address
```

If the address is to apply to the current CSECT within the current module, only the address is necessary; you do not need to qualify the address. The current module and CSECT is initially set to the program being tested. This setting is automatically changed each time a module under a different request block is invoked. This is referred to as *automatic qualification*. Automatic qualification occurs when a module is invoked by ATTACH, XCTL, SYNCH, or LINK. It does *not* occur when a module is loaded, called, or branched to.

The module or CSECT used in determining a base location for resolving symbolic and relative addresses can also be changed by using the QUALIFY subcommand.

For example, if the name of the module is OUTPUT, the CSECT is TAXES, and the symbolic address is YEAR77, you would specify either:

```
output.taxes.year77
```

or

```
.taxes.year77
```

If the current module is OUTPUT. You would specify:

```
year77
```

If the module name and CSECT name are the same as above and the address to be qualified is the relative address +4A, you would specify:

```
output.taxes.+4A
```

General Registers

You can refer to a general register using the AND, OR, assignment-of-value, COPY, or LIST subcommands by specifying a decimal integer followed by an R. The decimal integer indicates the number of the register and must be in the range 0 through 15. Other references to the general registers imply indirect addressing.

If your program issues the STIMER macro or involves asynchronous interruptions, the contents of your registers may be changed by interruptions even though you are in *subcommand* mode and your program does not get control.

Floating-Point Registers

You can refer to a floating-point register using the LIST or assignment-of-value subcommand by specifying a decimal integer followed by an E or D. The decimal integer indicates the number of the register and *must be* a zero, two, four, or six. An E indicates a floating-point register with single precision. A D indicates a floating-point register with double precision. You *must not* use floating-point registers for indirect addressing or in expressions.

Vector Registers

You can refer to a vector register using the LIST or assignment-of-value subcommand. You cannot use vector registers for indirect addressing or in expressions. Specify a vector register address by using the following format:

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

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

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

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

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

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

The list below shows several examples of specifying vector registers:

1V(*) Specifies the entire contents of vector register 1 in single precision.
1V(4) Specifies element 4 of vector register 1 in single precision.
0W(1) Specifies element 1 of vector registers 0 and 1 in double precision.

Vector Mask Register

You can refer to the vector mask register using the LIST or assignment-of-value subcommand. A vector mask register address consists of the decimal integer 0 followed by an M. You cannot use the vector mask register for indirect addressing or in expressions.

Access Registers

You can refer to an access register using the LIST, COPY, or assignment-of-value subcommand by specifying a decimal integer followed by an A. The decimal integer indicates the number of the register and must be in the range 0-15. You cannot use access registers for indirect addressing or in expressions.

Indirect Address

An indirect address is an absolute, relative, or symbolic address, (or a general register containing an address) of a location that contains another address. An indirect address *must* be followed by one or more indirection symbols to indicate a corresponding number of levels of indirect addressing.

The indirection symbols are:

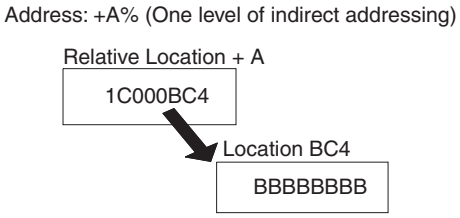
- The percent sign (%), indicating that the low-order three bytes of the address are used.
- The question mark (?), indicating that all 31 bits are used for the address.

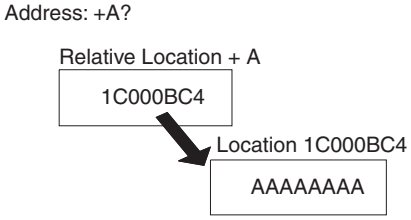
To use a general register as an indirect address, specify a decimal integer (0 through 15) followed by an R and a percent sign, or an R and a question mark. For example, if you want to refer to data whose address is located in register 7, you would specify:

```
7r%
```

Example: Use of a relative address to form an indirect address.

Address: +A% (One level of indirect addressing)





Example: Comparison of use of % and ?

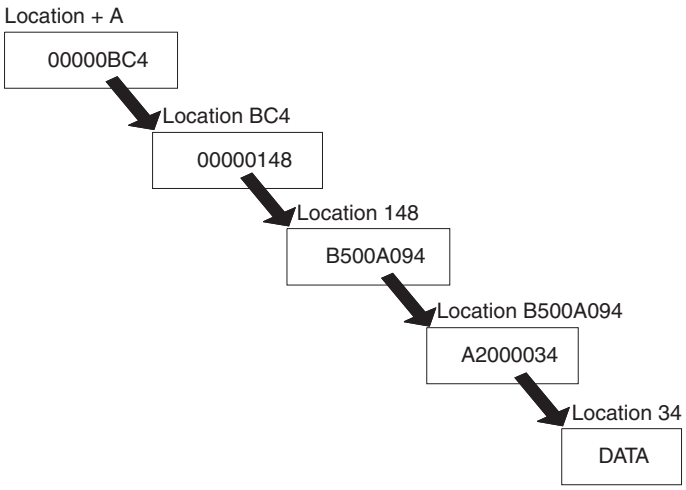
Address	**Data**
X	0100A080
0000A080	AAAAAAAA
0100A080	BBBBBBBB

Subcommand	**Data Displayed**
LIST X	0100A080
LIST X%	AAAAAAAA

Subcommand	Data Displayed
LIST X?	BBBBBBBB

Example: Indirect addressing using a combination of indirection symbols.

Address expression: +A%??% (Four levels of indirect addressing)



Address Expression

An address followed by any number of expression values. You can specify the address as:

- An absolute address
- A relative address (unqualified, partially or fully-qualified)
- A symbolic address (unqualified, partially or fully-qualified)
- An indirect address

An expression value consists of a plus or minus displacement value expressed as either 1 to 8 hexadecimal digits or 1 to 10 decimal digits from an address in virtual storage. Following are two examples of address expressions:

Decimal Example: address+14n specifies the location that is 14 bytes past that designated by ''address.''

Hexadecimal Example: address+14 specifies the location that is 20 decimal bytes past that designated by ''address.''

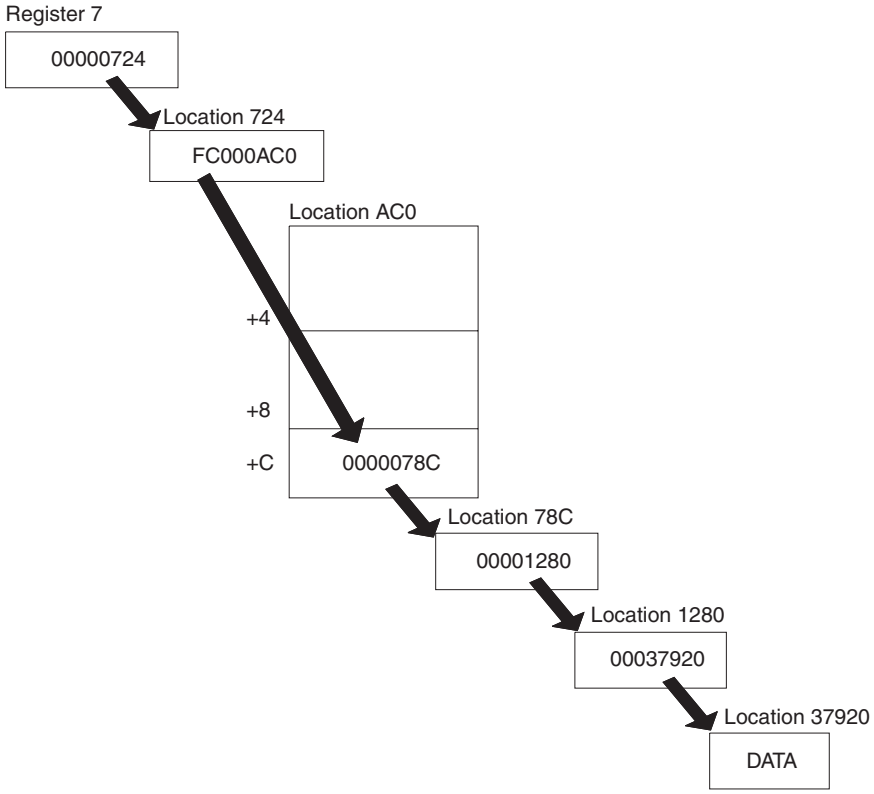
Decimal displacement (either plus or minus) is indicated by the n following the numeric offset. You can indicate up to 256 levels of indirect addressing by following the initial indirect address with a corresponding number indirection symbols (% or ?). An address expression is specified in the following format:

```
          {-}       [ ?    ] [{-}       [ ?    ] ]
address   {+} value [ % .. ] [{+} value  [ %  .. ] ]...
```

You can use any combination of percent signs and question marks after the value.

Example: Address expression with hexadecimal displacement using a combination of indirection symbols.

Address expression: 7R?%+C%?%



When processing an address expression, TEST and TESTAUTH check the high-order bit of the result of each addition or subtraction. If the bit is on, indicating a negative value or overflow condition, the address is rejected.

Restrictions on the Use of Symbols

The TEST and TESTAUTH command processors can resolve external and internal symbolic addresses, only if those addresses are available. Within certain limitations, symbolic addresses are available for both object modules (processed by the loader) and load modules (fetched by contents supervision). To ensure availability of symbols, use the EQUATE subcommand to define the symbols you intend to use. "Using Additional Features of TEST" on page 132 shows you how to do this.

External Symbols

You can access external symbols, such as CSECT names, for a program module, if the program was brought into main storage by the TEST or TESTAUTH command or one of its subtasks. This is the case for the program being tested, any program brought into storage through the tested program, and any program loaded by the LOAD subcommand.

External symbols for CSECT names that are in object modules are available only if the loader had enough main storage to build composite external symbol table dictionary (CESD) entries.

Internal Symbols

Internal symbols for load modules can be resolved if the CSECT containing the symbol was assembled with the TEST parameter, the module was link-edited with

the TEST parameter, and the program was brought into storage by the TEST or TESTAUTH command or one of its subtasks as previously explained. Names on EQU, ORG, LTORG, CNOP, and DSECT statements *cannot* be resolved.

You cannot access internal symbols for object modules.

Addressing Considerations

If the necessary conditions for symbol processing are not met, you can use absolute, relative, or indirect addressing, or you can define symbols with the EQUATE subcommand.

Symbols within DSECTs are available only if the DSECT name has been defined with the EQUATE subcommand.

For example, if NAME is a symbol in a DSECT named DATATBL, then to access the data associated with NAME, you would first have to determine the address to be used as a base address for the DSECT. (This is the address in the register on the assembler USING instruction.) If the address is in register 7, you can enter:

```
equate datatbl 7r%
```

This establishes addressability to the DSECT, allowing the symbol NAME and all other symbols in the DSECT to be accessed using the symbol.

TEST and TESTAUTH can access symbols and process CSECT names (to qualify addresses and satisfy deferred breakpoints) for a module loaded from a data set in LNKLIST concatenation, provided that the module was both assembled and link-edited with the TEST option, and the data set involved is not READ-protected. Symbols and CSECT names cannot be processed for a module accessed from LPA.

Examples of Valid Addresses in TEST and TESTAUTH Subcommands

The following are examples that can be used with subcommands. In these addresses, PROFIT is a module name, SALES is a CSECT name, and NAMES is a symbol.

Address:
Type of Address:

A23C40.
Absolute

+E4 Relative

5R% 24-bit indirect

5R? 31-bit indirect

NAMES
Symbol within program

.SALES.+26
Partially-qualified relative

14R%+28
Expression

PROFIT.SALES
Module and entry name

+16+10n
Expression

.SALES.NAMES
Partially-qualified symbol

PROFIT.SALES.NAMES+8n
Expression

DATA+10
Expression

.SALES
Entry name

PROFIT.SALES.NAMES
Fully-qualified symbol

6R%+4%+12n%%
Expression

PROFIT.SALES.+C0
Fully-qualified relative

Programming Considerations for Using TEST and TESTAUTH

When you use the TEST or TESTAUTH command, consider the environment in which your program is running. The topics that follow describe programming considerations for the following:

- Using 31-bit addressing
- Using the virtual fetch services
- Executing in a cross-memory environment
- Using the vector facility
- Using extended addressability
- Testing APPC/MVS transaction programs
- Evaluating a program's environment after testing

Considerations for 31-Bit addressing

- All subcommands that accept addresses can process addresses above 16 MB, regardless of the current addressing mode of the program.
- You can use the 31-bit indirection symbol (?) on any subcommands to reference data pointed to by 31-bit addresses.
- When TEST or TESTAUTH loads and executes a program, it uses the AMODE and RMODE characteristics to determine the addressing mode at entry, as well as whether the tested program is loaded above or below 16 MB.
- The AMODE operand on the CALL, GO, and RUN subcommands can change the addressing mode of the program being tested.
- The loader, which TEST and TESTAUTH invoke when testing an object module, loads the module above or below 16 MB based on the RMODE characteristics of the module's CSECTs. If the first CSECT is RMODE(ANY) and any other CSECTs are RMODE(24), the loader loads the module below 16 MB in storage and issues a warning message.
- Input passed in register 1 to the program being tested is below 16 MB in storage. If you invoke the TEST or TESTAUTH command with the CP operand, register 1 contains the address of the command processor parameter list (CPPL). Otherwise, register 1 contains the address of the input parameter list.

- The CALL subcommand places the return address of the tested program in register 14. The high-order bit of register 14 is set to reflect the addressing mode of the tested program.
- If the called program should not be invoked in the current addressing mode, specify AMODE on the CALL statement. When control is returned, verify that the addressing mode is appropriate before continuing execution.

Considerations for Using the Virtual Fetch Services

- External symbols are not available for a program fetch. For information on addressing considerations, see "Restrictions on the Use of Symbols" on page 105.
- Do not establish deferred breakpoints for a program managed by virtual fetch because they are ignored.
- If you are testing program A, which invokes program B using the virtual fetch services, you cannot use TEST or TESTAUTH subcommands to stop execution of program B.
- If, while testing program A, you want to debug program B, you can use the following method. Instead of allowing a virtual fetch GET request to pass control to program B, load and call program B using TEST or TESTAUTH subcommands.
  - Use the AT subcommand to establish a breakpoint immediately before the virtual fetch GET request in program A.
  - When you reach the breakpoint, use the LOAD subcommand to load a different copy of program B.
  - You can then establish breakpoints using the AT subcommand at any points in this copy of program B.
  - Use the CALL subcommand to execute program B. Specify an address on the RETURN parameter to bypass the virtual fetch GET request in program A.

Note: You cannot use the facilities of TEST or TESTAUTH to debug a program's interface with virtual fetch.

For a description of the virtual fetch services, see *z/OS MVS Using the Subsystem Interface*.

Considerations for a Cross-Memory Environment

- If an attention interruption occurs while the program being tested is executing in cross-memory mode, and you enter anything other than a null line, the cross-memory environment is terminated and a message is displayed.
- If TEST or TESTAUTH is used with cross-memory applications, access to storage by TEST or TESTAUTH subcommands is restricted to the home address space.
- If an abnormal termination (abend) occurs while a cross-memory application is executing outside the home address space, TEST and TESTAUTH do not preserve the cross-memory environment. The registers and PSW at the time of the abend and the abend code from the error message are the only debugging information available for a cross-memory abend.
- TEST provides only limited testing for programs that execute in cross-memory mode. The reason is that code running in cross-memory mode cannot issue any SVCs (except ABEND). Any system service that depends on SVCs is unavailable in cross-memory mode.

  If a breakpoint is set, TEST places a SVC in the code being tested. When the code being tested reaches the breakpoint, the SVC returns control to TEST

alerting it that a breakpoint has been hit. If the code under test is in cross-memory mode this SVC cannot be issued and an abnormal termination occurs.

Therefore, do not set breakpoints in code that will be running in cross-memory mode.

Breakpoints set on instructions that perform address space switching are intercepted by SVCs inserted by TEST.

Considerations for the Vector Facility

The TEST and TESTAUTH commands support the use of 16 vector registers, the vector status register, and the vector mask register. Each of the 16 vector registers contains a number of four-byte elements that are dependent on the model of the CPU. The vector status register has special information about the vector registers being processed.

You can use the TEST and TESTAUTH commands to display and modify the contents of the 16 vector registers, display the vector status register, set breakpoints at the vector opcodes, display the vector opcodes in their assembler language format, view the partial sum number, view the vector section size, and set fields in the vector status register.

- Use the LIST subcommand to display the contents of the 16 vector registers in one of the following formats:
  - A single element
  - Multiple elements
  - All elements at once

  You can list registers in single precision floating point, fixed- point fullword binary or hexadecimal format. The even numbered vector registers can also be displayed in double precision floating point format.
- You can use the LIST subcommand to display the vector mask register. TEST and TESTAUTH display the vector mask register in hexadecimal or binary format. The length of the data displayed depends on the model of the CPU that has the vector facility installed.
- Use the LISTVSR subcommand to display the vector status register.
- You can use the assignment function to modify the 16 vector registers in the same format as you list them, that is, one at a time, some, or all at once. You can use the assignment function to modify the vector mask register. The value you assign to the vector mask register must be binary or hexadecimal.
- You can use the LISTVP subcommand to display the partial sum number and vector section size.
- You can use the SETVSR subcommand to set fields in the vector status register. You can change the vector mask register control mode, update vector count (VCT), update vector interruption index (VIX), and update vector in-use bits (VIU).

For examples showing how to display and manipulate vector registers and elements, see “Testing Programs That Use the Vector Facility” on page 140.

Considerations for Extended Addressing

The TEST and TESTAUTH commands allow you to test programs that use extended addressing. You can use the TEST and TESTAUTH commands to display and modify the contents of the 16 access registers, set breakpoints at the opcodes

that support access registers, and display the opcodes in their assembler language format. However, you cannot set breakpoints for the program return (PR) instruction.

- Use the LIST subcommand to display:
  - The contents of access registers used to reference data in alternate address/data spaces. You can display the contents of the 16 access registers in either binary, hexadecimal or decimal format.
  - Data contained in alternate address/data spaces.
- Use the assignment function to modify the contents of the 16 access registers and of storage in alternate address/data spaces.
- Use the COPY subcommand to copy data to an alternate address/data space or from an alternate address/data space.
- Use the ASCMODE operand on the CALL, GO and RUN subcommands to change the address space control (ASC) mode for the executing program.

Note: You can set breakpoints at addresses in the primary address space only.

For examples showing how to display and manipulate access registers, see "Testing Programs That Use Extended Addressing" on page 141.

Considerations for Testing Inbound APPC/MVS Transaction Programs

You can use the TEST and TESTAUTH commands to test APPC/MVS transaction programs. You can use the commands with inbound transaction programs—those that are initiated in response to inbound conversation requests from their partner transaction programs. When testing an APPC/MVS transaction program:

- There must not be a transaction program existing in the TSO/E user's address space when the TSO/E TEST or TESTAUTH command is invoked to test an APPC/MVS transaction program. You should log on with a logon procedure that does not allocate APPC/MVS conversations if an APPC/MVS transaction program is to be tested. After TSO/E TEST has been invoked with the TP keyword, you can allocate DFM data sets. However, if you want to invoke TEST again, you may have to log off and then log on again.
- You cannot use the TSO/E TEST or the TESTAUTH command to completely test multi-trans (multiple transaction) transaction programs. You can, however, use TSO/E TEST to partially test an APPC/MVS multi-trans transaction program up to the point when the transaction program attempts to get the next transaction.
- The user-level transaction program profile for the LUs that are to be used for transaction program testing must be defined at installation time. To define a user-level transaction program profile, the LUADD statement in PARMLIB member APPCPMxx must include the TPLEVEL(USER) keyword.
- If you press the attention key while waiting for the inbound allocate request, and enter anything other than a null line, TSO/E cancels the test request.
- If you press the attention key twice to end the TSO/E TEST or TESTAUTH command, and enter anything other than a null line, TSO/E cancels the command. The TSO/E TEST or TESTAUTH command cleans up the transaction program you are testing unless the KEEPTP keyword is specified.

Considerations for a Tested Program's Environment

After the TEST or TESTAUTH command finishes processing a program, the program's environment has the following characteristics:

- The ASC mode is primary

- The primary, secondary, and home address spaces are equal
- The primary, secondary, and home address spaces are the same as when the program was invoked

If the program you are testing ends abnormally, the environment of the tested program might not be identical to the environment that it had prior to its ending. The environment of the tested program might have one or more of the following characteristics when it ends:

- The ASC mode was not primary
- The primary, secondary, and home address spaces were not equal
- The primary, secondary, and home address spaces were not the same as when the program was invoked

You can use the assignment functions of the TEST command to correct the ASC mode and address spaces. You can then correct the problem that caused the abend in the program and continue testing.

Refer to *z/OS TSO/E Command Reference* for additional information about the assignment functions of the TEST command.

If you cannot correct the environment of the tested program, you might be unable to test the remainder of the program. (For example, you cannot correct the environment of a tested program if that program was running in cross-memory mode.) You must end your test session, correct the abend, and then test the program again.

Chapter 17. A Tutorial Using the TEST Command

This chapter is presented in tutorial form, and is intended for TSO/E users who have never used the TEST command to test a program. Before reading this chapter, you must be familiar with the concepts and terminology presented in Chapter 16, "Testing a Program," on page 93.

This tutorial describes how to use TEST subcommands to test and debug a program. If you are an authorized user, you can use the TESTAUTH command to test authorized programs. Because TESTAUTH supports the same subcommands as TEST, you can use this tutorial to learn about using TESTAUTH.

This tutorial describes how to use the following subcommands:

Task	Subcommands
View storage and registers	LIST, LISTMAP, LISTPSW, LISTDCB, LISTDEB, LISTTCB, LISTVP, LISTVSR
Find addresses	WHERE
Control breakpoints	AT, OFF
Alter storage and registers	COPY, assignment of values function (=)
Alter vector registers	SETVSR
Add and alter symbols	EQUATE, DROP
Modify a base address	QUALIFY
Control program execution	GO, CALL, RUN
Obtain and free additional storage	GETMAIN, FREEMAIN
Obtain and free other programs	LOAD, DELETE
Obtain help information	HELP
Terminate a TEST session	END, RUN

The TEST subcommands use the address types described in "Addressing Conventions Associated with TEST and TESTAUTH" on page 100. This tutorial shows you how to use the following address forms and their notation:

Table 14. Address Forms Supported by TEST

Address Form	Notation
Single address	A single address.
Range of addresses	Two addresses separated by a colon.
List of addresses	Addresses enclosed in parentheses, and separated by blanks or commas.

The tutorial is presented in the following sequence:

1. "Preparing to Use TEST" on page 115 describes how to assemble and link-edit your program, and how to invoke TEST, run the program, and terminate the TEST session.

2. "Viewing a Program in Storage" on page 117 shows several examples using the LIST subcommand to view registers and storage. It also shows how to use the different forms and types of addresses TEST supports.
3. "Monitoring and Controlling Program Execution" on page 125 shows you how to use breakpoints with the AT, GO, and OFF subcommands.
4. "Altering Storage and Registers" on page 129 illustrates the use of the assignment function and the COPY subcommand for altering storage to set up test cases.
5. "Using Additional Features of TEST" on page 132 shows you how to use the specialized LIST subcommands: LISTMAP, LISTPSW, LISTDCB, LISTDEB, and LISTTCB. Also, it further demonstrates the use of the WHERE subcommand, and introduces the QUALIFY and EQUATE subcommands.

"More TEST Subcommands" on page 139 follows the tutorial and describes the uses of the GETMAIN, FREEMAIN, LOAD, DELETE, and CALL subcommands. "Testing Programs That Use the Vector Facility" on page 140 discusses how to test programs that use vector registers and elements.

How to Use This Tutorial

This tutorial is presented in a two-page format. The left-hand pages contain sample terminal sessions. Information that you enter at the terminal is in lower case; the responses from TSO/E are in upper case. The right-hand pages contain explanations of the sample terminal sessions.

To use this tutorial at your terminal, enter the commands shown on the sample terminal sessions (left-hand pages) and, at the same time, read the corresponding explanations on the right-hand pages. Numbers provide a cross-reference from the sample terminal sessions to the explanations.

You will need copies of the sample programs that are supplied in SYS1.SAMPLIB, along with their corresponding object files. The table below shows the members of SYS1.SAMPLIB that contain the source and object data used in this tutorial.

Source Member	Object Member
IKJSAMP1	IKJOBJ1
IKJSAMP2	IKJOBJ2

To use this tutorial at your terminal, create a data set called '*prefix*.SAMPLE1.ASM' that is a sequential data set of one track, with DCB characteristics of RECFM=FB, LRECL=80, BLKSIZE=3120. Copy the source code from 'SYS1.SAMPLIB(IKJSAMP1)' into your data set.

Note: The source code and the assembler listings for the sample programs used in this tutorial are also shown in "Example programs for the TEST tutorial" on page 144.

This tutorial guides you through:

- Using the subcommands of TEST to:
  - Change the contents of general purpose registers.
  - Control the execution of a program including the re-execution of instructions.
  - Change the contents of main storage variables and buffers.
  - Assign names to main storage locations.

  - Obtain and free additional main storage, as needed.
  - Locate the real address of your program.
  - List the PSW.
- Using the various methods of addressing main storage under TEST.

Preparing to Use TEST

To test your program, it must be in object module or load module form. Therefore, you must first perform an assembly or an assembly and link-edit on your program. You can do this with a batch job, or by issuing TSO/E commands in the foreground. This tutorial uses foreground commands.

```
(1) READY

(2) asm sample1 test

(3) ASSEMBLER (XF) DONE
    NO STATEMENTS FLAGGED IN THIS ASSEMBLY
    HIGHEST SEVERITY WAS    0
    OPTIONS FOR THIS ASSEMBLY
     ALIGN, ALOGIC, BUFSIZE(STD), NODECK, NOESD, FLAG(0), LINECOUNT(55),
     NOLIST, NOMCALL, YFLAG, WORKSIZE(2097152),
     NOMLOGIC, NUMBER, OBJECT, NORENT, NORLD, STMT, NOLIBMAC,
     TERMINAL, TEST, NOXREF(SHORT)
     SYSPARM()
    READY

(4) link sample1 test

    READY
```

1. After your data set is created, place yourself at READY mode of TSO/E.
2. The command to assemble a program is ASM:
   - ASM requires, as the first operand, the name of the data set that contains the program to be assembled.

     In this example, all you specify is SAMPLE1, because TSO/E naming conventions places your prefix (usually your user ID) to the left of the name you enter, and the type ASM to the right, yielding the fully-qualified name '*prefix*.SAMPLE1.ASM'.
   - The ASM command assembles your program and produces an output object module. This will be placed in the data set '*prefix*.SAMPLE1.OBJ', which will be created automatically if it does not exist.
   - You can specify assembler options after the data set name. The example requests the TEST option, which allows you to access internal symbols in your program. You will see the effect of this under the TEST command.
3. After the assembly is done, you should receive the final diagnostics, NO STATEMENTS FLAGGED.

   If you have assembly errors, correct your source code and reassemble until you have no errors.
4. After the program is assembled, use the LINK command to link-edit it.
   - As its first operand, the LINK command requires the name of the data set containing the object module to be link-edited.

     Again, the name SAMPLE1 suffices because TSO/E naming conventions places your prefix on the left, and the type OBJ on the right, yielding '*prefix*.SAMPLE1.OBJ'.

- Similar to ASM, you can specify link-edit options on the command. TEST is the option specified, and it continues to allow you to access internal symbols.
- The output of the LINK command is a load member of a partitioned data set (PDS). The default data set is '*prefix*.SAMPLE1.LOAD(TEMPNAME)', which is automatically created if it does not exist.

Now that your program is in load module form, you can execute it using the TEST command.

```
(5) test sample1
(6)  TEST

(7) go
(8)  IKJ57023I PROGRAM UNDER TEST HAS TERMINATED NORMALLY+
(9)  TEST

(10) help

SUBCOMMANDS -
  ASSIGN,AND,AT,CALL,COPY,DELETE,DROP,END,EQUATE,FREEMAIN,GETMAIN,GO,
  LIST,LISTDCB,LISTDEB,LISTMAP,LISTPSW,LISTTCB,LISTVSR,LOAD,OFF,OR,
  QUALIFY,RUN,WHERE.
IKJ56804I FOR MORE INFORMATION ENTER HELP SUBCOMMANDNAME OR HELP HELP
TEST

(11) end
(12)  READY
```

5. The TEST command requires just the name of the program to test.

   Again, you need only specify SAMPLE1, because TSO/E supplies the prefix on the left and the TEST command assumes the type LOAD. Also, the default member name is TEMPNAME, yielding the data set '*prefix*.SAMPLE1.LOAD(TEMPNAME)'.
6. TEST displays a TEST mode message, indicating that it is your turn to enter TEST subcommands to control TEST's processing.

   Right now, your program has been loaded into storage, but has not yet started execution.
7. Run your program with the GO subcommand.

   This subcommand tells TEST to start executing the program wherever it left off (in this case, at the start of the program), and continue until the program stops.
8. The program runs to normal completion, meaning there was no abend.
9. TEST returns control to the terminal with another mode message.

   Now that the program has executed, you may wish to use other TEST subcommands to view and alter the program.
10. The HELP subcommand provides a list of the TEST subcommands.

    HELP also describes the function, syntax, and operands of the the TEST command (not shown).
11. To terminate TEST processing, use the END subcommand.
12. TEST returns to READY mode.

So far, you have seen how to get a program ready for TEST using foreground assembly and link-edit, how to invoke TEST, run the program, and terminate the TEST session.

Viewing a Program in Storage

One function you will need when executing under TEST is to view the contents of your data items and registers.

The LIST subcommand (abbreviated "L") provides this facility, and it requires just one operand — the register(s) or address(es) of storage you want to view.

When using TEST, you specify what you want to work with via an *address*. There are several forms of addresses: registers, symbolic, relative, absolute, and indirect, as you will see in the following examples.

This section of the tutorial shows you the LIST subcommand of TEST and uses LIST to demonstrate the various forms of the addresses you can use on the TEST subcommands.

You will need to view the source code for SAMPLE1 while performing the following exercise; a listing of it appears in "Example programs for the TEST tutorial" on page 144.

```
(1) test sample1

     TEST

(2) list 0r

     0R  0001AD0C
     TEST

(3) list 5r:8r

     5R  FFFFFFFF   6R  FFFFFFFF   7R  FFFFFFFF   8R  FFFFFFFF
     TEST

(4) list 14r:3r

     14R  0000B82C  15R  0001CF68   0R  0001AD0C   1R  0001BFB0
      2R  FFFFFFFF   3R  FFFFFFFF
     TEST

(5) list (3r 9r 11r)

     3R  FFFFFFFF
     9R  FFFFFFFF
     11R  FFFFFFFF
     TEST

(6) go

     IKJ57023I PROGRAM UNDER TEST HAS TERMINATED NORMALLY+
     TEST

(7) list 0r:15r

     0R  0001AD0C   1R  0001BFB0   2R  FFFFFFFF   3R  FFFFFFFF
     4R  FFFFFFFF   5R  FFFFFFFF   6R  FFFFFFFF   7R  FFFFFFFF
     8R  FFFFFFFF   9R  FFFFFFFF  10R  FFFFFFFF  11R  FFFFFFFF
    12R  FFFFFFFF  13R  0001BFB8  14R  0000B82C  15R  0001CF68
    TEST
```

1. Test your program again, stopping at the start of your program.
2. To see the contents of a register, use the LIST subcommand, specifying the register number followed by the letter "R", meaning "register".

Hint: Be careful not to get confused with a common programming convention of naming the registers “R0”, “R1”, and so on. Under TEST, such names would designate a main storage location, as you will see later.

3. At times you may wish to view a consecutive range of registers. The notation for this is a colon separating the two registers of the range.

   This subcommand requests registers 5 through 8.

4. Note that registers can wrap around, so you may specify a higher register number first.
5. You may wish to view the contents of a list of registers. In this case, you must enclose the list in parentheses and separate each register in the list with a comma or a blank.

   This subcommand requests the contents of registers 3, 9, and 11.

6. Now execute the program.
7. This subcommand displays all of the registers, after the program has terminated.

   Note that TEST initializes registers 2 through 12 with X'FFFFFFFF' to allow you to see which registers are changed by the tested program.

   The above examples of registers show the three standard forms in which addresses can be specified on the TEST subcommands:

   - A single address
   - A range of addresses (separated by a colon)
   - A list of addresses (enclosed in parentheses and separated by blanks or commas)

   This is true whether the addresses are registers or main storage locations.

```
(8) list charcon

     CHARCON  TEST EXAMP
     TEST

(9) list fullcon

     FULLCON  -1
     TEST

    list halfcon

     HALFCON  +32
     TEST

(10) list adcon

      ADCON  118676
      TEST

     list hexcon

      HEXCON  0000001F
      TEST

     list packcon

      PACKCON  +25
      TEST

     list bincon

      BINCON  10101100
      TEST

(11) list charcon:bincon

      CHARCON  TEST EXAMP...*.
      TEST

(12) list (adcon packcon 5r)

      ADCON  118676
      PACKCON  +25
       5R  FFFFFFFF
      TEST
```

Viewing the contents of main storage locations is just as easy as viewing registers. All you need to specify is the address or addresses of the locations you want to view.

Because you assembled and link-edited your program with the TEST option, you will be able to view the contents of your program by using the symbolic names on your data items and instructions.

8. This causes TEST to display the contents of CHARCON in character form. TEST chooses character because you defined the item in your assembler code as CL10.
9. Listing a fullword or halfword item causes TEST to convert the value to a signed decimal number, which is easier to read than binary.
10. Here are the other data items in the program, listed with the default characteristics implied by their definition.
11. Again, you may use the range and list forms of the LIST subcommand. This subcommand displays all of the storage between CHARCON and BINCON. TEST uses the first data type to determine how to display the storage.

    Note: A dot indicates an unprintable character.

12. Here is a list of addresses, in parentheses. You can mix registers and storage locations in the list.

```
(13) list save

      SAVE
            +0  +0
            +4  +114616
            +8  +0
            +C  +0
           +10  +0
           +14  +0
           +18  +0
           +1C  +0
           +20  +0
           +24  +0
           +28  +0
           +2C  +0
           +30  +0
           +34  +0
           +38  +0
           +3C  +0
           +40  +0
           +44  +0
      TEST

(14) list fullcon:bincon

      FULLCON
            +0  -1
            +4  +31
            +8  +2155461
            +C  -488423227
           +10  -406727465
           +14  +604
           +18  -1409285540
      TEST
```

13. Notice that displaying an item defined with a duplication factor causes TEST to recognize this in its format.
14. Sometimes LIST's default data type is not very helpful; here all the storage after FULLCON is treated as fullwords.

    Note: The default is the type specified in the program for the first item in the range.

```
(15) list bitcon

(16)  IKJ57280I ADDRESS BITCON NOT FOUND+
      IKJ56703A REENTER THIS OPERAND -

(17) ?

      IKJ57280I BITCON NOT IN INTERNAL SYMBOL TABLE FOR TEMPNAME . SAMP1

(18) |

      TEST

(19) list stop1

      STOP1
           +0    LH    R3,122(,R12)
      TEST

     list stop2

      STOP2
           +0    A     R3,114(,R12)
      TEST

     list stop3

      STOP3
           +0    ST    R3,118(,R12)
      TEST

(20) list stop1:stop3

      STOP1
           +0    LH    R3,122(,R12)
           +4    A     R3,114(,R12)
           +8    ST    R3,118(,R12)
      TEST

(21) list (stop1 stop3)

      STOP1
           +0    LH    R3,122(,R12)
      STOP3
           +0    ST    R3,118(,R12)
      TEST
```

15. This is a sample of an error, because the symbol BITCON does not exist in the program.
16. TEST asks you to reenter the incorrect address.

    A "+" sign at the end of a message means that more information is available.
17. You can obtain this extra information by typing "?" with nothing else on the line.
18. To cancel the erroneous subcommand and return to the TEST mode, press the attention key. This produces the "|" symbol on the screen.
19. TEST also allows you to display instructions. You see the explicit assembler form.
20. This subcommand shows a range of instructions.
21. This subcommand shows a list of instructions.

```
(22) list charcon x

     CHARCON
          +0  E3C5E2E3 40C5E7C1 D4D7
     TEST

(23) list fullcon x

     FULLCON  FFFFFFFF
     TEST

(24) list fullcon:bincon x

     FULLCON
          +0  FFFFFFFF 0000001F 0020E3C5 E2E340C5
              E7C1D4D7 0000025C AC
     TEST

(25) list charcon x length(5)

     CHARCON  E3C5E2E3 40
     TEST

(26) list save x multiple(5)

     SAVE
          +0  00000000
          +4  0001BFB8
          +8  00000000
          +C  00000000
         +10  00000000
     TEST

    list charcon length(2) multiple(5)

     CHARCON
          +0  TE
          +2  ST
          +4   E
          +6  XA
          +8  MP
     TEST

(27) list save:bincon print(sample1)

     TEST
```

Now, for some other operands of LIST. You may wish to read the syntax of the LIST subcommand of TEST in *z/OS TSO/E Command Reference* at this time.

22. After the address on LIST, you can specify a data type that TEST should use to display storage. This overrides the defined data type of the symbol.

    This requests TEST to display CHARCON in hexadecimal.

23. Here is FULLCON displayed in hexadecimal.
24. Here is a range of storage, also in hexadecimal.
25. The *length* operand specifies the number of bytes you want to have displayed. This overrides the defined length of the symbol.
26. The *multiple* operand allows you to specify a multiplicity factor for the item. You can use this to display a table, or to format a long area for readability.
27. The *print* operand allows you to specify a data set, rather than the terminal, to which the list should be directed. You may later print the data set to read the information easily.

The name of the data set that will contain the list consists of the name you specify, preceded by your prefix and followed by TESTLIST as the descriptive qualifier. Therefore, in this example, the information will be placed in '*prefix*.SAMPLE1.TESTLIST'.

```
(28) where charcon

(29) 1CFEA. LOCATED AT +82 IN TEMPNAME.SAMP1 UNDER TCB LOCATED AT 7C2560.
     TEST

(30) list 1cfea.

      0001CFEA.  E3C5E2E3
      TEST

(31) list 1cfea. c length(5)

      0001CFEA.  TEST
      TEST

(32) list +82 length(8)

          +82  E3C5E2E3 40C5E7C1
      TEST

(33) list save+20

      0001CFB4.  00000000
      TEST

(34) list save+32n

      0001CFB4.  00000000
      TEST

(35) list save-10

      0001CF84.  5030C076
      TEST

(36) where save-10

 1CF84.  LOCATED AT +1C IN TEMPNAME.SAMP1 UNDER TCB LOCATED AT 7C2560.
TEST

(37) list +1c

          +1C  5030C076
      TEST
```

So far, you have seen the various operands of the LIST subcommand of TEST. All addresses were specified as *symbolic* addresses. Normally, your symbolic names cannot be referenced at execution time, but using the TEST operand on the ASM and LINK commands caused your symbols to be available at execution time.

Other forms of addresses besides symbolic are absolute, relative, indirect, and address expressions, as shown below.

28. To obtain the absolute address of data or an instruction, use the WHERE subcommand.

    This example is requesting the location of CHARCON.

29. TEST gives the absolute address and the address relative to the CSECT.

30. To use an absolute address in a LIST command, follow the address with a period. The period is a signal that you have entered an absolute address and not a register number or a symbolic address.

Because TEST does not know the data type of this address, it displays the data in hexadecimal, for the default length of four.

31. Using the data type and length operands lets you control how much storage is displayed, and in what format.
32. A *relative* address is indicated by a plus sign followed by a hexadecimal number. It represents the displacement of something from the beginning of the CSECT. (Later you will be able to change this "base" address.)
33. This is an *address expression*, meaning a symbolic, absolute, or relative address followed by a plus or a minus sign, followed by a hexadecimal number (modifying value).
34. If you prefer to express the modifying value in decimal, you must follow the number with the letter "n".

    "+32n" is the same as "+20".
35. Here is a negative modifying value.
36. Just to check the address involved, WHERE tells you the absolute and relative addresses of SAVE-10.
37. This displays the same storage location as SAVE-10.

```
(38) list 1r

       1R  0001BFB0
      TEST

(39) list 1bfb0.

      0001BFB0.  8001BFB4
      TEST

(40) list 1r%

      0001BFB0.  8001BFB4
      TEST

(41) list 1r%%

      8001BFB4.  00000000
      TEST

     end

      READY

(42) listcat

      IN CATALOG:USERCAT
      USER01.SAMPLE1.ASM
      USER01.SAMPLE1.LOAD
      USER01.SAMPLE1.OBJ
      USER01.SAMPLE1.TESTLIST
      READY
```

Now to look at *indirect* addresses.

38. Viewing the contents of a register, as you have already seen.
39. Here is the data at the address specified in register 1.
40. Specifying a "%" or a "?" after a register says you want to view, not the register, but the data to which the register points.

    This subcommand produces the same result as the previous two subcommands.

    When you use "%", the effective address is treated as a 24-bit address. When you use "?", the effective address is treated as a 31-bit address.

41. You can stack "%" and "?" signs to indicate multiple levels of indirect addressing.

    This means go to the address to which register 1 points, and then use that as an address, and view the data there.
42. After ending TEST, a LISTCAT command displays the data sets involved so far. Notice that the SAMPLE1.TESTLIST data set was created from your LIST subcommand with the PRINT operand.

In this section of the tutorial, you have seen several examples of the LIST subcommand to view registers and storage. You have also seen the three general forms of address operands on TEST subcommands:
- Single address
- Range of addresses
- List of addresses.

Finally, you have seen the various types of addresses:
- Symbolic
- Absolute
- Relative
- Indirect
- Expressions
- Registers

Monitoring and Controlling Program Execution

So far, you have simply executed your program from start to finish, viewing storage before execution began and after the program terminated. TEST also allows you to interrupt execution of your program at selected points so that you can use the TEST subcommands to view storage and perform other functions. The points at which execution is interrupted are known as *breakpoints*.

You establish breakpoints with the AT subcommand of TEST. Whenever TEST encounters a breakpoint, it returns control to you at the terminal. You can inspect and modify storage and registers, and then resume execution at the breakpoint or elsewhere. This facility allows you to monitor your program execution.

You can remove breakpoints with the OFF subcommand.

```
(1) test sample1

     TEST

(2) at stop1

     TEST

(3) at stop3

     TEST

(4) go

(5)  IKJ57024I AT STOP1
     TEST

(6) go

(7)  IKJ57024I AT STOP3
     TEST

(8) go

     IKJ57023I PROGRAM UNDER TEST HAS TERMINATED NORMALLY+
     TEST

(9) off

     TEST

(10) go +0
      IKJ57023I PROGRAM UNDER TEST HAS TERMINATED NORMALLY+
      TEST
```

1. You are testing program SAMPLE1 again.
2. The AT subcommand establishes breakpoints at one or more instructions in your program.

   This is setting a breakpoint at the label STOP1.
3. This sets another breakpoint at the label STOP3.
4. Start execution of the program.
5. When TEST encounters a breakpoint, it returns control to the terminal.

   You receive control *before* execution of the instruction where the breakpoint was placed.
6. The GO subcommand resumes execution at the point it was interrupted.
7. The program stops at the second breakpoint and returns control to the terminal.
8. The last GO causes the program to run to completion.
9. The OFF subcommand with no operands removes all breakpoints from the program.
10. The GO subcommand with an address causes TEST to restart execution from that address. “+0” is the relative address of the start the program, so this subcommand causes TEST to re-execute the program from the beginning.

    Note: Register contents are *not* reset to the original entry values when you execute this command.

    Because there are no breakpoints, the program runs to completion.

```
(11) at stop1:stop3

      TEST

(12) go +0

      IKJ57024I AT STOP1
      TEST

(13) list 3r

      3R  FFFFFFFF
      TEST

(14) go

      IKJ57024I AT +4 FROM STOP1
      TEST

(15) list 3r

      3R  00000020
      TEST

(16) at stop1:stop3 (list 3r)

      TEST

(17) go +0

(18)  IKJ57024I AT STOP1
       3R  00000020
      TEST

(19) go

      IKJ57024I AT +4 FROM STOP1
       3R  00000020
      TEST

     go

      IKJ57024I AT +8 FROM STOP1
       3R  0000001F
      TEST

     go

      IKJ57023I PROGRAM UNDER TEST HAS TERMINATED NORMALLY+
      TEST
```

11. By specifying a range of addresses on the AT subcommand, you can cause TEST to stop at every instruction in the range. This allows you to "instruction step" through your program.
12. Start from the beginning of the program again.
13. View the contents of register 3 at the STOP1 breakpoint.
14. Continue execution.
15. View register 3 again.

    If you want to perform some standard action, such as listing registers or storage, every time TEST hits a breakpoint, you may include a list of TEST subcommands on the AT subcommand.

    The list of subcommands, even if it is only one subcommand, must be enclosed in parentheses after the address(es) of the breakpoint(s).
16. This is requesting TEST to list the contents of register 3 before it returns control to you at a breakpoint.

This action overrides the previous breakpoints established at STOP1 through STOP3. (You don't have to issue an intervening OFF.)

17. Run the program from the beginning once more.
18. TEST displays register 3 before giving control to the terminal.
19. You must still enter GO to continue execution.

```
(20) at stop1:stop3 (list 3r;go)

      TEST

(21) go +0

      IKJ57024I AT STOP1
       3R  00000020
      IKJ57024I AT +4 FROM STOP1
       3R  00000020
      IKJ57024I AT +8 FROM STOP1
       3R  0000001F
      IKJ57023I PROGRAM UNDER TEST HAS TERMINATED NORMALLY+
      TEST

(22) off (stop1 stop2)

      TEST

(23) go +0

      IKJ57024I AT +8 FROM STOP1
       3R  0000001F
      IKJ57023I PROGRAM UNDER TEST HAS TERMINATED NORMALLY+
      TEST

     end

      READY
```

20. If you do not want to type GO after every breakpoint, include GO in the list of subcommands.

    The individual subcommands must be delimited by semicolons within the parentheses.
21. Start from the beginning again. TEST displays register 3 at every breakpoint, but does not give control to the terminal because of the GO subcommand.

    The program runs to completion.
22. You can selectively remove breakpoints by specifying the address(es), either single, list, or range, on the OFF subcommand.

    This is removing a list of breakpoints (at addresses STOP1 and STOP2). The breakpoint at STOP3 remains.
23. Running from the beginning, TEST shows the breakpoint at STOP3 only, with no stops made.

In this section of the tutorial, you have seen how to interrupt the execution of your program by establishing breakpoints at instructions of your choice. You can do this when you first enter TEST, before your program starts executing. Of course, you can add additional breakpoints at any time during TEST processing.

To remove breakpoints, use the OFF subcommand of TEST.

You can start execution at any point in your program by specifying an address on the GO subcommand of TEST.

Altering Storage and Registers

TEST provides subcommands with which you can alter the contents of main storage and registers. Two subcommands used in this tutorial are the assignment function and the COPY subcommand. Assignment allows you to alter storage or registers by entering a new value from the terminal. COPY allows you to move data from one storage location or register to another.

You can use these functions in preparing test cases:

- To "set up" various combinations of data items, causing different sections of your program to be executed.
- To make temporary "fixes" while your program is running.

```
(1) test sample1

     TEST

(2) charcon=c'abcde'

     TEST

(3) list charcon

     CHARCON  abcdeEXAMP
     TEST

(4) charcon=c'ABCDE     '

     TEST

    list charcon

     CHARCON  ABCDE
     TEST

(5) 5r=f'100'

     TEST

    list 5r

      5R  00000064
     TEST
```

1. Execute TEST for your program again.
2. This is an example of the assignment function. CHARCON is given the character ("c") value 'abcde'.

   There is no subcommand name for assignment. Just type the receiving field followed by an equal sign, followed by the data type (same as for LIST), and the value in quotes.
3. Notice that the assignment works somewhat differently than an assembler language "DC" in that it does not pad character strings with blanks.

   Also, if you enter your characters in lower case, they are not translated to upper case.
4. To get upper case and padding with blanks, explicitly request it in the value you are assigning.
5. Here is an assignment to a register. (Note the padding to the left with zeros.)

```
(6) list (hexcon fullcon) x

     HEXCON  0000FD38
     FULLCON  FFFFFFFF
     TEST

(7) copy hexcon fullcon

     TEST

(8) list (hexcon fullcon) x

     HEXCON  0000FD38
     FULLCON  0000FD38
     TEST

(9) charcon=c'ABCDEFGHIJ'

     TEST

    list charcon

     CHARCON  ABCDEFGHIJ
     TEST

(10) copy charcon charcon+3

      TEST

     list charcon

(11)  CHARCON  ABCABCDHIJ
      TEST

(12) charcon=c'ABCDEFGHIJ'

      TEST

(13) copy charcon charcon+1 L(6)

      TEST

(14) list charcon

      CHARCON  AAAAAAAHIJ
      TEST
```

6. View the contents of HEXCON and FULLCON.
7. The COPY subcommand requires two operands: the “from” address and the “to” address. It always moves four bytes by default.

   Here, COPY moves the contents of HEXCON to FULLCON.
8. Because both of the items are four bytes long, their contents are the same now.
9. Assign a new value to CHARCON.
10. Issue the COPY subcommand for the area where the fields overlap.
11. TEST picks up the first four bytes of CHARCON and moves it to CHARCON+3.
12. Set up CHARCON again.
13. Now the “to” field address is just one byte greater than the “from” field address. Also, a length of six bytes is specified with the LENGTH operand, “L(6)”.
14. In this situation, the first byte of the “from” location is propagated for the specified length.

```
(15) list 14r:1r

       14R  0000B82C  15R  0001CF68   0R  0001AD0C   1R  0001BFB0
      TEST

(16) copy 14r save+12n L(16)

      TEST

(17) list save+12n L(16) x

      0001CFA0.  0000B82C 0001CF68 0001AD0C 0001BFB0
      TEST

(18) list stop1 x L(4)

      STOP1  4830C07A
      TEST

(19) where stop1

 1CF7C. LOCATED AT +14 IN TEMPNAME.SAMP1 UNDER TCB LOCATED AT 7C2850.
TEST

(20) 6r=f'0'

      TEST

(21) copy stop1 6r

      TEST

     list 6r

       6R  4830C07A
      TEST

(22) copy stop1 6r pointer

      TEST

     list 6r

       6R  0001CF7C
      TEST

     end

      READY
```

15. View registers 14, 15, 0, and 1.
16. You can use COPY to restore registers. In this case, the registers are considered contiguous storage, so the LENGTH operand picks up 16 bytes of registers starting from register 14 (registers 14, 15, 0, and 1), and moves the information to SAVE+12n (remember, the “n” means decimal).
17. View the saved registers.
18. View the instruction STOP1.
19. Obtain the address of the instruction STOP1.
20. Clear register 6 (all zeros).
21. Copying storage to a register “loads” that storage into the register.
22. Issuing the same command, but with the POINTER operand, causes TEST to move, not the data at STOP1, but rather the address of STOP1, to the register, (in effect, performing a Load Address).

Altering storage and registers

In this section of the tutorial, you have seen several examples of the assignment function and the COPY subcommand of TEST.

Assignment allows you to alter the contents of storage or registers by supplying a new value from the terminal. You must also specify the data type of the value you are assigning.

COPY allows you to move data from registers or storage to other registers or storage. It is roughly equivalent to the various Move, Load, and Store instructions available in System/370 Assembler Language.

Using Additional Features of TEST

You have been introduced to the basic features of TEST, so that you are now able to view your program in storage, modify your program in storage, monitor your program's execution using breakpoints, and restart execution at any point in your program. You have also seen the various forms of addresses used on TEST subcommands.

This section of the tutorial shows you some of the additional features of TEST, including the specialized LIST subcommands and some further addressing possibilities.

The exercise in this section uses the IKJSAMP2 member of SYS1.SAMPLIB. To perform this exercise at your terminal, create a data set called '*prefix*.SAMPLE2.ASM' and copy the source code from 'SYS1.SAMPLIB(IKJSAMP2)' into your data set. A copy of the source code and assembler listing for the sample program used in this exercise are also included in "Example programs for the TEST tutorial" on page 144.

The source program in SAMPLE2.ASM will assemble and link-edit correctly, but it will not run to normal completion. There is an error in the program that you will locate and fix during the following TEST session.

Note: The program in SAMPLE2.ASM was assembled for this exercise on an MVS/370 system. The assembler output will not match what is shown here. Therefore, you should skip the assemble step in the tutorial (step 1) and copy 'SYS1.SAMPLIB(IKJOBJ2)' into your own data set called '*prefix*.SAMPLE2.OBJ'.

```
     READY

(1) asm sample2 test

     ASSEMBLER (XF) DONE
     NO STATEMENTS FLAGGED IN THIS ASSEMBLY
     HIGHEST SEVERITY WAS    0
     OPTIONS FOR THIS ASSEMBLY
       ALIGN, ALOGIC, BUFSIZE(STD), NODECK, NOESD, FLAG(0),
       LINECOUNT(55), NOLIST, NOMCALL, YFLAG, WORKSIZE(2097152),
       NOMLOGIC, NUMBER, OBJECT, NORENT, NORLD, STMT, NOLIBMAC,
       TERMINAL, TEST, NOXREF(SHORT), SYSPARM()
     READY

(2) link sample2 test

     READY

(3) alloc dd(outdd) da(*)

     READY

(4) test sample2

     TEST

(5) at addit

     TEST

(6) go

(7)  IKJ57024I AT ADDIT
     TEST

(8) listmap

     REGION SIZE 007FB000 AT ADDRESS 00005000
     REGION SIZE 7D800000 AT ADDRESS 02800000
     UNDER TCB AT  007C2AA8
          PROGRAM NAME  LENGTH   LOCATION
          TEMPNAME     000000B0  0001CF50
     ACTIVE RBS:  TYPE   PROGRAM-ID
                  PRB     TEMPNAME
     SUBPOOL INFORMATION:
     NUMBER   LOCATION    LENGTH
        0     0001B000   00001000
       78     0000B000   00001000
     IKJ57395I MAP COMPLETE
     TEST
```

1. Assemble your SAMPLE2.ASM data set with the TEST operand.
2. Link-edit the data set with the TEST operand.
3. This program requires an output data set with a ddname of OUTDD. The ALLOC command of TSO/E is the equivalent of a JCL DD statement. This command allocates the ddname OUTDD to the terminal by specifying "DA(*)".
4. Test the load module.
5. Set a breakpoint at the label ADDIT.
6. Start execution.
7. The program stops at the breakpoint.
8. The LISTMAP subcommand of TEST provides a map of the storage in your region.

LISTMAP is one of the specialized LIST subcommands. Like LIST, it has an optional PRINT operand that allows you to specify the name of a data set to which you would like the output of the subcommand directed.

```
(9) listpsw
     IKJ57652I PSW LOCATED AT 7B8208
       XRXXXTIE   KEY  XMWP  AS  CC  PROGMASK  EA BA  INSTR ADDR
       01000111    8   1101  00  01    0000     0  1   0001DEBE
     TEST

(10) listpsw addr(20.)

      IKJ57652I PSW LOCATED AT 20
        XRXXXTIE   KEY  XMWP  AS  CC  PROGMASK  EA BA  INSTR ADDR
        01000111    8   1101  00  10   0000      0  1   00CB9D60
      TEST

(11) listdcb outdcb

      IKJ57652I DCB LOCATED AT 01DF9C
        DEVICE INTERFACE SEGMENT
         RELAD    KEYCN        FDAD        DVTBL   KEYLE  DEVT  TRBAL
        00000000   00    0000000000000000  000000   00     4F   0000

        COMMON INTERFACE
        BUFNO  BUFCB   BUFL  DSORG   IOBAD
         01    01EFC0  0000  4000   00000001

        FOUNDATION EXTENSION
        HIARC-BFTEK-BFALN  EODAD   RECFM  EXLST
               06          000001   80    000000

        FOUNDATION
        TIOT  MACRF  IFLGS  DEBAD   OFLGS
        0504  0050    00    7B7A24   92
        TEST

(12) listdcb outdcb field(dcbdebad)

      IKJ57652I DCB LOCATED AT 01DF9C
      DEBAD
      7B7A24
      TEST
```

9. The LISTPSW subcommand displays the contents of the PSW, the current PSW by default.
10. With the ADDR operand, you can specify the address of the PSW you wish to see.

    LISTPSW also has the PRINT operand.
11. The LISTDCB subcommand lists the contents of the DCB, with fields labelled. You must specify the address of the DCB you wish to view, because TEST cannot choose a suitable default.

    This subcommand is displaying the DCB in your program.
12. If you wish to see only selected fields of the DCB, use the FIELD operand. In parentheses, you can specify one or more field names (standard DSECT names) that you wish to have displayed.

    LISTDCB also has the PRINT operand.

```
(13) listdeb 7b7a24.

IKJ57652I DEB LOCATED AT 7B7A24
BASIC SECTION
NMSUB  TCBAD   AMLNG  DEBAD   OFLGS  IRBAD   OPATB  QSCNT  FLGS1  RESERVED
01     7B2588  10     000000  C8     000000  0F     00     11     00
NMEXT  USRPG   PRIOR  ECBAD   PROTG/DEBID  DCBAD   EXSCL  APPAD
01     000000  FF     000000  8F           01DF9C  02     7B7A00
IKJ57334I DEB DOES NOT HAVE A DIRECT ACCESS SECTION+
TEST

(14) equate outdeb 7b7a24.

      TEST

(15) listdeb outdeb

IKJ57652I DEB LOCATED AT 7B7A24
BASIC SECTION
NMSUB  TCBAD   AMLNG  DEBAD   OFLGS  IRBAD   OPATB  QSCNT  FLGS1  RESERVED
01     7B2588  10     000000  C8     000000  0F     00     11     00
NMEXT  USRPG   PRIOR  ECBAD   PROTG/DEBID  DCBAD   EXSCL  APPAD
01     000000  FF     000000  8F           01DF9C  02     7B7A00
IKJ57334I DEB DOES NOT HAVE A DIRECT ACCESS SECTION+
TEST

(16) drop outdeb

      TEST
```

13. The LISTDEB subcommand is similar to the LISTDCB command, in that you must specify the address of the DEB that you wish to view.

    It also has the optional FIELD and PRINT operands.

14. With the EQUATE subcommand, you can add additional symbols to your TEST session, or you can override the address or attributes of existing symbols.

    The example is equating the symbol OUTDEB to the absolute address of the DEB. Additional operands on EQUATE would allow you to specify the data type, length, and multiplicity of the symbol.

    EQUATE is useful for providing symbolic names to storage locations that are otherwise addressable only via absolute or relative addresses.

15. List the DEB now using the symbolic name from the EQUATE.

16. The DROP subcommand removes symbols added with the EQUATE subcommand. It can remove all symbols, by specifying no operands, or it can remove selective symbols, as in this example.

```
(17) listtcb

IKJ57652I TCB LOCATED AT 7B2588
RBP       PIE       DEB       TIO       CMP       TRN       MSS       PKF
007FE8F0  00000000  007B7A24  007CA000  00000000  4401B798  7FFFDAA8  80

FLGS        LMP  DSP  LLS       JLB       JPQ
0000000001  FF   FF   007C2818  0001AD0C  00000000

GRS
R0        R1        R2        R3
00000001  0001B798  007FE028  7FFFE070
R4        R5        R6        R7
007B2588  00FDD630  007FE8F0  007FE8F0
R8        R9        R10       R11
01DE619E  0001B798  01DE519F  02902108
R12       R13       R14       R15
81DE41A0  02902108  02902270  80FF30BA

FSAB    TCB       TME       JSTCB     NTC       OTC       LTC       IQE
01CFB8  00000000  00000000  007C2E88  00000000  007B2290  00000000  00000000

ECB       TSFLG  STPCT  TSLP  TSDP  PQE/RD    AQE/AE    NSTAE  STABB   TCT
000097E0  00     01     00    00    7FF1451C  00000000  00     7EAFB0  807F9200

USER      NDSP0  NDSP1  NDSP2  NDSP3  RESERVED  JSCB      SSAT      IOBRC
00000000  00     00     40     00     00000000  007B81C4  00FDB938  00000000

RESERVED  EXT1      NDSP4  NDSP5  FLGS6  FLGS7  DAR  RESERVED  SYSCT  STMCT
00000000  00000000  00     00     00     00     00   00        00     00

EXT2      AECB      XSB       BACK      RTWA      NSSP      XLAS      ABCUR
007B26E0  00000000  007FE9C0  007B2290  00000000  00000000  00000000  00

RESERVED  TID  RESERVED  XSCT1  CCPVI  FOE       SWA       STAWA     TCBID
00        00   00        0000   0041   00000000  7F71F238  00000000  E3C3C240

RTM12     ESTAE     RESERVED  SEQNO  AFFN  FBYT1-2
00000000  00000000  7F71E388  0009   FFFF  0000

RESERVED                                ERD       EAE
C0000000000000000000007C1DD800000000  7FF1452C  00000000

TCB EXTENT
TFLG  GTFA    RESERVED  RCMP    EVENT     TQE       CAUF      PERCP     PERCT
00    000000  00        000000  00000000  00000000  00000000  00000000  00000000

TEST
```

17. The LISTTCB subcommand is the last of the specialized LIST subcommands. By default, it displays the current TCB, but by using the ADDR operand (like LISTPSW) you can display any TCB.

    It also has the optional PRINT operand.

```
(18) off

      TEST

(19) go

      IKJ56641I TEMPNAME ENDED DUE TO ERROR
      IKJ56640I SYSTEM ABEND CODE 0C9   REASON CODE 0009
      TEST

(20) where

 1DED2.  LOCATED AT +3A  IN TEMPNAME.TABAVG  UNDER TCB LOCATED AT 7B2588.
TEST

(21) list 4r:7r

       4R  00000000   5R  0000016E   6R  00000018   7R  00000000
      TEST

(22) 7r=f'12'

      TEST

     list 7r
       7R  0000000C
      TEST

(23) go 1ded0.

      TABLE AVERAGE PROGRAM
      IKJ57023I PROGRAM UNDER TEST HAS TERMINATED NORMALLY+
      TEST

(24) list (sum avg)

      SUM  +366
      AVG
           +0  +6
           +4  +30
      TEST
```

Execute the program again. It had stopped at the breakpoint at ADDIT.

18. Remove the breakpoint so that the program will run to completion.
19. GO continues execution. The program does not complete normally. The abend code for the program is 0C9, which indicates that a fixed-point-divide exception occurred (division by zero).

    If the abend information does not appear on your screen as shown, enter a question mark.
20. The WHERE subcommand returns the failing location and the displacement within the CSECT.

    Looking at the source for SAMPLE2, you can see that the failing instruction is really the one prior, the DR. This instruction is two bytes before the “failing location” shown by the WHERE subcommand.
21. Viewing the registers, register 7 contains zero, and the program is dividing by register 7.
22. Put the correct value in register 7 for the divide (the number of elements in the table is 12).

    This provides a temporary “fix” for the problem.
23. Restart the program from the DR instruction.

    The program now runs to normal completion.
24. View the program's result fields.

The example on this page shows how to determine the abend code and the failing instruction if your program under TEST should not complete normally.

```
(25) where +10

 1DEA8.  LOCATED AT +10 IN TEMPNAME.TABAVG UNDER TCB LOCATED AT 7B2588.
TEST

(26) qualify outdcb

      TEST

(27) where +10
 1DFAC.  LOCATED AT +114 IN TEMPNAME.TABAVG UNDER TCB LOCATED AT 7B2588.
TEST

(28) list +10

          +10  004F0000
      TEST

(29) list outdcb+10

      0001DFAC.  004F0000
      TEST

(30) qualify tempname.tabavg

      IKJ57277I QUALIFICATION IS UNDER TCB AT 7B2588
      TEST

(31) where +10

 1DEA8.  LOCATED AT +10 IN TEMPNAME.TABAVG UNDER TCB LOCATED AT 7B2588.
TEST

     end

      READY
```

Earlier in the topic on addressing, it was stated that relative addresses are calculated in relationship to the start of the CSECT currently active. The following examples show you how to alter this "base" address for relative addresses.

25. Find out where +10 is currently located.
26. The QUALIFY subcommand changes the base address for relative addresses. Now any relative addresses will be calculated from the address of OUTDCB, rather than from the start of the CSECT.
27. The address +10 is now at a different absolute location (+114 from the start of the CSECT).
28. View the contents of +10.
29. View the OUTDCB+10; these are now the same location.
30. To return to a base of the CSECT, use QUALIFY again with the qualified address of the CSECT (the load module name, TEMPNAME), followed by a period, then the CSECT name (in this case, TABAVG).
31. Now the relative address +10 is calculated from the CSECT.

Qualified address are useful when you are working with more than one CSECT, or more than one load module in storage. By qualifying a symbolic or relative address with the load module name and the CSECT name, you can alternate between programs, viewing and altering storage.

More TEST Subcommands

So far, this tutorial has shown you some of the subcommands of TEST, what they can do for you, and some of the variations you can use. There are still more versions of these subcommands.

GETMAIN and FREEMAIN

The GETMAIN and FREEMAIN functions of TEST are essentially the same as the system macros. That is, they enable you to acquire additional storage dynamically, and to free it. You must specify the amount of storage and the subpool (default is 0) from which you wish to acquire the storage. On GETMAIN, you can optionally EQUATE the new storage to a symbol so you do not have to remember the absolute address of it.

GETMAIN and FREEMAIN are useful for maintaining a "scratch pad" with copies of data areas that you need while testing, and especially for "scaffolding" when setting up test cases. You can build your own parameters in the acquired area to send to a routine, and see how the program alters these parameters.

LOAD and DELETE

At times, you may wish to bring additional programs into storage, or obtain your own copy of a module that normally resides in the link pack area. If your program attempts to LOAD, LINK, XCTL, or ATTACH another module, the system looks for the module in the following search order sequence:

1. TASKLIB
2. STEPLIB
3. JOBLIB
4. LPA
5. LNKLST

If the module is not in any of these areas, it will not be found. To avoid this, bring the module into virtual storage by using the LOAD subcommand of TEST.

The LOAD command allows you to load a program from a data set. This program will be a fresh copy, placed in your own region, to which you can add breakpoints. (You cannot put breakpoints in LPA-resident modules because they are write-protected.)

DELETE allows you to remove the module from your region.

CALL

If you wish to invoke a program and pass it parameters, without setting up the parameter list yourself, you can use the CALL subcommand of TEST. You can also specify where CALL should return control. TEST will build a parameter list and set register 1 to its address, and it will set register 15 to the calling address, and register 14 to the return address (if you request). When using CALL, you may want to save registers 1, 14, and 15 prior to the CALL, to protect this information.

An alternative to CALL is to obtain storage with GETMAIN, use assignment and COPY to build the parameter list and set the registers, and use GO to branch to the routine.

Testing Programs That Use the Vector Facility

You can display and alter vector registers by using the LIST, LISTVSR, and SETVSR subcommands of TEST and the TEST assignment function.

You can list vector system parameters by using the LISTVP subcommand of TEST.

Use the AT subcommand of TEST to set breakpoints at the vector opcodes in the same way you use it to set breakpoints in other programs.

Examples showing the use of the LIST subcommand to display the contents of vector registers follow:

Subcommand
Function

LIST 1V(*) X
Displays the entire contents of vector register 1 in hexadecimal.

LIST 1V(4) F
Displays the fourth element in vector register 1 in fullword fixed point binary.

LIST 3V(3):3V(20)
Displays elements 3 through 20 in vector register 3 in single precision floating point.

LIST 0V(*):15V(*)
Displays the entire contents of all 16 vector registers in single precision floating point.

LIST 0W(*)
Displays the entire contents of vector register 0 in double precision floating point.

Use the assignment function to modify the contents of the 16 vector registers. Some examples illustrating the use of the assignment function follow:

Subcommand
Function

1V(*)=X'00000000'
Sets the entire contents of vector register 1 to hexadecimal zeros.

1V(10)=F'33'
Sets the tenth element in vector register 1 to decimal 33.

3V(3)=(X'00',X'02')
Sets elements 3 and 4 of vector register 3 to X'00' and X'02', respectively.

0W(1)=D'+33E+2'
Sets the first element of vector registers 0 and 1 to the double precision floating point value of +33E+2.

To display the contents of the vector status register, use the LISTVSR subcommand of TEST. LISTVSR works the same as the LISTPSW subcommand. The following example shows the syntax and output of the LISTVSR subcommand after issuing a RESTORE VSR instruction (VSRRS):

```
listvsr

  VSR LOCATED AT 7FFF9EF8
        RESERVED        VMM    VCT    VIX      VIU       VCH
     00000000 00000000   0    00127  00127   00000000  00000000
  TEST
```

You can use the LIST subcommand and the TEST assignment function to display and modify the vector mask register. Some examples follow:

Subcommand
Function

0M=X'046C471F'
Sets the contents of the vector mask register to X'046C471F'

There is one bit in the vector mask register for each vector element. In this example, the number of vector elements, which is also referred to as the section size, is 32 decimal.

LIST 0M
Displays the contents of the vector mask register in hexadecimal.

You can use the LISTVP subcommand of TEST to display the number of vector elements (vector section size) and the partial sum number. For example:

```
listvp

  IKJ57026I VECTOR SYSTEM PARAMETER
     SECTION SIZE:  00256
     PARTIAL SUM:   00004
  TEST
```

You can use the SETVSR subcommand of TEST to change the vector mask register control mode, update vector count (VCT), update vector interruption index (VIX), and update vector in-use bits (VIU).

The following examples show the use of the SETVSR subcommand.

Subcommand
Function

SETVSR MASK
Changes the vector mask register control mode.

SETVSR NOMASK
Changes the vector mask register control mode.

SETVSR VCT(X'nnnn')
Updates the vector count (VCT), where X'nnnn' is the number of vector elements to process.

SETVSR VIX(X'nnnn')
Updates the vector interruption index (VIX), where X'nnnn' is the vector element to start processing with.

SETVSR VIU(X'nn')
Updates the vector in-use bits (VIU), where X'nn' indicates the active register pairs.

Testing Programs That Use Extended Addressing

You can use the subcommands of TEST to display and modify access registers, and display and modify data in alternate address/data spaces. For information about writing programs that use extended addressing, see *z/OS MVS Programming: Assembler Services Reference ABE-HSP*.

Displaying and modifying access registers

You can display and modify access registers by using the LIST subcommand and the TEST assignment function.

The following examples show the use of the LIST subcommand to display the contents of access registers:

Subcommand
Function

LIST 1A
Displays the contents of access register 1 in hexadecimal.

LIST 4A F
Displays the contents of access register 4 in decimal.

LIST 0A:15A
Displays the contents of access registers 0 through 15 in hexadecimal.

The following examples show the use of the assignment function to modify the contents of the 16 access registers:

Subcommand
Function

7A=X'00000000'
Sets the contents of access register 7 to zeros.

2A=F'234'
Sets the contents of access register 2 to decimal 234.

5A=(X'00',X'11')
Sets the contents of access registers 5 and 6 to zero and X'11', respectively.

Displaying and modifying data in alternate address spaces

You can display and modify the contents of storage in alternate address/data spaces by using the AR and ALET keywords on the LIST subcommand and the TEST assignment function. Use the AR keyword to specify the number of the access register to be used to reference data in an alternate address/data space. Use the ALET keyword to specify from one to eight hexadecimal characters that indicate which address/data space is to be referenced.

The following examples show the use of the LIST subcommand to display data in an alternate address/data space.

Subcommand
Function

LIST 4AD8. AR(4)
Displays the contents of the storage at address 4AD8 in the address/data space indicated by access register 4.

LIST 2R? AR(8)
Displays the contents of the storage at the location pointed to by general register 2 in the address/data space indicated by access register 8.

LIST 1000. ALET(A624)
Displays the contents of the storage at location 1000 in the address/data space indicated by the ALET value A624.

The following examples show how to modify the contents of storage in alternate address/data spaces:

Subcommand
Function

5558.=X'0002' AR(4)
Sets two bytes of storage at address 5558 in the address/data space indicated by access register 4, to the value X'0002'.

9R?=F'100' ALET(9E00)
Sets four bytes of storage, located at the address pointed to by general register 9 in the address/data space indicated by the ALET value 9E00, to a value of decimal 100.

Copying Data to and from Alternate Address Spaces

You can use the ARFROM and ALETFROM keywords on the COPY subcommand to copy data from alternate address/data spaces to other locations. Similarly, you can use the ARTO and ALETTO keywords to copy data to storage in alternate address/data spaces. The ARTO and ARFROM keywords allow you to specify the number of the access register to be used to reference data in an alternate address/data space. Use the ALETTO and ALETFROM keywords to specify from one to eight hexadecimal characters that indicate which address/data space is to be referenced.

The following examples show the use of the COPY subcommand to copy data to and from alternate address/data spaces.

Subcommand
Function

COPY 13R? 1000. ARTO(5) LENGTH(72)
Copies 72 bytes starting at the location pointed to by register 13 to location 1000 in the address/data space indicated by access register 5.

COPY 1R? A080. ARFROM(1) ALETTO(40C3A)
Copies 4 bytes of storage at the location pointed to by register 1 in the address/data space indicated by access register 1 to location A080 in the address/data space indicated by the ALET value 40C3A.

Providing Symbolic Names for Locations in Alternate Address Spaces

You can use the AR and ALET keywords on the EQUATE subcommand to establish symbols to refer to storage locations in alternate address/data spaces. The AR keyword allows you to specify the number of the access register to be used to reference data in an alternate address/data space. Use the ALET keyword to specify from one to eight hexadecimal characters that indicate which address/data space is to be referenced.

The following examples show the use of the EQUATE subcommand to associate a symbolic name with a location in an alternate address/data space.

Subcommand
Function

EQUATE X 2290. AR(5)
Symbol X refers to the address of location 2290 in the address/data space indicated by access register 5.

EQUATE Y 1R? ALET(4AC2)
Symbol Y refers to the address pointed to by general register 1 in the address/data space indicated by the ALET value 4AC2.

Example programs for the TEST tutorial

The following two programs are used in the TEST tutorial. The first program is used early in the tutorial and must be placed in a sequential data set named '*prefix.*SAMPLE1.ASM'. The second program is used later in the tutorial and must be placed in a sequential data set named '*prefix.*SAMPLE2.ASM'.

Source for first sample program

```
*************************************************************
**                                                         *
**  This is a sample assembler language program that is    *
**  used with the TEST tutorial in the publication,        *
**  TSO/E Programming Guide.                               *
**                                                         *
*************************************************************
SAMP1    CSECT
         STM   14,12,12(13)
         BALR  12,0
         USING *,12
         ST    13,SAVE+4
         LA    15,SAVE
         ST    15,8(13)
         LR    13,15
STOP1    LH    3,HALFCON
STOP2    A     3,FULLCON
STOP3    ST    3,HEXCON
         L     13,4(13)
         LM    14,12,12(13)
         BR    14
SAVE     DC    18F'0'
ADCON    DC    A(SAVE)
FULLCON  DC    F'-1'
HEXCON   DC    XL4'FD38'
HALFCON  DC    H'32'
CHARCON  DC    CL10'TEST EXAMP'
PACKCON  DC    PL4'25'
BINCON   DC    B'10101100'
         END   SAMP1
```

Listing for first sample program

```
         External Symbol Dictionary
Symbol   Type   Id     Address  Length   Owner Id Flags Alias-of          HLASM R6.0  2008/09/23 11.08
SAMP1     SD 00000001 00000000 00000091             00

LOC  OBJECT CODE      ADDR1 ADDR2  STMT SOURCE STATEMENT                 HLASM R6.0  2008/09/23 11.08
                                      1 *************************************************************
                                      2 **                                                         *
                                      3 **  This is a sample assembler language program that is    *
                                      4 **  used with the TEST tutorial in the publication,        *
                                      5 **  TSO/E Programming Guide.                               *
                                      6 **                                                         *
                                      7 *************************************************************
000000                00000 00091     8 SAMP1    CSECT
000000 90EC D00C            0000C     9          STM   14,12,12(13)
000004 05C0                          10          BALR  12,0
                 R:C  00006          11          USING *,12
000006 50D0 C02A            00030    12          ST    13,SAVE+4
00000A 41F0 C026            0002C    13          LA    15,SAVE
00000E 50FD 0008            00008    14          ST    15,8(13)
000012 18DF                          15          LR    13,15
000014 4830 C07A            00080    16 STOP1    LH    3,HALFCON
000018 5A30 C072            00078    17 STOP2    A     3,FULLCON
00001C 5030 C076            0007C    18 STOP3    ST    3,HEXCON
000020 58DD 0004            00004    19          L     13,4(13)
000024 98EC D00C            0000C    20          LM    14,12,12(13)
000028 07FE                          21          BR    14
00002A 0000
00002C 0000000000000000              22 SAVE     DC    18F'0'
000074 0000002C                      23 ADCON    DC    A(SAVE)
000078 FFFFFFFF                      24 FULLCON  DC    F'-1'
00007C 0000FD38                      25 HEXCON   DC    XL4'FD38'
000080 0020                          26 HALFCON  DC    H'32'
000082 E3C5E2E340C5E7C1              27 CHARCON  DC    CL10'TEST EXAMP'
```

```
00008C 0000025C                     28 PACKCON  DC    PL4'25'
000090 AC                           29 BINCON   DC    B'10101100'
000000                              30          END   SAMP1
                                                Relocation Dictionary
Pos.Id   Rel.Id   Address   Type  Action                                  HLASM R6.0  2008/09/23 11.08
00000001 00000001 00000074   A 4     +
                                  Ordinary Symbol and Literal Cross Reference
Symbol   Length   Value     Id    R Type Asm  Program   Defn References  HLASM R6.0  2008/09/23 11.08
FULLCON       4 00000078 00000001     F  F                24   17
HALFCON       2 00000080 00000001     H  H                26   16
HEXCON        4 0000007C 00000001     X  X                25   18M
SAMP1         1 00000000 00000001     J                    8   30
SAVE          4 0000002C 00000001     F  F                22   12M   13    23
                                  Unreferenced Symbols Defined in CSECTs
Defn  Symbol                                                             HLASM R6.0  2008/09/23 11.08
  23  ADCON
  29  BINCON
  27  CHARCON
  28  PACKCON
  16  STOP1
  17  STOP2
  18  STOP3
                                                       Using Map
                                                                         HLASM R6.0  2008/09/23 11.08
Stmt  -----Location----- Action ----------------Using----------------- Reg Max     Last Label and Using Text
        Count      Id           Type          Value    Range     Id        Disp    Stmt
  11  00000006  00000001 USING  ORDINARY    00000006 00001000 00000001  12 0007A     18 *,12
                                    General Purpose Register Cross Reference
Register  References (M=modified, B=branch, U=USING, D=DROP, N=index)    HLASM R6.0  2008/09/23 11.08
   0(0)       9    20M
   1(1)       9    20M
   2(2)       9    20M
   3(3)       9    16M   17M   18    20M
   4(4)       9    20M
   5(5)       9    20M
   6(6)       9    20M
   7(7)       9    20M
   8(8)       9    20M
   9(9)       9    20M
  10(A)       9    20M
  11(B)       9    20M
  12(C)       9    10M   11U   20M
  13(D)       9    12    14N   15M   19M   19N   20
  14(E)       9    20M   21B
  15(F)       9    13M   14    15    20M
                                  Diagnostic Cross Reference and Assembler Summary
                                                                         HLASM R6.0  2008/09/23 11.08
No Statements Flagged in this Assembly
HIGH LEVEL ASSEMBLER, 5696-234, RELEASE 6.0, PTF UK37157
SYSTEM: z/OS 01.10.00              JOBNAME: MKASPERN    STEPNAME: SAMPLE1    PROCSTEP: (NOPROC)
Data Sets Allocated for this Assembly
 Con DDname   Data Set Name                                Volume  Member
  P1 SYSIN    MKASPER.SAMPLE1.SAM                         SL8D18
  L1          SYS1.MODGEN                                 PRIPKR
  L2          SYS1.MACLIB                                 PRIPKR
     SYSLIN   NULLFILE
     SYSPRINT SYS08267.T110803.RA000.MKASPERN.ASMPRT.H01   SMSSC5
     SYSPUNCH SYS08267.T110803.RA000.MKASPERN.OBJKEEP.H01

1533936K allocated to Buffer Pool       Storage required     228K
     30  Primary Input Records Read           0 Library Records Read                 0 Work File Reads
      0  ASMAOPT Records Read               188 Primary Print Records Written        0 Work File Writes
      6  Object Records Written               0 ADATA Records Written
Assembly Start Time: 11.08.03 Stop Time: 11.08.03 Processor Time: 00.00.00.0004
Return Code 000
```

Source for second sample program

```
*************************************************************
**                                                         *
**  This is a sample assembler language program that is    *
**  used with the TEST tutorial in the publication,        *
**  TSO/E Programming Guide.                               *
**                                                         *
** NOTE: There is an error in this program.  It is intended *
**       to be located and fixed in the TEST tutorial.     *
*************************************************************
TABAVG   CSECT
```

```
         STM   14,12,12(13)
         BALR  12,0
         USING *,12
         ST    13,SAVE+4
         LA    15,SAVE
         ST    15,8(13)
         LR    13,15
         OPEN  (OUTDCB,(OUTPUT))
         L     7,TABSIZE
         SR    6,6
         SR    5,5
ADDIT    AH    5,TAB(6)
         LA    6,2(6)
         BCT   7,ADDIT
         ST    5,SUM
         SR    4,4
         DR    4,7
         STM   4,5,AVG
         PUT   OUTDCB,OUTMSG
         CLOSE (OUTDCB)
         L     13,4(13)
         LM    14,12,12(13)
         BR    14
SAVE     DC    18F'0'
TABSIZE  DC    F'12'
AVG      DC    2F'0'
SUM      DC    F'0'
TAB      DC    H'31'
         DC    H'29'
         DC    H'31'
         DC    H'30'
         DC    H'31'
         DC    H'30'
         DC    H'31'
         DC    H'31'
         DC    H'30'
         DC    H'31'
         DC    H'30'
         DC    H'31'
OUTMSG   DC    CL50'TABLE AVERAGE PROGRAM                          '
OUTDCB   DCB   DDNAME=OUTDD,LRECL=50,BLKSIZE=50,RECFM=F,              *
               MACRF=(PM),DSORG=PS
         END   TABAVG
```

Listing for second sample program

```
          External Symbol Dictionary
 Symbol   Type   Id     Address  Length   Owner Id Flags Alias-of                HLASM R6.0  2008/09/23 11.25
 TABAVG    SD 00000001 00000000 00000168             00

 LOC  OBJECT CODE      ADDR1 ADDR2  STMT SOURCE STATEMENT                        HLASM R6.0  2008/09/23 11.25
                                      1 ************************************************************
                                      2 **                                                        *
                                      3 **  This is a sample assembler language program that is      *
                                      4 **  used with the TEST tutorial in the publication,         *
                                      5 **  TSO/E Programming Guide.                                *
                                      6 **                                                        *
                                      7 ** NOTE: There is an error in this program.  It is intended  *
                                      8 **       to be located and fixed in the TEST tutorial.      *
                                      9 ************************************************************
 000000                 00000 00168    10 TABAVG   CSECT
 000000 90EC D00C             0000C    11          STM   14,12,12(13)
 000004 05C0                           12          BALR  12,0
                   R:C  00006          13          USING *,12
 000006 50D0 C062             00068    14          ST    13,SAVE+4
 00000A 41F0 C05E             00064    15          LA    15,SAVE
 00000E 50FD 0008             00008    16          ST    15,8(13)
 000012 18DF                           17          LR    13,15
                                       18          OPEN  (OUTDCB,(OUTPUT))
 000014                                19+         CNOP  0,4                        ALIGN LIST TO FULLWORD
 000014 4510 C016             0001C    20+         BAL   1,*+8                      LOAD REG1 W/LIST ADDR. @L2A
 000018 8F                             21+         DC    AL1(143)                   OPTION BYTE
 000019 000108                         22+         DC    AL3(OUTDCB)                DCB ADDRESS
 00001C 0A13                           23+         SVC   19                         ISSUE OPEN SVC
 00001E 5870 C0A6             000AC    24          L     7,TABSIZE
 000022 1B66                           25          SR    6,6
 000024 1B55                           26          SR    5,5
```

```
000026 4A56 C0B6            000BC    27 ADDIT    AH    5,TAB(6)
00002A 4166 0002            00002    28          LA    6,2(6)
00002E 4670 C020            00026    29          BCT   7,ADDIT
000032 5050 C0B2            000B8    30          ST    5,SUM
000036 1B44                          31          SR    4,4
000038 1D47                          32          DR    4,7
00003A 9045 C0AA            000B0    33          STM   4,5,AVG
                                     34          PUT   OUTDCB,OUTMSG
00003E 4110 C102            00108    36+         LA    1,OUTDCB                      LOAD PARAMETER REG 1
000042 4100 C0CE            000D4    37+         LA    0,OUTMSG                      LOAD PARAMETER REG 0
000046 1FFF                          38+         SLR   15,15                   CLEAR REGISTER             @L1A
000048 BFF7 1031            00031    39+         ICM   15,7,49(1)              LOAD PUT ROUTINE ADDR      @L1C
00004C 05EF                          40+         BALR  14,15                   LINK TO PUT ROUTINE
                                     41          CLOSE (OUTDCB)
00004E 0700                          42+         CNOP  0,4                           ALIGN LIST TO FULLWORD
000050 4510 C052            00058    43+         BAL   1,*+8                     LOAD REG1 W/LIST ADDR. @L2A
000054 80                            44+         DC    AL1(128)                      OPTION BYTE
000055 000108                        45+         DC    AL3(OUTDCB)                   DCB ADDRESS
000058 0A14                          46+         SVC   20                            ISSUE CLOSE SVC
00005A 58DD 0004            00004    47          L     13,4(13)
00005E 98EC D00C            0000C    48          LM    14,12,12(13)
000062 07FE                          49          BR    14
000064 0000000000000000              50 SAVE     DC    18F'0'
0000AC 0000000C                      51 TABSIZE  DC    F'12'
0000B0 0000000000000000              52 AVG      DC    2F'0'
0000B8 00000000                      53 SUM      DC    F'0'
0000BC 001F                          54 TAB      DC    H'31'
0000BE 001D                          55          DC    H'29'
0000C0 001F                          56          DC    H'31'

LOC  OBJECT CODE    ADDR1 ADDR2    STMT SOURCE   STATEMENT
0000C2 001E                          57          DC    H'30'
0000C4 001F                          58          DC    H'31'
0000C6 001E                          59          DC    H'30'
0000C8 001F                          60          DC    H'31'
0000CA 001F                          61          DC    H'31'
0000CC 001E                          62          DC    H'30'
0000CE 001F                          63          DC    H'31'
0000D0 001E                          64          DC    H'30'
0000D2 001F                          65          DC    H'31'
0000D4 E3C1C2D3C540C1E5              66 OUTMSG   DC    CL50'TABLE AVERAGE PROGRAM                                '
                                     67 OUTDCB   DCB   DDNAME=OUTDD,LRECL=50,BLKSIZE=50,RECFM=F,                *
                                                       MACRF=(PM),DSORG=PS
                                     70+*                      DATA CONTROL BLOCK
                                     71+*
000106 0000
000108                               72+OUTDCB   DC    0F'0'           ORIGIN ON WORD BOUNDARY
                                     73+*                      DIRECT ACCESS DEVICE INTERFACE
000108 0000000000000000              74+         DC    BL16'0'         FDAD, DVTBL
000118 00000000                      75+         DC    A(0)            KEYLEN, DEVT, TRBAL
                                     76+*                      COMMON ACCESS METHOD INTERFACE
00011C 00                            77+         DC    AL1(0)          BUFNO, NUMBER OF BUFFERS
00011D 000001                        78+         DC    AL3(1)          BUFCB, BUFFER POOL CONTROL BLOCK
000120 0000                          79+         DC    AL2(0)          BUFL, BUFFER LENGTH
000122 4000                          80+         DC    BL2'0100000000000000' DSORG, DATA SET ORGANIZATION
000124 00000001                      81+         DC    A(1)            IOBAD FOR EXCP OR RESERVED
                                     82+*                      FOUNDATION EXTENSION
000128 00                            83+         DC    BL1'00000000'   BFTEK, BFALN, DCBE INDICATORS
000129 000001                        84+         DC    AL3(1)          EODAD (END OF DATA ROUTINE ADDRESS)
00012C 80                            85+         DC    BL1'10000000'   RECFM (RECORD FORMAT)
00012D 000000                        86+         DC    AL3(0)          EXLST (EXIT LIST ADDRESS)
                                     87+*                      FOUNDATION BLOCK
000130 D6E4E3C4C4404040              88+         DC    CL8'OUTDD'      DDNAME
000138 02                            89+         DC    BL1'00000010'   OFLGS (OPEN FLAGS)
000139 00                            90+         DC    BL1'00000000'   IFLGS (IOS FLAGS)
00013A 0050                          91+         DC    BL2'0000000001010000' MACR (MACRO FORMAT)
                                     92+*                      BSAM-BPAM-QSAM INTERFACE
00013C 00                            93+         DC    BL1'00000000'   OPTCD, OPTION CODES
00013D 000001                        94+         DC    AL3(1)          CHECK OR INTERNAL QSAM SYNCHRONIZING RTN.
000140 00000001                      95+         DC    A(1)            SYNAD, SYNCHRONOUS ERROR RTN. (3 BYTES)
000144 0000                          96+         DC    H'0'            INTERNAL ACCESS METHOD FLAGS
000146 0032                          97+         DC    AL2(50)         BLKSIZE, BLOCK SIZE
000148 00000000                      98+         DC    F'0'            INTERNAL ACCESS METHOD FLAGS
00014C 00000001                      99+         DC    A(1)            INTERNAL ACCESS METHOD USE
                                    100+*                     QSAM INTERFACE
000150 00000001                     101+         DC    A(1)            EOBAD
000154 00000001                     102+         DC    A(1)            RECAD
000158 0000                         103+         DC    H'0'            QSWS (FLAGS) AND EITHER DIRCT OR BUFOFF
00015A 0032                         104+         DC    AL2(50)         LRECL
```

Example programs for the TEST tutorial

```
00015C 00                          105+         DC    BL1'00000000'  EROPT, ERROR OPTION
00015D 000001                      106+         DC    AL3(1)         CNTRL
000160 00000000                    107+         DC    H'0,0'         RESERVED AND PRECL
000164 00000001                    108+         DC    A(1)           EOB, INTERNAL ACCESS METHOD FIELD
000000                             109          END   TABAVG
                                                                 Relocation Dictionary
Pos.Id   Rel.Id   Address  Type  Action                                        HLASM R6.0  2008/09/23 11.25
00000001 00000001 00000019   A 3     +
00000001 00000001 00000055   A 3     +
                             Ordinary Symbol and Literal Cross Reference
Symbol   Length   Value     Id    R Type Asm  Program   Defn References           HLASM R6.0  2008/09/23 11.25
ADDIT         4 00000026 00000001     I                   27   29B
AVG           4 000000B0 00000001     F  F                52   33M
OUTDCB        4 00000108 00000001     F  F                72   22    36    45
OUTMSG       50 000000D4 00000001     C  C                66   37
SAVE          4 00000064 00000001     F  F                50   14M   15
SUM           4 000000B8 00000001     F  F                53   30M
TAB           2 000000BC 00000001     H  H                54   27
TABAVG        1 00000000 00000001     J                   10  109
TABSIZE       4 000000AC 00000001     F  F                51   24
                                                  Using Map
                                                                              HLASM R6.0  2008/09/23 11.25
Stmt  -----Location----- Action ----------------Using----------------- Reg Max     Last Label and Using Text
        Count      Id           Type          Value    Range     Id        Disp    Stmt
  13  00000006  00000001 USING  ORDINARY    00000006 00001000 00000001  12 00102     43 *,12
                                  General Purpose Register Cross Reference
Register  References (M=modified, B=branch, U=USING, D=DROP, N=index)             HLASM R6.0  2008/09/23 11.25
   0(0)      11    37M   48M
   1(1)      11    20M   36M   39    43M   48M
   2(2)      11    48M
   3(3)      11    48M
   4(4)      11    31M   31    32M   33    48M
   5(5)      11    26M   26    27M   30    32M   33    48M
   6(6)      11    25M   25    27N   28M   28N   48M
   7(7)      11    24M   29M   32    48M
   8(8)      11    48M
   9(9)      11    48M
  10(A)      11    48M
  11(B)      11    48M
  12(C)      11    12M   13U   48M
  13(D)      11    14    16N   17M   47M   47N   48
  14(E)      11    40M   48M   49B
  15(F)      11    15M   16    17    38M   38    39M   40B   48M
                                  Diagnostic Cross Reference and Assembler Summary
                                                                              HLASM R6.0  2008/09/23 11.25
No Statements Flagged in this Assembly
HIGH LEVEL ASSEMBLER, 5696-234, RELEASE 6.0, PTF UK37157
SYSTEM: z/OS 01.10.00              JOBNAME: MKASPERO    STEPNAME: SAMPLE2    PROCSTEP: (NOPROC)
Data Sets Allocated for this Assembly
Con DDname   Data Set Name                                Volume  Member
 P1 SYSIN    MKASPER.SAMPLE2.SAM                         SL6294
 L1          SYS1.MODGEN                                 PRIPKR
 L2          SYS1.MACLIB                                 PRIPKR
    SYSLIN   NULLFILE
    SYSPRINT SYS08267.T112528.RA000.MKASPERO.ASMPRT.H01   SMSSC4
    SYSPUNCH SYS08267.T112528.RA000.MKASPERO.OBJKEEP.H01

1533756K allocated to Buffer Pool       Storage required     500K
     53  Primary Input Records Read        2675 Library Records Read                 0 Work File Reads
      0  ASMAOPT Records Read               268 Primary Print Records Written        0 Work File Writes
     10  Object Records Written               0 ADATA Records Written
Assembly Start Time: 11.25.29 Stop Time: 11.25.29 Processor Time: 00.00.00.0116
Return Code 000
```

Part 4. Appendixes

Appendix. Accessibility

Accessible publications for this product are offered through the z/OS Information Center, which is available at www.ibm.com/systems/z/os/zos/bkserv/.

If you experience difficulty with the accessibility of any z/OS information, please send a detailed message to mhvrcfs@us.ibm.com or to the following mailing address:

IBM Corporation
Attention: MHVRCFS Reader Comments
Department H6MA, Building 707
2455 South Road
Poughkeepsie, NY 12601-5400
USA

Accessibility features

Accessibility features help a user who has a physical disability, such as restricted mobility or limited vision, to use software products successfully. The major accessibility features in z/OS enable users to:

- Use assistive technologies such as screen readers and screen magnifier software
- Operate specific or equivalent features using only the keyboard
- Customize display attributes such as color, contrast, and font size.

Using assistive technologies

Assistive technology products, such as screen readers, function with the user interfaces found in z/OS. Consult the assistive technology documentation for specific information when using such products to access z/OS interfaces.

Keyboard navigation of the user interface

Users can access z/OS user interfaces using TSO/E or ISPF. Refer to *z/OS TSO/E Primer*, *z/OS TSO/E User's Guide*, and *z/OS ISPF User's Guide Vol I* for information about accessing TSO/E and ISPF interfaces. These guides describe how to use TSO/E and ISPF, including the use of keyboard shortcuts or function keys (PF keys). Each guide includes the default settings for the PF keys and explains how to modify their functions.

Dotted decimal syntax diagrams

Syntax diagrams are provided in dotted decimal format for users accessing the z/OS Information Center using a screen reader. In dotted decimal format, each syntax element is written on a separate line. If two or more syntax elements are always present together (or always absent together), they can appear on the same line, because they can be considered as a single compound syntax element.

Each line starts with a dotted decimal number; for example, 3 or 3.1 or 3.1.1. To hear these numbers correctly, make sure that your screen reader is set to read out punctuation. All the syntax elements that have the same dotted decimal number (for example, all the syntax elements that have the number 3.1) are mutually

exclusive alternatives. If you hear the lines 3.1 USERID and 3.1 SYSTEMID, you know that your syntax can include either USERID or SYSTEMID, but not both.

The dotted decimal numbering level denotes the level of nesting. For example, if a syntax element with dotted decimal number 3 is followed by a series of syntax elements with dotted decimal number 3.1, all the syntax elements numbered 3.1 are subordinate to the syntax element numbered 3.

Certain words and symbols are used next to the dotted decimal numbers to add information about the syntax elements. Occasionally, these words and symbols might occur at the beginning of the element itself. For ease of identification, if the word or symbol is a part of the syntax element, it is preceded by the backslash (\) character. The * symbol can be used next to a dotted decimal number to indicate that the syntax element repeats. For example, syntax element *FILE with dotted decimal number 3 is given the format 3 * FILE. Format 3* FILE indicates that syntax element FILE repeats. Format 3* * FILE indicates that syntax element * FILE repeats.

Characters such as commas, which are used to separate a string of syntax elements, are shown in the syntax just before the items they separate. These characters can appear on the same line as each item, or on a separate line with the same dotted decimal number as the relevant items. The line can also show another symbol giving information about the syntax elements. For example, the lines 5.1*, 5.1 LASTRUN, and 5.1 DELETE mean that if you use more than one of the LASTRUN and DELETE syntax elements, the elements must be separated by a comma. If no separator is given, assume that you use a blank to separate each syntax element.

If a syntax element is preceded by the % symbol, this indicates a reference that is defined elsewhere. The string following the % symbol is the name of a syntax fragment rather than a literal. For example, the line 2.1 %OP1 means that you should refer to separate syntax fragment OP1.

The following words and symbols are used next to the dotted decimal numbers:

- ? means an optional syntax element. A dotted decimal number followed by the ? symbol indicates that all the syntax elements with a corresponding dotted decimal number, and any subordinate syntax elements, are optional. If there is only one syntax element with a dotted decimal number, the ? symbol is displayed on the same line as the syntax element, (for example 5? NOTIFY). If there is more than one syntax element with a dotted decimal number, the ? symbol is displayed on a line by itself, followed by the syntax elements that are optional. For example, if you hear the lines 5 ?, 5 NOTIFY, and 5 UPDATE, you know that syntax elements NOTIFY and UPDATE are optional; that is, you can choose one or none of them. The ? symbol is equivalent to a bypass line in a railroad diagram.
- ! means a default syntax element. A dotted decimal number followed by the ! symbol and a syntax element indicates that the syntax element is the default option for all syntax elements that share the same dotted decimal number. Only one of the syntax elements that share the same dotted decimal number can specify a ! symbol. For example, if you hear the lines 2? FILE, 2.1! (KEEP), and 2.1 (DELETE), you know that (KEEP) is the default option for the FILE keyword. In this example, if you include the FILE keyword but do not specify an option, default option KEEP will be applied. A default option also applies to the next higher dotted decimal number. In this example, if the FILE keyword is omitted, default FILE(KEEP) is used. However, if you hear the lines 2? FILE, 2.1, 2.1.1!

(KEEP), and 2.1.1 (DELETE), the default option KEEP only applies to the next higher dotted decimal number, 2.1 (which does not have an associated keyword), and does not apply to 2? FILE. Nothing is used if the keyword FILE is omitted.

- * means a syntax element that can be repeated 0 or more times. A dotted decimal number followed by the * symbol indicates that this syntax element can be used zero or more times; that is, it is optional and can be repeated. For example, if you hear the line 5.1* data area, you know that you can include one data area, more than one data area, or no data area. If you hear the lines 3*, 3 HOST, and 3 STATE, you know that you can include HOST, STATE, both together, or nothing.

  Note:

  1. If a dotted decimal number has an asterisk (*) next to it and there is only one item with that dotted decimal number, you can repeat that same item more than once.
  2. If a dotted decimal number has an asterisk next to it and several items have that dotted decimal number, you can use more than one item from the list, but you cannot use the items more than once each. In the previous example, you could write HOST STATE, but you could not write HOST HOST.
  3. The * symbol is equivalent to a loop-back line in a railroad syntax diagram.

- + means a syntax element that must be included one or more times. A dotted decimal number followed by the + symbol indicates that this syntax element must be included one or more times; that is, it must be included at least once and can be repeated. For example, if you hear the line 6.1+ data area, you must include at least one data area. If you hear the lines 2+, 2 HOST, and 2 STATE, you know that you must include HOST, STATE, or both. Similar to the * symbol, the + symbol can only repeat a particular item if it is the only item with that dotted decimal number. The + symbol, like the * symbol, is equivalent to a loop-back line in a railroad syntax diagram.

Notices

This information was developed for products and services offered in the U.S.A. or elsewhere.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan, Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

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

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

Site Counsel
IBM Corporation
2455 South Road
Poughkeepsie, NY 12601-5400
USA

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

COPYRIGHT LICENSE:

This information might contain sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Policy for unsupported hardware

Various z/OS elements, such as DFSMS, HCD, JES2, JES3, and MVS, contain code that supports specific hardware servers or devices. In some cases, this device-related element support remains in the product even after the hardware devices pass their announced End of Service date. z/OS may continue to service element code; however, it will not provide service related to unsupported hardware devices. Software problems related to these devices will not be accepted

for service, and current service activity will cease if a problem is determined to be associated with out-of-support devices. In such cases, fixes will not be issued.

Minimum supported hardware

The minimum supported hardware for z/OS releases identified in z/OS announcements can subsequently change when service for particular servers or devices is withdrawn. Likewise, the levels of other software products supported on a particular release of z/OS are subject to the service support lifecycle of those products. Therefore, z/OS and its product publications (for example, panels, samples, messages, and product documentation) can include references to hardware and software that is no longer supported.

- For information about software support lifecycle, see: http://www.ibm.com/software/support/systemsz/lifecycle/
- For information about currently-supported IBM hardware, contact your IBM representative.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at www.ibm.com/legal/copytrade.shtml (http://www.ibm.com/legal/copytrade.shtml).

Index

Special characters

Numerics

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

Product Number: 5650-ZOS

Printed in USA

SA32-0981-00